Alessandro Aldini   Roberto Gorrieri
Fabio Martinelli (Eds.)

Foundations of Security Analysis and Design III

FOSAD 2004/2005 Tutorial Lectures

Springer

Volume Editors

Alessandro Aldini
Università degli Studi di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
E-mail: aldini@sti.uniurb.it

Roberto Gorrieri
Università degli Studi di Bologna
Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: gorrieri@cs.unibo.it

Fabio Martinelli
Istituto di Informatica e Telematica - IIT
National Research Council - C.N.R., Pisa Research Area
Via G. Moruzzi 1, 56100 Pisa, Italy
E-mail: Fabio.Martinelli@iit.cnr.it

Library of Congress Control Number: 2005931798

CR Subject Classification (1998): D.4.6, C.2, K.6.5, K.4, D.3, F.3, E.3

ISSN	0302-9743
ISBN-10	3-540-28955-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-28955-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11554578 06/3142 5 4 3 2 1 0

Preface

The increasing relevance of security to real-life applications, such as electronic commerce and Internet banking, is attested by the fast-growing number of research groups, events, conferences, and summer schools that address the study of foundations for the analysis and the design of security aspects. The "International School on Foundations of Security Analysis and Design" (FOSAD, see http://www.sti.uniurb.it/events/fosad/) has been one of the foremost events established with the goal of disseminating knowledge in this critical area, especially for young researchers approaching the field and graduate students coming from less-favoured and non-leading countries.

The FOSAD school is held annually at the Residential Centre of Bertinoro (http://www.ceub.it/), in the fascinating setting of a former convent and episcopal fortress that has been transformed into a modern conference facility with computing services and Internet access. Since the first school, in 2000, FOSAD has attracted more than 250 participants and 50 lecturers from all over the world. A collection of tutorial lectures from FOSAD 2000 was published in Springer's LNCS volume 2171. Some of the tutorials given at the two successive schools (FOSAD 2001 and 2002) are gathered in a second volume, LNCS 2946. To continue this tradition, the present volume collects a set of tutorials from the fourth FOSAD, held in 2004, and from FOSAD 2005.

The opening paper by Michael Backes, Birgit Pfitzmann, and Michael Waidner, reports on the integration between the classical Dolev-Yao model of security and the computational view of cryptography. In particular, the authors present an idealized cryptographic library that extends the applicability of the Dolev-Yao model for automated proofs of cryptographic protocols to provably secure cryptographic implementations. Jan Jürjens gives an overview of UMLsec, an extension of the Unified Modelling Language that allows the expression of security-relevant information within the diagrams in a system specification. François Koeune and François-Xavier Standaert present a survey on implementation-specific attacks, which attempt to exploit the physical constraints of any real-life cryptographic device (running time, power consumption, ...) to expose the device's secrets. The authors provide a tutorial on this subject, overviewing the main kinds of attacks and highlighting their underlying principles. Riccardo Focardi's paper presents the basics of authentication protocols and illustrates a specific technique for statically analyzing protocol specifications. The technique works in the presence of both malicious outsiders and compromized insiders, with no limitation on the number of parallel sessions.

Gilles Barthe and Guillaume Dufay illustrate some applications of formal methods to increase the reliability of smartcards and trusted personal devices, with respect to both platform correctness and applet validation. Their paper focuses on devices that embed Java Virtual Machines or their variants, in particular Java Card Virtual Machines. Elisa Bertino, Ji-Won Byun, and Ninghui Li deal with various aspects of privacy-preserving data management systems. In particular, they focus on database management systems that are able to enforce privacy promises encoded in privacy languages such as P3P. Herve Debar and Jouni Viinikka's paper covers intrusion detection and security information management technologies, focusing on data sources and analysis techniques. To conclude, Fabio Massacci, Paolo Giorgini, and Nicola Zannone review the state of the art in security requirements engineering and discuss their approach to modelling and analyzing security, the Secure Tropos methodology.

We think that this tutorial book offers an interesting view of what is going on worldwide at present in the security field. We would like to thank all the institutions that have promoted and founded this school and, in particular, the IFIP Working Group on "Theoretical Foundations of Security Analysis and Design" (http://www.dsi.unive.it/IFIPWG1_7/), which was established to promote research and education in security-related issues. FOSAD 2005 was sponsored by CNR-IIT, Create-Net, and the Università di Bologna, and has been supported by EATCS-IT, EEF, and the ERCIM Working Group on Security and Trust Management (http://www.iit.cnr.it/STM-WG/). Finally, we also wish to thank the whole staff of the University Residential Centre of Bertinoro for the organizational and administrative support.

June 2005

Alessandro Aldini
Roberto Gorrieri
Fabio Martinelli

Table of Contents

Justifying a Dolev-Yao Model Under Active Attacks*

Michael Backes, Birgit Pfitzmann, and Michael Waidner

IBM Zurich Research Lab
{mbc, bpf, wmi}@zurich.ibm.com

Abstract. We present the first idealized cryptographic library that can be used like the Dolev-Yao model for automated proofs of cryptographic protocols that use nested cryptographic operations, while coming with a cryptographic implementation that is provably secure under active attacks.

To illustrate the usefulness of the cryptographic library, we present a cryptographically sound security proof of the well-known Needham-Schroeder-Lowe public-key protocol for entity authentication. This protocol was previously only proved over unfounded abstractions from cryptography. We show that the protocol is secure against arbitrary active attacks if it is implemented using standard provably secure cryptographic primitives. Conducting the proof by means of the idealized cryptographic library does not require us to deal with the probabilistic aspects of cryptography, hence the proof is in the scope of current automated proof tools. Besides establishing the cryptographic security of the Needham-Schroeder-Lowe protocol, this exemplifies the potential of this cryptographic library and paves the way for the cryptographically sound verification of security protocols by automated proof tools.

1 Introduction

Many practically relevant cryptographic protocols like SSL/TLS, S/MIME, IPSec, or SET use cryptographic primitives like signature schemes or encryption in a black-box way, while adding many non-cryptographic features. Vulnerabilities have accompanied the design of such protocols ever since early authentication protocols like Needham-Schroeder [59,31], over carefully designed de-facto standards like SSL and PKCS [71,20], up to current widely deployed products like Microsoft Passport [35]. However, proving the security of such protocols has been a very unsatisfactory task for a long time.

One possibility was to take the cryptographic approach. This means reduction proofs between the security of the overall system and the security of the cryptographic primitives, i.e., one shows that if one could break the overall system, one could also break one of the underlying cryptographic primitives with respect to their cryptographic definitions, e.g., adaptive chosen-message security for signature schemes. For authentication protocols, this approach was first used in [19]. In principle, proofs in this approach are as rigorous as typical proofs in mathematics. In practice, however, human

* Parts of this work appeared in Proc. 10th ACM Conference on Computer and Communications Security [12] and Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science [7].

beings are extremely fallible with this type of proofs. This is not due to the cryptography, but to the distributed-systems aspects of the protocols. It is well-known from non-cryptographic distributed systems that many wrong protocols have been published even for very small problems. Hand-made proofs are highly error-prone because following all the different cases how actions of different machines interleave is extremely tedious. Humans tend to take wrong shortcuts and do not want to proof-read such details in proofs by others. If the protocol contains cryptography, this obstacle is even much worse: Already a rigorous definition of the goals gets more complicated, and often not only trace properties (integrity) have to be proven but also secrecy. Further, in principle the complexity-theoretic reduction has to be carried out across all these cases, and it is not at all trivial to do this rigorously. In consequence, there is almost no real cryptographic proof of a larger protocol, and several times supposedly proven, relatively small systems were later broken, e.g., [63,32].

The other possibility was to use formal methods. There one leaves the tedious parts of proofs to machines, i.e., model checkers or automatic theorem provers. This means to code the cryptographic protocols into the language of such tools, which may need more or less start-up work depending on whether the tool already supports distributed systems or whether interaction models have to be encoded first. None of these tools, however, is currently able to deal with reduction proofs. Nobody even thought about this for a long time, because one felt that protocol proofs could be based on simpler, idealized abstractions from cryptographic primitives. Almost all these abstractions are variants of the Dolev-Yao model [33], which represents all cryptographic primitives as operators of a term algebra with cancellation rules. For instance, public-key encryption is represented by operators E for encryption and D for decryption with one cancellation rule, $\mathsf{D}(\mathsf{E}(m)) = m$ for all m. Encrypting a message m twice in this model does not yield another message from the basic message space but the term $\mathsf{E}(\mathsf{E}(m))$. Further, the model assumes that two terms whose equality cannot be derived with the cancellation rules are not equal, and every term that cannot be derived is completely secret. However, originally there was no foundation at all for such assumptions about real cryptographic primitives, and thus no guarantee that protocols proved with these tools were still secure when implemented with real cryptography. Although no previously proved protocol has been broken when implemented with standard provably secure cryptosystems, this was clearly an unsatisfactory situation, and artificial counterexamples can be constructed.

1.1 A Dolev-Yao Model That Is Cryptographically Sound Under Active Attacks

The conference version [12] underlying this paper is the first one that offers a provably secure variant of the Dolev-Yao model for proofs that people typically make with the Dolev-Yao model, because for the first time we cover both active attacks and nested cryptographic operations. While [12] addressed the soundness of asymmetric cryptographic primitives such as public-key encryption and digital signatures, subsequent papers extended the soundness result to symmetric authentication [13] and symmetric encryption [9]. Combining security against active attacks and nesting cryptographic operations arbitrarily is essential: First, most cryptographic protocols are broken by active attacks, e.g., man-in-the-middle attacks or attacks where an adversary reuses a message from one protocol step in a different protocol step where it suddenly gets a different semantics. Such attacks are not covered by [2,1]. Secondly, the main use of the Dolev-Yao model is to represent nested protocol messages like $\mathsf{E}_{pke_v}(\mathsf{sign}_{sks_u}(m, N_1), N_2)$, where m denotes an arbitrary message and N_1, N_2 two nonces. No previous idealization proved in the reactive cryptographic models contains abstractions from cryptographic primitives (here mainly encryption and signatures, but also the nonces and the list operation) that can be used in such nested terms. Existing abstractions are either too high-level, e.g., the secure channels in [65,5] combine encryption and signatures in a fixed way. Or they need immediate interaction with the adversary [23,22], i.e., the adversary learns the structure of every term any honest party ever builds, and even every signed message. This abstraction is not usable for a term as above because one may want to show that m is secret because of the outer encryption, but the abstraction gives m to the adversary. (A similar immediate application of the model of [65] to such primitives would avoid this problem, but instead keep all signatures and ciphertexts in the system, so that nesting is also not possible.) Finally, there exist some semi-abstractions which still depend on cryptographic details [49,65]. Thus they are not suitable for abstract protocol representations and proof tools, but we use such a semi-abstraction of public-key encryption as a submodule below.

The first decision in the design of an ideal library that supports both nesting and general active attacks was how we can represent an idealized cryptographic term and the corresponding real message in the *same* way to a higher protocol. This is necessary for using the reactive cryptographic models and their composition theorems. We do this by handles, i.e., local names. In the ideal system, these handles essentially point to Dolev-Yao-like terms, while in the real system they point to real cryptographic messages. Our model for storing the terms belonging to the handles is stateful and in the ideal system comprises the knowledge of who knows which terms. Thus our overall ideal cryptographic library corresponds more to "the CSP Dolev-Yao model" or "the Strand-space Dolev-Yao model" than the pure algebraic Dolev-Yao model. Once one has the idea of handles, one has to consider whether one can put the exact Dolev-Yao terms under them or how one has to or wants to deviate from them in order to allow a provably secure cryptographic realization, based on a more or less general class of underlying primitives. An overview of these deviations is given in Section 1.4, and Section 1.5 surveys how the cryptographic primitives are augmented to give a secure implementation of the ideal library.

The vast majority of the work was to make a credible proof that the real cryptographic library securely implements the ideal one. This is a hand-made proof based on cryptographic primitives and with many distributed-systems aspects, and thus with all the problems mentioned above for cryptographic proofs of large protocols. Indeed we needed a novel proof technique consisting of a probabilistic, imperfect bisimulation with an embedded static information-flow analysis, followed by cryptographic reductions proofs for so-called error sets of traces where the bisimulation did not work. As this proof needs to be made only once, and is intended to be the justification for later basing many protocol proofs on the ideal cryptographic library and proving them with higher assurance using automatic tools, we carefully worked out all the tedious details, and we encourage some readers to double-check the 68-page full version of this paper [14] and the extension to symmetric cryptographic operations [13,9]. Based on our experience with making this proof and the errors we found by making it, we strongly discourage the reader against accepting idealizations of cryptographic primitives where a similar security property, simulatability, is claimed but only the first step of the proof, the definition of a simulator, is made. In the following, we sketch the ideal cryptographic library in Section 3, the concrete cryptographic realization in Section 4, and the proof of soundness in Section 5. We restrict our attention to asymmetric cryptographic operations in this paper and refer the reader to [13,9] for the cases of symmetric encryption and message authentication.

1.2 An Illustrating Example – A Cryptographically Sound Proof of the Needham-Schroeder-Lowe Protocol

To illustrate the usefulness of the ideal cryptographic library, we investigate the well-known Needham-Schroeder public-key authentication protocol [59,50], which arguably constitutes the most prominent example demonstrating the usefulness of the formal-methods approach after Lowe used the FDR model checker to discover a man-in-the-middle attack against the protocol. Lowe later proposed a repaired version of the protocol [51] and used the model checker to prove that this modified protocol (henceforth known as the Needham-Schroeder-Lowe protocol) is secure in the Dolev-Yao model. The original and the repaired Needham-Schroeder public-key protocols are two of the most often investigated security protocols, e.g., [69,53,68,70]. Various new approaches and proof tools for the analysis of security protocols were validated by rediscovering the known flaw or proving the fixed protocol in the Dolev-Yao model.

It is well-known and easy to show that the security flaw of the original protocol in the Dolev-Yao model can be used to mount a successful attack against any cryptographic implementation of the protocol. However, all previous security proofs of the repaired protocol are in the Dolev-Yao model, and no theorem carried these results over to the cryptographic approach with its much more comprehensive adversary. We close this gap, i.e., we show that the Needham-Schroeder-Lowe protocol is secure in the cryptographic approach. More precisely, we show that it is secure against arbitrary active attacks, including arbitrary concurrent protocol runs and arbitrary manipulation of bit-strings within polynomial time. The underlying assumption is that the Dolev-Yao-style abstraction of public-key encryption is implemented using a chosen-ciphertext secure public-key encryption scheme with small additions like ciphertext tagging. Chosen-ciphertext security was introduced in [66] and formulated as IND-CCA2 in [17]. Efficient encryption systems secure in this sense exist under reasonable assumptions [28].

Our proof is built upon the ideal cryptographic library, and a composition theorem for the underlying security notion implies that protocol proofs can be made using the ideal library, and security then carries over automatically to the cryptographic realization. However, because of the extension to the Dolev-Yao model, no prior formal-methods proof carries over directly. Our paper therefore validates this approach by the first protocol proof over the new ideal cryptographic library, and cryptographic security follows as a corollary. Besides its value for the Needham-Schroeder-Lowe protocol, the proof shows that in spite of the extensions and differences in presentation with respect to prior Dolev-Yao models, a proof can be made over the new library that seems easily accessible to current automated proof tools. In particular, the proof contains neither probabilism nor computational restrictions. In the following, we express the Needham-Schroeder-Lowe protocol based on the ideal cryptographic library in Section 6 and 7. We formally capture the entity authentication requirement in Section 8, and we prove in Section 9 that entity authentication based on the ideal library implies (the cryptographic definition of) entity authentication based on the concrete realization of the library. Finally, we prove the entity authentication property based on the ideal library in Section 10.

1.3 Further Related Literature

Both the cryptographic and the idealizing approach at proving cryptographic systems started in the early 80s. Early examples of cryptographic definitions and reduction proofs are [39,40]. Applied to protocols, these techniques are at their best for relatively small protocols where there is still a certain interaction between cryptographic primitives, e.g., [18,67]. The early methods of automating proofs based on the Dolev-Yao model are summarized in [46]. More recently, such work concentrated on using existing general-purpose model checkers [51,58,30] and theorem provers [34,61], and on treating larger protocols, e.g., [16].

Work intended to bridge the gap between the cryptographic approach and the use of automated tools started independently with [62,64] and [2]. In [2], Dolev-Yao terms, i.e., with nested operations, are considered specifically for symmetric encryption. However, the adversary is restricted to passive eavesdropping. Consequently, it was not necessary to define a reactive model of a system, its honest users, and an adversary, and the security goals were all formulated as indistinguishability of terms. This was extended in [1] from terms to more general programs, but the restriction to passive adversaries remains, which is not realistic in most practical applications. Further, there are no theorems about composition or property preservation from the abstract to the real system. Several papers extended this work for specific models or specific properties. For instance, [41] specifically considers strand spaces and information-theoretically secure authentication only. In [47] a deduction system for information flow is based on the same operations as in [2], still under passive attacks only.

The approach in [62,64] was from the other end: It starts with a general reactive system model, a general definition of cryptographically secure implementation by simulatability, and a composition theorem for this notion of secure implementation. This work is based on definitions of secure *function* evaluation, i.e., the computation of one set of outputs from one set of inputs [38,54,15,21]; earlier extensions towards reactive systems were either without real abstraction [49] or for quite special cases [44]. The approach was extended from synchronous to asynchronous systems in [65,23]. All the reactive works come with more or less worked-out examples of abstractions of cryptographic systems, and first tool-supported proofs were made based on such an abstraction [5,4] using the theorem prover PVS [60]. However, even with a composition theorem this does not automatically give a cryptographic library in the Dolev-Yao sense, i.e., with the possibility to nest abstract operations, as explained above. Our cryptographic library overcomes these problems. It supports nested operations in the intuitive sense; operations that are performed locally are not visible to the adversary. It is secure against arbitrary active attacks, and works in the context of arbitrary surrounding interactive protocols. This holds independently of the goals that one wants to prove about the surrounding protocols; in particular, property preservation theorems for the simulatability definition we use have been proved for integrity, secrecy, liveness, and non-interference [4,10,11,6,8].

Concurrently to [12], an extension to asymmetric encryption, but still under passive attacks only, has been presented in [43]. The underlying masters thesis [42] considers asymmetric encryption under active attacks, but in the random oracle model, which is itself an idealization of cryptography and not justifiable [24]. Laud [48] has subsequently presented a cryptographic underpinning for a Dolev-Yao model of symmetric encryption under active attacks. His work enjoys a direct connection with a formal proof tool, but it is specific to certain confidentiality properties, restricts the surrounding protocols to straight-line programs in a specific language, and does not address a connection to the remaining primitives of the Dolev-Yao model. Herzog et al. [43] and Micciancio and Warinschi [55] have subsequently also given a cryptographic underpinning under active attacks. Their results are narrower than that in [12] since they are specific for public-key encryption, but consider slightly simpler real implementations; moreover, the former relies on a stronger assumption whereas the latter severely restricts the classes of protocols and protocol properties that can be analyzed using this primitive. Section 6 of [55] further points out several possible extensions of their work which all already exist in the earlier work of [12]. Recently, Canetti and Herzog [25] have linked ideal functionalities for mutual authentication and key exchange protocols to corresponding representations in a formal language. They apply their techniques to the Needham-Schroeder-Lowe protocol by considering the exchanged nonces as secret keys. Their work is restricted to the mentioned functionalities and in contrast to the universally composable library [12] hence does not address soundness of Dolev-Yao models in their usual generality. The considered language does not allow loops and offers public-key encryption as the only cryptographic operation. Moreover, their approach to define a mapping between ideal and real traces following the ideas of [55] only captures trace-based properties (i.e., integrity properties); reasoning about secrecy properties additionally requires ad-hoc and functionality-specific arguments.

Efforts are also under way to formulate syntactic calculi for dealing with probabilism and polynomial-time considerations, in particular [56,49,57,45] and, as a second step, to encode them into proof tools. This approach can not yet handle protocols with any degree of automation. It is complementary to the approach of proving simple deterministic abstractions of cryptography and working with those wherever cryptography is only used in a blackbox way.

The first cryptographically sound security proofs of the Needham-Schroeder-Lowe protocol have been presented concurrently and independently in [7] and [72]. While the first paper conducts the proof by means of the ideal cryptographic library and hence within a deterministic framework that is accessible for machine-assisted verification, the proof in the second paper is done from scratch in the cryptographic approach and is hence vulnerable to the aforementioned problems. On the other hand, the second paper proves stronger properties; we discuss this in Section 8. It further shows that chosen-plaintext-secure encryption is insufficient for the security of the protocol. While certainly no full Dolev-Yao model would be needed to model just the Needham-Schroeder-Lowe protocol, there was no prior attempt to prove this or a similar cryptographic protocol based on a sound abstraction from cryptography in a way accessible to automated proof tools.

1.4 Overview of the Ideal Cryptographic Library

The ideal cryptographic library offers its users abstract cryptographic operations, such as commands to encrypt or decrypt a message, to make or test a signature, and to generate a nonce. All these commands have a simple, deterministic semantics. In a reactive scenario, this semantics is based on state, e.g., of who already knows which terms. We store state in a "database". Each entry has a type, e.g., "signature", and pointers to its arguments, e.g., a key and a message. This corresponds to the top level of a Dolev-Yao term; an entire term can be found by following the pointers. Further, each entry contains handles for those participants who already know it. Thus the database index and these handles serve as an infinite, but efficiently constructible supply of global and local names for cryptographic objects. However, most libraries have export operations and leave message transport to their users ("token-based"). An actual implementation of the simulatable library might internally also be structured like this, but higher protocols are only automatically secure if they do not use this export function except via the special send operations.

The ideal cryptographic library does not allow cheating. For instance, if it receives a command to encrypt a message m with a certain key, it simply makes an abstract entry in a database for the ciphertext. Each entry further contains handles for those participants who already know it. Another user can only ask for decryption of this ciphertext if he has handles to both the ciphertext and the secret key. Similarly, if a user issues a command to sign a message, the ideal system looks up whether this user should have the secret key. If yes, it stores that this message has been signed with this key. Later tests are simply look-ups in this database. A send operation makes an entry known to other participants, i.e., it adds handles to the entry. Recall that our ideal library is an entire reactive system and therefore contains an abstract network model. We offer three types of send commands, corresponding to three channel types $\{\mathsf{s}, \mathsf{r}, \mathsf{i}\}$, meaning secure, authentic (but not private), and insecure. The types could be extended. Currently, our library contains public-key encryption and signatures, nonces, lists, and application data. We have subsequently added symmetric authentication [13] and symmetric encryption [9]).

The main differences between our ideal cryptographic library and the standard Dolev-Yao model are the following. Some of them already exist in prior extensions of the Dolev-Yao model.

- Signature schemes are not "inverses" of encryption schemes.
- Secure encryption schemes are necessarily probabilistic, and so are most secure signature schemes. Hence if the same message is signed or encrypted several times, we distinguish the versions by making different database entries.
- Secure signature schemes often have memory. The standard definition [40] does not even exclude that one signature divulges the entire history of messages signed before. We have to restrict this definition, but we allow a signature to divulge

the number of previously signed messages, so that we include the most efficient provably secure schemes under classical assumptions like the hardness of factoring [40,26,27].[1]

- We cannot (easily) allow participants to send secret keys over the network because then the simulation is not always possible.[2] Fortunately, for public-key cryptosystems this is not needed in typical protocols.
- Encryption schemes cannot keep the length of arbitrary cleartexts entirely secret. Typically one can even see the length quite precisely because message expansion is minimized. Hence we also allow this in the ideal system. A fixed-length version would be an easy addition to the library, or can be implemented on top of the library by padding to a fixed length.
- Adversaries may include incorrect messages in encrypted parts of a message which the current recipient cannot decrypt, but may possibly forward to another recipient who can, and will thus notice the incorrect format. Hence we also allow certain "garbage" terms in the ideal system.

1.5 Overview of the Real Cryptographic Library

The real cryptographic library offers its users the same commands as the ideal one, i.e., honest users operate on cryptographic objects via handles. This is quite close to standard APIs for existing implementations of cryptographic libraries that include key storage. The database of the real system contains real cryptographic keys, ciphertexts, etc., and the commands are implemented by real cryptographic algorithms. Sending a term on an insecure channel releases the actual bitstring to the adversary, who can do with it what he likes. The adversary can also insert arbitrary bitstrings on non-authentic channels. The simulatability proof will show that nevertheless, everything a real adversary can achieve can also be achieved by an adversary in the ideal system, or otherwise the underlying cryptography can be broken.

We base the implementation of the commands on arbitrary secure encryption and signature systems according to standard cryptographic definitions. However, we "idealize" the cryptographic objects and operations by measures similar to robust protocol design [3].

- All objects are tagged with a type field so that, e.g., signatures cannot also be acceptable ciphertexts or keys.
- Several objects are also tagged with their parameters, e.g., signatures with the public key used.
- Randomized operations are randomized completely. For instance, as the ideal system represents several signatures under the same message with the same key as different, the real system has to guarantee that they *will* be different, except for small error probabilities. Even probabilistic encryptions are randomized additionally because they are not always sufficiently random for keys chosen by the adversary.

[1] Memory-less schemes exist with either lower efficiency or based on stronger assumptions (e.g., [37,29,36]). We could add them to the library as an additional primitive.

[2] The primitives become "committing". This is well-known from individual simulation proofs. It also explains why [2] is restricted to passive attacks.

The reason to tag signatures with the public key needed to verify them is that the usual definition of a secure signature scheme does not exclude "signature stealing:" Let (sks_h, pks_h) denote the key pair of a correct participant. With ordinary signatures an adversary might be able to compute a valid key pair (sks_a, pks_a) such that signatures that pass the test with pks_h also pass the test with pks_a. Thus, if a correct participant receives an encrypted signature on m, it might accept m as being signed by the adversary, although the adversary never saw m. It is easy to see that this would result in protocols that could not be simulated. Our modification prevents this anomaly.

For the additional randomization of signatures, we include a random string r in the message to be signed. Alternatively we could replace r by a counter, and if a signature scheme is strongly randomized already we could omit r. Ciphertexts are randomized by including the same random string r in the message to be encrypted and in the ciphertext. The outer r prevents collisions among ciphertexts from honest participants, the inner r ensures continued non-malleability.

2 Preliminary Definitions

We briefly sketch the definitions from [65]. A *system* consists of several possible *structures*. A structure consists of a set $\hat{M}$ of connected correct machines and a subset S of free ports, called *specified ports*. A machine is a probabilistic IO automaton (extended finite-state machine) in a slightly refined model to allow complexity considerations. For these machines Turing-machine realizations are defined, and the complexity of those is measured in terms of a common security parameter k, given as the initial work-tape content of every machine. Readers only interested in using the ideal cryptographic library in larger protocols only need normal, deterministic IO automata.

In a *standard real cryptographic system*, the structures are derived from one intended structure and a trust model consisting of an access structure $\mathcal{ACC}$ and a channel model χ. Here $\mathcal{ACC}$ contains the possible sets $\mathcal{H}$ of indices of uncorrupted machines among the intended ones, and χ designates whether each channel is secure, authentic (but not private) or insecure. In a typical ideal system, each structure contains only one machine TH called *trusted host*.

Each structure is complemented to a *configuration* by an arbitrary *user* machine H and *adversary* machine A. H connects only to ports in S and A to the rest, and they may interact. The set of configurations of a system Sys is called $\mathsf{Conf}(Sys)$. The general scheduling model in [65] gives each connection c (from an output port $\mathsf{c}!$ to an input port $\mathsf{c}?$) a buffer, and the machine with the corresponding clock port $\mathsf{c}^{\triangleleft}!$ can schedule a message there when it makes a transition. In real asynchronous cryptographic systems, network connections are typically scheduled by A. A configuration is a runnable system, i.e., for each k one gets a well-defined probability space of *runs*. The *view* of a machine in a run is the restriction to all in- and outputs this machine sees and its internal states. Formally, the view $view_{conf}(\mathsf{M})$ of a machine M in a configuration *conf* is a *family of random variables* with one element for each security parameter value k.

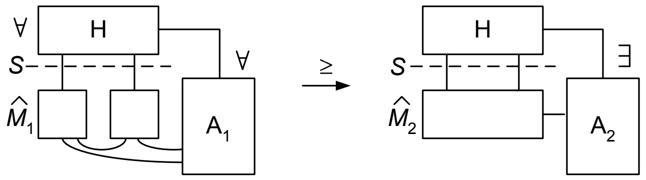


Fig. 1. Simulatability: The two views of H must be indistinguishable

2.1 Simulatability

Simulatability is the cryptographic notion of secure implementation. For reactive systems, it means that whatever might happen to an honest user in a real system $\mathit{Sys}_{\mathsf{real}}$ can also happen in the given ideal system $\mathit{Sys}_{\mathsf{id}}$: For every structure $(\hat{M}_1, S) \in \mathit{Sys}_{\mathsf{real}}$, every polynomial-time user H, and every polynomial-time adversary A_1, there exists a polynomial-time adversary A_2 on a corresponding ideal structure $(\hat{M}_2, S) \in \mathit{Sys}_{\mathsf{id}}$ such that the view of H is computationally indistinguishable in the two configurations. This is illustrated in Figure 1. Indistinguishability is a well-known cryptographic notion from [73].

Definition 1. *(Computational Indistinguishability) Two families* $(\mathsf{var}_k)_{k\in\mathbb{N}}$ *and* $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ *of random variables on common domains* D_k *are* computationally indistinguishable *("$\approx$") iff for every algorithm* Dis *(the distinguisher) that is probabilistic polynomial-time in its first input,*

$$|P(\mathsf{Dis}(1^k, \mathsf{var}_k) = 1) - P(\mathsf{Dis}(1^k, \mathsf{var}'_k) = 1)| \in \mathit{NEGL},$$

where NEGL denotes the set of all negligible functions*, i.e.,* $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \in \mathit{NEGL}$ *iff for all positive polynomials Q,* $\exists k_0 \forall k \geq k_0\colon\ g(k) \leq 1/Q(k)$. ◊

Intuitively, given the security parameter and an element chosen according to either var_k or var'_k, Dis tries to guess which distribution the element came from.

Definition 2. *(Simulatability) Let systems* $\mathit{Sys}_{\mathsf{real}}$ *and* $\mathit{Sys}_{\mathsf{id}}$ *be given. We say* $\mathit{Sys}_{\mathsf{real}} \geq \mathit{Sys}_{\mathsf{id}}$ *(*at least as secure as*) iff for every polynomial-time configuration* $\mathit{conf}_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(\mathit{Sys}_{\mathsf{real}})$*, there exists a polynomial-time configuration* $\mathit{conf}_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(\mathit{Sys}_{\mathsf{id}})$ *(with the same* H*) such that* $\mathit{view}_{\mathit{conf}_1}(\mathsf{H}) \approx \mathit{view}_{\mathit{conf}_2}(\mathsf{H})$. ◊

For the cryptographic library, we even show blackbox simulatability, i.e., A_2 consists of a simulator Sim that depends only on $(\hat{M}_1, S)$ and uses A_1 as a blackbox submachine. An essential feature of this definition of simulatability is a composition theorem [65], which essentially says that one can design and prove a larger system based on the ideal system $\mathit{Sys}_{\mathsf{id}}$, and then securely replace $\mathit{Sys}_{\mathsf{id}}$ by the real system $\mathit{Sys}_{\mathsf{real}}$.

2.2 Notation

We write ":=" for deterministic and "$\leftarrow$" for probabilistic assignment, and "$\xleftarrow{\mathcal{R}}$" for uniform random choice from a set. By $x := y$++ for integer variables x, y we mean

$y := y + 1; x := y$. The length of a message m is denoted as $\mathsf{len}(m)$, and $\downarrow$ is an error element available as an addition to the domains and ranges of all functions and algorithms. The list operation is denoted as $l := (x_1, \ldots, x_j)$, and the arguments are unambiguously retrievable as $l[i]$, with $l[i] = \downarrow$ if $i > j$. A database D is a set of functions, called entries, each over a finite domain called attributes. For an entry $x \in D$, the value at an attribute att is written $x.att$. For a predicate $pred$ involving attributes, $D[pred]$ means the subset of entries whose attributes fulfill $pred$. If $D[pred]$ contains only one element, we use the same notation for this element. Adding an entry x to D is abbreviated $D :\Leftarrow x$.

3 Ideal Cryptographic Library

The ideal cryptographic library consists of a trusted host $\mathsf{TH}_{\mathcal{H}}$ for every subset $\mathcal{H}$ of a set $\{1, \ldots, n\}$ of users. It has a port $\mathsf{in}_u?$ for inputs from and a port $\mathsf{out}_u!$ for outputs to each user $u \in \mathcal{H}$ and for $u = \mathsf{a}$, denoting the adversary.

As mentioned in Section 1.4, we do not assume encryption systems to hide the length of the message. Furthermore, higher protocols may need to know the length of certain terms even for honest participants. Thus the trusted host is parameterized with certain length functions denoting the length of a corresponding value in the real system. The tuple of these functions is contained in a system parameter L.

For simulatability by a polynomial-time real system, the ideal cryptographic library has to be polynomial-time. It therefore contains explicit bounds on the message lengths, the number of signatures per key, and the number of accepted inputs at each port. They are also contained in the system parameter L. The underlying IO automata model guarantees that a machine can enforce such bounds without additional Turing steps even if an adversary tries to send more data. For all details, we refer to [14].

3.1 States

The main data structure of $\mathsf{TH}_{\mathcal{H}}$ is a database D. The entries of D are abstract representations of the data produced during a system run, together with the information on who knows these data. Each entry $x \in D$ is of the form

$$(ind, type, arg, hnd_{u_1}, \ldots, hnd_{u_m}, hnd_{\mathsf{a}}, len)$$

where $\mathcal{H} = \{u_1, \ldots, u_m\}$ and:

- $ind \in \mathbb{N}_0$ is called the *index* of x. We write $D[i]$ instead of $D[ind = i]$.
- $type \in typeset := \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}, \mathsf{ske}, \mathsf{pke}, \mathsf{enc}, \mathsf{sks}, \mathsf{pks}, \mathsf{sig}, \mathsf{garbage}\}$ identifies the *type* of x. Future extensions of the library can extend this set.
- $arg = (a_1, a_2, \ldots, a_j)$ is a possibly empty *list of arguments*.
- $hnd_u \in \mathbb{N}_0 \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ identifies how u knows this entry. The value a represents the adversary, and $hnd_u = \downarrow$ indicates that u does not know this entry. A value $hnd_u \neq \downarrow$ is called the *handle* for u to entry x. We always use a superscript "hnd" for handles and usually denote a handle to an entry $D[i]$ by i^{hnd}.

– $len \in \mathbb{N}_0$ denotes the *length* of the abstract entry. It is computed by $\mathsf{TH}_\mathcal{H}$ using the given length functions from the system parameter L.

Initially, D is empty. $\mathsf{TH}_\mathcal{H}$ keeps a variable $size$ denoting the current number of elements in D. New entries x always receive the index $ind := size$++, and $x.ind$ is never changed. For each $u \in \mathcal{H} \cup \{\mathsf{a}\}$, $\mathsf{TH}_\mathcal{H}$ maintains a counter $curhnd_u$ (current handle) over $\mathbb{N}_0$ initialized with 0, and each new handle for u will be chosen as $i^{\mathsf{hnd}} := curhnd_u$++.

3.2 Inputs and Their Evaluation

Each input c at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$ should be a list $(cmd, x_1, \ldots, x_j)$. We usually write it $y \leftarrow cmd(x_1, \ldots, x_j)$ with a variable y designating the result that $\mathsf{TH}_\mathcal{H}$ returns at $\mathsf{out}_u!$. The value cmd should be a command string, contained in one of the following four *command sets*. Commands in the first two sets are available for both the user and the adversary, while the last two sets model special adversary capabilities and are only accepted for $u = \mathsf{a}$. The command sets can be enlarged by future extensions of the library.

Basic Commands. First, we have a set *basic_cmds* of *basic commands*. Each basic command represents one cryptographic operation; arbitrary terms similar to the Dolev-Yao model are built up or decomposed by a sequence of commands. For instance there is a command gen_nonce to create a nonce, encrypt to encrypt a message, and list to combine several messages into a list. Moreover, there are commands store and retrieve to store real-world messages (bitstrings) in the library and to retrieve them by a handle. Thus other commands can assume that everything is addressed by handles. We only allow lists to be signed and transferred, because the list-operation is a convenient place to concentrate all verifications that no secret items are put into messages. Altogether, we have

$$basic\_cmds := \{\mathsf{get\_type}, \mathsf{get\_len}, \mathsf{store}, \mathsf{retrieve}, \mathsf{list}, \mathsf{list\_proj}, \mathsf{gen\_nonce}, \mathsf{gen\_sig\_keypair}, \mathsf{sign}, \mathsf{verify}, \mathsf{pk\_of\_sig}, \mathsf{msg\_of\_sig}, \mathsf{gen\_enc\_keypair}, \mathsf{encrypt}, \mathsf{decrypt}, \mathsf{pk\_of\_enc}\}.$$

The commands not yet mentioned have the following meaning: get_type and get_len retrieve the type and abstract length of a message; list_proj retrieves a handle to the i-th element of a list; gen_sig_keypair and gen_enc_keypair generate key pairs for signatures and encryption, respectively, initially with handles for only the user u who input the command; sign, verify, and decrypt have the obvious purpose, and pk_of_sig, msg_of_sig; and pk_of_enc retrieve a public key or message, respectively, from a signature or ciphertext. (Retrieving public keys will be possible in the real cryptographic library because we tag signatures and ciphertexts with public keys as explained above.)

We only present the details of how $\mathsf{TH}_\mathcal{H}$ evaluates such basic commands based on its abstract state for two examples, nonce generation and encryption; see the full version [14] for the complete definition. We assume that the command is entered at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$. Basic commands are *local*, i.e., they produce a result at port $\mathsf{out}_u!$ and possibly update the database D, but do not produce outputs at other ports. They also do not touch handles for participants $v \neq u$. The functions $\mathsf{nonce\_len}^*$, $\mathsf{enc\_len}^*$, and $\mathsf{max\_len}$ are length functions and the message-length bound from the system parameter L.

For nonces, $\mathsf{TH}_\mathcal{H}$ just creates a new entry with type nonce, no arguments, a handle for user u, and the abstract nonce length. This models that in the real system nonces are randomly chosen bitstrings of a certain length, which should be all different and not guessable by anyone else than u initially. It outputs the handle to u.

– *Nonce Generation:* $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
  Set $n^{\mathsf{hnd}} := \mathit{curhnd}_u\text{++}$ and

$$D :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{nonce}, \mathit{arg} := (),\\ \mathit{hnd}_u := n^{\mathsf{hnd}}, \mathit{len} := \mathsf{nonce\_len}^*(k)).$$

The inputs for public-key encryption are handles to the public key and the plaintext list. $\mathsf{TH}_\mathcal{H}$ verifies the types (recall the notation $D[\mathit{pred}]$) and verifies that the ciphertext will not exceed the maximum length. If everything is ok, it makes a new entry of type enc, with the indices of the public key and the plaintext as arguments, a handle for user u, and the computed length. The fact that each such entry is new models probabilistic encryption, and the arguments model the highest layer of the corresponding Dolev-Yao term.

– *Public-Key Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.
  Let $pk := D[\mathit{hnd}_u = pk^{\mathsf{hnd}} \wedge \mathit{type} = \mathsf{pke}].\mathit{ind}$ and $l := D[\mathit{hnd}_u = l^{\mathsf{hnd}} \wedge \mathit{type} = \mathsf{list}].\mathit{ind}$ and $\mathit{length} := \mathsf{enc\_len}^*(k, D[l].\mathit{len})$. If $\mathit{length} > \mathsf{max\_len}(k)$ or $pk = \downarrow$ or $l = \downarrow$, then return $\downarrow$. Else set $c^{\mathsf{hnd}} := \mathit{curhnd}_u\text{++}$ and

$$D :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{enc}, \mathit{arg} := (pk, l),\\ \mathit{hnd}_u := c^{\mathsf{hnd}}, \mathit{len} := \mathit{length}).$$

Honest Send Commands. Secondly, we have a set $\mathit{send\_cmds} := \{\mathsf{send\_s}, \mathsf{send\_r}, \mathsf{send\_i}\}$ of *honest send commands* for sending messages on channels of different degrees of security. As an example we present the details of the most important case, insecure channels.

– $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$.
  Let $l^{\mathsf{ind}} := D[\mathit{hnd}_u = l^{\mathsf{hnd}} \wedge \mathit{type} = \mathsf{list}].\mathit{ind}$. If $l^{\mathsf{ind}} \neq \downarrow$, output $(u, v, \mathsf{i}, \mathsf{ind2hnd_a}(l^{\mathsf{ind}}))$ at $\mathsf{out_a}!$.

The used algorithm $\mathsf{ind2hnd}_u$ retrieves the handle for user u to the entry with the given index if there is one, otherwise it assigns a new such handle as $\mathit{curhnd}_u\text{++}$. Thus this command means that the database D now stores that this message is known to the adversary, and that the adversary learns by the output that user u wanted to send this message to user v.

Most protocols should only use the other two send commands, i.e., secret or authentic channels, for key distribution at the beginning. As the channel type is part of the send-command name, syntactic checks can ensure that a protocol designed with the ideal cryptographic library fulfills such requirements.

Local Adversary Commands. Thirdly, we have a set $adv\_local\_cmds := \{\mathsf{adv\_garbage}, \mathsf{adv\_invalid\_ciph}, \mathsf{adv\_transform\_sig}, \mathsf{adv\_parse}\}$ of *local adversary commands*. They model tolerable imperfections of the real system, i.e., actions that may be possible in real systems but that are not required. First, an adversary may create *invalid entries* of a certain length; they obtain the type garbage. Secondly, *invalid ciphertexts* are a special case because participants not knowing the secret key can reasonably ask for their type and query their public key, hence they cannot be of type garbage. Thirdly, the security definition of signature schemes does not exclude that the adversary *transforms signatures* by honest participants into other valid signatures on the same message with the same public key. Finally, we allow the adversary to retrieve all information that we do not explicitly require to be hidden, which is denoted by a command adv_parse. This command returns the type and usually all the abstract arguments of a value (with indices replaced by handles), e.g., parsing a signature yields the public key for testing this signature, the signed message, and the value of the signature counter used for this message. Only for ciphertexts where the adversary does not know the secret key, parsing only returns the length of the cleartext instead of the cleartext itself.

Adversary Send Commands. Fourthly, we have a set $adv\_send\_cmds := \{\mathsf{adv\_send\_s}, \mathsf{adv\_send\_r}, \mathsf{adv\_send\_i}\}$ of *adversary send commands*, again modeling different degrees of security of the channels used. In contrast to honest send commands, the sender of a message is an additional input parameter. Thus for insecure channels the adversary can pretend that a message is sent by an arbitrary honest user.

3.3 A Small Example

Assume that a cryptographic protocol has to perform the step

$$u \rightarrow v\colon \mathsf{enc}_{pke_v}(\mathsf{sign}_{sks_u}(m, N_1), N_2),$$

where m is an input message and N_1, N_2 are two fresh nonces. Given our library, this is represented by the following sequence of commands input at port $\mathsf{in}_u?$. We assume that u has already received a handle pke_v^{hnd} to the public encryption key of v, and created signature keys, which gave him a handle sks_u^{hnd}.

1. $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$.
2. $N_1^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
3. $l_1^{\mathsf{hnd}} \leftarrow \mathsf{list}(m^{\mathsf{hnd}}, N_1^{\mathsf{hnd}})$.
4. $sig^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sks_u^{\mathsf{hnd}}, l_1^{\mathsf{hnd}})$.
5. $N_2^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
6. $l_2^{\mathsf{hnd}} \leftarrow \mathsf{list}(sig^{\mathsf{hnd}}, N_2^{\mathsf{hnd}})$.
7. $enc^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_v^{\mathsf{hnd}}, l_2^{\mathsf{hnd}})$.
8. $m^{\mathsf{hnd}} \leftarrow \mathsf{list}(enc^{\mathsf{hnd}})$.
9. $\mathsf{send\_i}(v, m^{\mathsf{hnd}})$

Note that the entire term is constructed by a local interaction of user u and the ideal library, i.e., the adversary does not learn anything about this interaction until Step 8. In Step 9, the adversary gets an output $(u, v, \mathsf{i}, m_{\mathsf{a}}^{\mathsf{hnd}})$ with a handle $m_{\mathsf{a}}^{\mathsf{hnd}}$ for him to the resulting entry. In the real system described below, the sequence of inputs for constructing and sending this term is identical, but real cryptographic operations are used to build up a bitstring m until Step 8, and m is sent to v via a real insecure channel in Step 9.

4 Real Cryptographic Library

The real system is parameterized by a digital signature scheme $\mathcal{S}$ and a public-key encryption scheme $\mathcal{E}$. The ranges of all functions are $\{0,1\}^+ \cup \{\downarrow\}$. The signature scheme has to be secure against existential forgery under adaptive chosen-message attacks [40]. This is the accepted security definition for general-purpose signing. The encryption scheme has to fulfill that two equal-length messages are indistinguishable even in adaptive chosen-ciphertext attacks. Chosen-ciphertext security has been introduced in [66] and formalized as "IND-CCA2" in [17]. It is the accepted definition for general-purpose encryption. An efficient encryption system secure in this sense is [28]. Just like the ideal system, the real system is parameterized by a tuple L' of length functions and bounds.

4.1 Structures

The intended structure of the real cryptographic library consists of n machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$. Each M_u has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$, so that the same honest users can connect to the ideal and the real library. Each M_u has three connections $\mathsf{net}_{u,v,x}$ to each M_v for $x \in \{\mathsf{s}, \mathsf{r}, \mathsf{i}\}$. They are called network connections and the corresponding ports network ports. Network connections are scheduled by the adversary.

The actual system is a standard cryptographic system as defined in [65] and sketched in Section 2. Any subset of the machines may be corrupted, i.e., any set $\mathcal{H} \subseteq \{1, \ldots, n\}$ can denote the indices of correct machines. The channel model means that in an actual structure, an honest intended recipient gets all messages output at network ports of type s (secret) and a (authentic) and the adversary gets all messages output at ports of type a and i (insecure). Furthermore, the adversary makes all inputs to a network port of type i. This is shown in Figure 2.

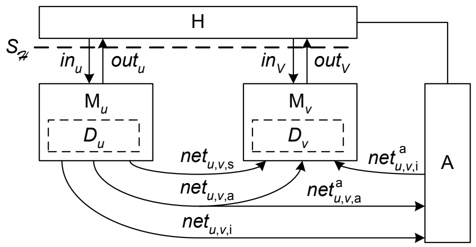


Fig. 2. Connections from a correct machine to another in the real system

4.2 States of a Machine

The main data structure of M_u is a database D_u that contains implementation-specific data such as ciphertexts and signatures produced during a system run, together with the handles for u and the type as in the ideal system, and possibly additional internal attributes. Thus each entry $x \in D_u$ is of the form

$$(hnd_u, word, type, add\_arg).$$

- $hnd_u \in \mathbb{N}_0$ is the *handle* of x and consecutively numbers all entries in D_u.
- $word \in \{0,1\}^+$, called *word*, is the real cryptographic representation of x.
- $type \in typeset \cup \{\mathsf{null}\}$ is the *type* of x, where null denotes that the entry has not yet been parsed.
- add_arg is a list of *additional arguments*. Typically it is $()$, only for signing keys it contains the signature counter.

Similar to the ideal system, M_u maintains a counter $curhnd_u$ over $\mathbb{N}_0$ denoting the current number of elements in D_u. New entries x always receive $hnd_u := curhnd_u$++, and $x.hnd_u$ is never changed.

4.3 Inputs and Their Evaluation

Now we describe how M_u evaluates individual inputs. Inputs at port $\mathsf{in}_u?$ should be basic commands and honest send commands as in the ideal system, while network inputs can be arbitrary bitstrings. Often a bitstrings has to be parsed. This is captured by a functional algorithm parse, which outputs a pair $(type, arg)$ of a type $\in typeset$ and a list of real arguments, i.e., of bitstrings. This corresponds to the top level of a term, similar to the abstract arguments in the ideal database D. By "parse m^{hnd}" we abbreviate that M_u calls $(type, arg) \leftarrow \mathsf{parse}(D_u[m^{\mathsf{hnd}}].word)$, assigns $D_u[m^{\mathsf{hnd}}].type := type$ if it was still null, and may then use arg.

Basic Commands. Basic commands are again *local*, i.e., they do not produce outputs at network ports. The basic commands are implemented by the underlying cryptographic operations with the modifications motivated in Section 1.5. For general unambiguousness, not only all cryptographic objects are tagged, but also data and lists. Similar to the ideal system, we only show two examples of the evaluation of basic commands, and additionally how ciphertexts are parsed. All other commands can be found in the full version [14].

In nonce generation, a real nonce n is generated by tagging a random bitstring n' of a given length with its type nonce. Further, a new handle for u is assigned and the handle, the word n, and the type are stored without additional arguments.

- *Nonce Generation:* $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
  Let $n' \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $n := (\mathsf{nonce}, n')$, $n^{\mathsf{hnd}} := curhnd_u$++ and $D_u :\Leftarrow (n^{\mathsf{hnd}}, n, \mathsf{nonce}, ())$.

For the encryption command, let $\mathsf{E}_{pk}(m)$ denote probabilistic encryption of a string m with the public key pk in the underlying encryption system $\mathcal{E}$. The parameters are first parsed in case they have been received over the network, and their types are verified. Then the second component of the (tagged) public-key word is the actual public key pk, while the message l is used as it is. Further, a fresh random value r is generated for additional randomization as explained in Section 1.5.

Recall that r has to be included both inside the encryption and in the final tagged ciphertext c^*.

– *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.
  Parse pk^{hnd} and l^{hnd}. If $D_u[pk^{\mathsf{hnd}}].type \neq \mathsf{pke}$ or $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, return $\downarrow$. Else set $pk := D_u[pk^{\mathsf{hnd}}].word[2]$, $l := D_u[l^{\mathsf{hnd}}].word$, $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, encrypt $c \leftarrow \mathsf{E}_{pk}((r, l))$, and set $c^* := (\mathsf{enc}, pk, c, r)$. If $c^* = \downarrow$ or $\mathsf{len}(c^*) > \mathsf{max\_len}(k)$ then return $\downarrow$, else set $c^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (c^{\mathsf{hnd}}, c^*, \mathsf{enc}, ())$.

Parsing a ciphertext verifies that the components and lengths are as in c^* above, and outputs the corresponding tagged public key, whereas the message is only retrieved by a decryption command.

Send Commands and Network Inputs. Send commands simply output real messages at the appropriate network ports. We show this for an insecure channel.

– $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$ for $v \in \{1, \ldots, n\}$.
  Parse l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$, output $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,\mathsf{i}}!$.

Upon receiving a bitstring l at a network port $\mathsf{net}_{w,u,x}?$, machine M_u parses it and verifies that it is a list. If yes, and if l is new, M_u stores it in D_u using a new handle l^{hnd}, else it retrieves the existing handle l^{hnd}. Then it outputs (w, x, l^{hnd}) at port $\mathsf{out}_u!$.

5 Security Proof

The security claim is that the real cryptographic library is as secure as the ideal cryptographic library, so that protocols proved on the basis of the deterministic, Dolev-Yao-like ideal library can be safely implemented with the real cryptographic library. To formulate the theorem, we need additional notation: Let $Sys^{\mathsf{cry,id}}_{n,L}$ denote the ideal cryptographic library for n participants and with length functions and bounds L, and $Sys^{\mathsf{cry,real}}_{n,\mathcal{S},\mathcal{E},L}$ the real cryptographic library for n participants, based on a secure signature scheme $\mathcal{S}$ and a secure encryption scheme $\mathcal{E}$, and with length functions and bounds L'. Let $RPar$ be the set of valid parameter tuples for the real system, consisting of the number $n \in \mathbb{N}$ of participants, secure signature and encryption schemes $\mathcal{S}$ and $\mathcal{E}$, and length functions and bounds L'. For $(n, \mathcal{S}, \mathcal{E}, L') \in RPar$, let $Sys^{\mathsf{cry,real}}_{n,\mathcal{S},\mathcal{E},L}$ be the resulting real cryptographic library. Further, let the corresponding length functions and bounds of the ideal system be formalized by a function $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$, and let $Sys^{\mathsf{cry,id}}_{n,L}$ be the ideal cryptographic library with parameters n and L. Using the notation of Definition 2, we then have

Theorem 1. *(Security of Cryptographic Library) For all parameters $(n, \mathcal{S}, \mathcal{E}, L') \in RPar$, we have*

$$Sys^{\mathsf{cry,real}}_{n,\mathcal{S},\mathcal{E},L} \geq Sys^{\mathsf{cry,id}}_{n,L},$$

where $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$. □

For proving this theorem, we define a simulator $\mathsf{Sim}_{\mathcal{H}}$ such that even the combination of arbitrary polynomial-time users H and an arbitrary polynomial-time adversary A cannot distinguish the combination of the real machines M_u from the combination $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ (for all sets $\mathcal{H}$ indicating the correct machines). We first sketch the simulator and then the proof of correct simulation.

5.1 Simulator

Basically $\mathsf{Sim}_{\mathcal{H}}$ has to translate real messages from the real adversary A into handles as $\mathsf{TH}_{\mathcal{H}}$ expects them at its adversary input port $\mathsf{in_a}?$ and vice versa; see Figure 3. In both directions, $\mathsf{Sim}_{\mathcal{H}}$ has to parse an incoming messages completely because it can only construct the other version (abstract or real) bottom-up. This is done by recursive algorithms. In some cases, the simulator cannot produce any corresponding message. We collect these cases in so-called *error sets* and show later that they cannot occur at all or only with negligible probability.

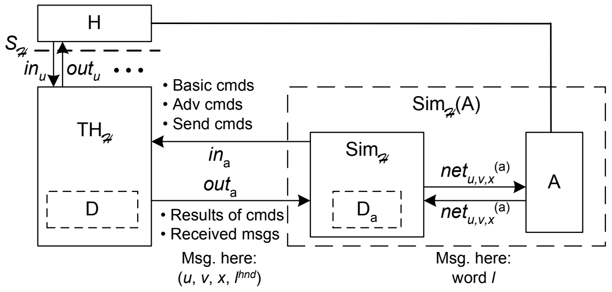


Fig. 3. Ports and in- and output types of the simulator

The state of $\mathsf{Sim}_{\mathcal{H}}$ mainly consists of a database D_{a}, similar to the databases D_u, but storing the knowledge of the adversary. The behavior of $\mathsf{Sim}_{\mathcal{H}}$ is sketched as follows.

– *Inputs from* $\mathsf{TH}_{\mathcal{H}}$. Assume that $\mathsf{Sim}_{\mathcal{H}}$ receives an input $(u, v, x, l^{\mathsf{hnd}})$ from $\mathsf{TH}_{\mathcal{H}}$. If a bitstring l for l^{hnd} already exists in D_{a}, i.e., this message is already known to the adversary, the simulator immediately outputs l at port $\mathsf{net}_{u,v,x}!$. Otherwise, it first constructs such a bitstring l with a recursive algorithm id2real. This algorithm decomposes the abstract term using basic commands and the adversary command adv_parse. At the same time, id2real builds up a corresponding real bitstring using real cryptographic operations and enters all new message parts into D_{a} to recognize them when they are reused, both by $\mathsf{TH}_{\mathcal{H}}$ and by A.
Mostly, the simulator can construct subterms exactly like the correct machines would do in the real system. Only for encryptions with a public key of a correct machine, adv_parse does not yield the plaintext; thus there the simulator encrypts a fixed message of equal length. This simulation presupposes that all new message parts are of the standard formats, not those resulting from local adversary commands; this is proven correct in the bisimulation.
– *Inputs from* A. Now assume that $\mathsf{Sim}_{\mathcal{H}}$ receives a bitstring l from A at a port $\mathsf{net}_{u,v,x}?$. If l is not a valid list, $\mathsf{Sim}_{\mathcal{H}}$ aborts the transition. Otherwise it translates l into a corresponding handle l^{hnd} by an algorithm real2id, and outputs the abstract sending command $\mathsf{adv\_send\_}x(w, u, l^{\mathsf{hnd}})$ at port $\mathsf{in_a}!$.
If a handle l^{hnd} for l already exists in D_{a}, then real2id reuses that. Otherwise it recursively parses a real bitstring using the functional parsing algorithm. At the

same time, it builds up a corresponding abstract term in the database of $\mathsf{TH}_{\mathcal{H}}$. This finally yields the handle l^{hnd}. Furthermore, real2id enters all new subterms into D_{a}. For building up the abstract term, real2id makes extensive use of the special capabilities of the adversary modeled in $\mathsf{TH}_{\mathcal{H}}$. In the real system, the bitstring may, e.g., contain a transformed signature, i.e., a new signature for a message for which the correct user has already created another signature. Such a transformation of a signature is not excluded by the definition of secure signature schemes, hence it might occur in the real system. Therefore the simulator also has to be able to insert such a transformed signature into the database of $\mathsf{TH}_{\mathcal{H}}$, which explains the need for the command adv_transform_signature. Similarly, the adversary might send invalid ciphertexts or simply bitstrings that do not yield a valid type when being parsed. All these cases can be covered by using the special capabilities.
The only case for which no command exists is a forged signature under a new message. This leads the simulator to abort. (Such runs fall into an error set which is later shown to be negligible.)

As all the commands used by id2real and real2id are local, these algorithms give uninterrupted dialogues between $\mathsf{Sim}_{\mathcal{H}}$ and $\mathsf{TH}_{\mathcal{H}}$, which do not show up in the views of A and H.

Two important properties have to be shown about the simulator before the bisimulation. First, the simulator has to be polynomial-time. Otherwise, the joint machine $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ of $\mathsf{Sim}_{\mathcal{H}}$ and A might not be a valid polynomial-time adversary on the ideal system. Secondly, it has to be shown that the interaction between $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ in the recursive algorithms cannot fail because one of the machines reaches its runtime bound. The proof of both properties is quite involved, using an analysis of possible recursion depths depending on the number of existing handles (see [14]).

5.2 Proof of Correct Simulation

Given the simulator, we show that arbitrary polynomial-time users H and an arbitrary polynomial-time adversary A cannot distinguish the combination of the real machine M_u from the combination of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$. The standard technique in non-cryptographic distributed systems for rigorously proving that two systems have identical visible behaviors is a bisimulation, i.e., one defines a mapping between the respective states and shows that identical inputs in mapped states retain the mapping and produce identical outputs. We need a probabilistic bisimulation because the real system and the simulator are probabilistic, i.e., identical inputs should yield mapped states with the correct probabilities and identically distributed outputs. (For the former, we indeed use mappings, not arbitrary relations for the bisimulation.) In the presence of cryptography and active attacks however, a normal probabilistic bisimulation is still insufficient for three crucial reasons. First, the adversary might succeed in attacking the real system with a very small probability, while this is impossible in the ideal system. This means that we have to cope with *error probabilities*. Secondly, encryption only gives computational indistinguishability, which cannot be captured by a bisimulation, because the actual values in the two systems may be quite different. Thirdly, the adversary might guess a random value, e.g., a nonce that has already been created by some machine but

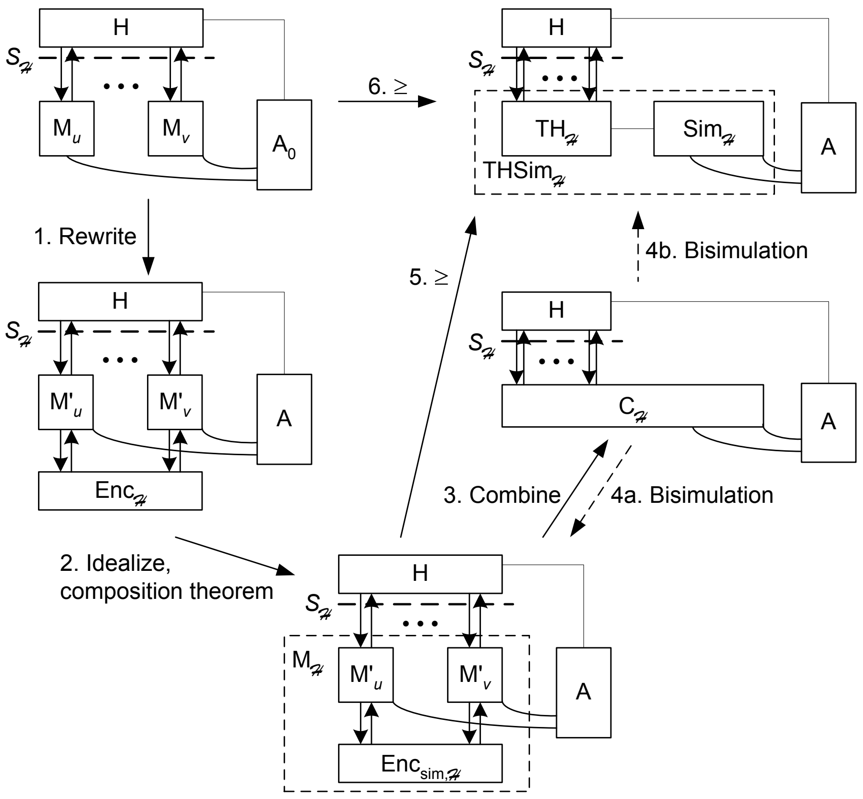


Fig. 4. Overview of the proof of correct simulation

that the adversary has ideally not yet seen. (Formally, "ideally not yet seen" just means that the bisimulation fails if the adversary sends a certain value which already exists in the databases but for which there is no command to give the adversary a handle.) In order to perform a rigorous reduction proof in this case, we have to show that no *partial information* about this value has already leaked to the adversary because the value was contained in a nested term, or because certain operations would leak partial information. For instance, here the proof would fail if we allowed arbitrary signatures according to the definition of [40], which might divulge previously signed messages, or if we did not additionally randomize probabilistic ciphertexts made with keys of the adversary.

We meet these challenges by first factoring out the computational aspects by a special treatment of ciphertexts. Then we use a new bisimulation technique that includes a static information-flow analysis, and is followed by the remaining cryptographic reductions. The rigorous proof takes 30 pages [14]; hence we can only give a very brief overview here, see also Figure 4.

– *Introducing encryption machines.* We use the two encryption machines $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ from [65] to handle the encryption and decryption needs of the system. Roughly, the first machine calculates the correct encryption of every message m, whereas the second one always encrypts the fixed message $m_{sim}\mathsf{len}(m)$ and answers decryption requests for the resulting ciphertexts by table look-up. By [65], $\mathsf{Enc}_{\mathcal{H}}$ is at least as secure as $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$. We rewrite the machines M_u such that they use $\mathsf{Enc}_{\mathcal{H}}$ (Step 1 in Figure 4); this yields modified machines M'_u. We then replace $\mathsf{Enc}_{\mathcal{H}}$ by its idealized counterpart $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ (Step 2 in Figure 4) and use the composition theorem to show that the original system is at least as secure as the resulting system.

- *Combined system.* We now want to compare the combination $\mathsf{M}_{\mathcal{H}}$ of the machines M'_u and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ with the combination $\mathsf{THSim}_{\mathcal{H}}$ of the machines $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$. However, there is no direct invariant mapping between the states of these two joint machines. Hence we defining an intermediate system $\hat{C}_{\mathcal{H}}$ with a state space combined from both these systems (Step 3 in Figure 4).
- *Bisimulations with error sets and information-flow analysis.* We show that the joint view of H and A is equal in interaction with the combined machine $\hat{C}_{\mathcal{H}}$ and the two machines $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$, except for certain runs, which we collect in *error sets*. We show this by performing two bisimulations simultaneously (Step 4 in Figure 4). Transitivity and symmetry of indistinguishability then yield the desired result for $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$. Besides several normal state invariants of $\hat{C}_{\mathcal{H}}$, we also define and prove an information-flow invariant on the variables of $\hat{C}_{\mathcal{H}}$.
- *Reduction proofs.* We show that the aggregated probability of the runs in error sets is negligible, as we could otherwise break the underlying cryptography. I.e., we perform reduction proofs against the security definitions of the primitives. For signature forgeries and collisions of nonces or ciphertexts, these are relatively straightforward proofs. For the fact that the adversary cannot guess "official" nonces as well as additional randomizers in signatures and ciphertext, we use the information-flow invariant on the variables of $\hat{C}_{\mathcal{H}}$ to show that the adversary has no partial information about such values in situations where correct guessing would put the run in an error set. This proves that $\mathsf{M}_{\mathcal{H}}$ is computationally at least as secure as the ideal system (Step 5 in Figure 4).

Finally, simulatability is transitive [65]. Hence the original real system is also as secure as the ideal system (Step 6 in Figure 4).

6 The Needham-Schroeder-Lowe Protocol

The original Needham-Schroeder public-key protocol and Lowe's variant consist of seven steps. Four steps deal with key generation and public-key distribution. They are usually omitted in a security analysis, and it is simply assumed that keys have already been generated and distributed. We do this as well to keep the proof short. However, the underlying cryptographic library offers commands for modeling these steps as well. The main part of the Needham-Schroeder-Lowe public-key protocol consists of the following three steps, expressed in the typical protocol notation, as in, e.g., [50].

$$\begin{aligned}
&1. \quad u \rightarrow v : E_{pk_v}(N_u, u)\\
&2. \quad v \rightarrow u : E_{pk_u}(N_u, N_v, v)\\
&3. \quad u \rightarrow v : E_{pk_v}(N_v).
\end{aligned}$$

Here, user u seeks to establish a session with user v. He generates a nonce N_u and sends it to v together with his identity, encrypted with v's public key (first message). Upon receiving this message, v decrypts it to obtain the nonce N_u. Then v generates a new nonce N_v and sends both nonces and her identity back to u, encrypted with u's public key (second message). Upon receiving this message, u decrypts it and tests whether the contained identity v equals the sender of the message and whether u earlier sent the first contained nonce to user v. If yes, u sends the second nonce back to v, encrypted with v's public key (third message). Finally, v decrypts this message; and if v had earlier sent the contained nonce to u, then v believes to speak with u.

7 The Needham-Schroeder-Lowe Protocol Using the Dolev-Yao-Style Cryptographic Library

Almost all formal proof techniques for protocols such as Needham-Schroeder-Lowe first need a reformulation of the protocol into a more detailed version than the three steps above. These details include necessary tests on received messages, the types and generation rules for values like u and N_u, and a surrounding framework specifying the number of participants, the possibilities of multiple protocol runs, and the adversary capabilities. The same is true when using the Dolev-Yao-style cryptographic library from [12], i.e., it plays a similar role in our proof as "the CSP Dolev-Yao model" or "the inductive-approach Dolev-Yao model" in other proofs. Our protocol formulation in this framework is given in Algorithms 1 and 2.[3] We first explain this formulation, and then consider general aspects of the surrounding framework as far as needed in our proofs.

7.1 Detailed Protocol Descriptions

Recall that the underlying framework is automata-based, i.e., protocols are executed by interacting machines, and event-based, i.e., machines react on received inputs. By $\mathsf{M}_i^{\mathsf{NS}}$ we denote the Needham-Schroeder machine for a participant i; it can act in the roles of both u and v above.

The first type of input that $\mathsf{M}_i^{\mathsf{NS}}$ can receive is a start message $(\mathsf{new\_prot}, v)$ from its user denoting that it should start a protocol run with user v. The number of users is called n. User inputs are distinguished from network inputs by arriving at a port $\mathsf{EA\_in}_u?$. The "EA" stands for entity authentication because the user interface is the same for all entity authentication protocols. The reaction on this input, i.e., the sending of the first message, is described in Algorithm 1.

The command gen_nonce generates the nonce. $\mathsf{M}_u^{\mathsf{NS}}$ adds the result n_u^{hnd} to a set $Nonce_{u,v}$ for future comparison. The command store inputs arbitrary application data into the cryptographic library, here the user identity u. The command list forms a list and encrypt is encryption. The final command send_i means that $\mathsf{M}_u^{\mathsf{NS}}$ attempts to send the resulting term to v over an insecure channel. The list operation directly before sending

[3] For some frameworks there are compilers to generate these detailed protocol descriptions, e.g., [52]. This should be possible for this framework in a similar way.

Algorithm 1 Evaluation of User Inputs in $\mathsf{M}_u^{\mathsf{NS}}$

```
Input: $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$ with $v \in \{1, \ldots, n\} \setminus \{u\}$.
1: $n_u^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
2: $Nonce_{u,v} := Nonce_{u,v} \cup \{n_u^{\mathsf{hnd}}\}$.
3: $u^{\mathsf{hnd}} \leftarrow \mathsf{store}(u)$.
4: $l_1^{\mathsf{hnd}} \leftarrow \mathsf{list}(n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$.
5: $c_1^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_1^{\mathsf{hnd}})$.
6: $m_1^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_1^{\mathsf{hnd}})$.
7: $\mathsf{send\_i}(v, m_1^{\mathsf{hnd}})$.
```

is a technicality: recall that only lists are allowed to be sent in this library because the list operation concentrates verifications that no secret items are put into messages.

The behavior of the Needham-Schroeder machine of participant u upon receiving a network input is defined similarly in Algorithm 2. The input arrives at port $\mathsf{out}_u?$ and is of the form $(v, u, \mathsf{i}, m^{\mathsf{hnd}})$ where v is the supposed sender, i denotes that the channel is insecure, and m^{hnd} is a handle to a list. The port $\mathsf{out}_u?$ is connected to the cryptographic library, whose two implementations represent the obtained Dolev-Yao-style term or real bitstring, respectively, to the protocol in a unified way by a handle.

In this algorithm, the protocol machine first decrypts the list content using its secret key; this yields a handle l^{hnd} to an inner list. This list is parsed into at most three components using the command $\mathsf{list\_proj}$. If the list has two elements, i.e., it could correspond to the first message of the protocol, and if it contains the correct identity, the machine generates a new nonce and stores its handle in the set $Nonce_{u,v}$. Then it builds up a new list according to the protocol description, encrypts it and sends it to user v. If the list has three elements, i.e., it could correspond to the second message of the protocol, the machine tests whether the third list element equals v and the first list element is contained in the set $Nonce_{u,v}$. If one of these tests does not succeed, $\mathsf{M}_u^{\mathsf{NS}}$ aborts. Otherwise, it again builds up a term according to the protocol description and sends it to user v. Finally, if the list has only one element, i.e., it could correspond to the third message of the protocol, the machine tests if the handle of this element is contained in $Nonce_{u,v}$. If so, $\mathsf{M}_u^{\mathsf{NS}}$ outputs (ok, v) at $\mathsf{EA\_out}_u!$. This signals to user u that the protocol with user v has terminated successfully, i.e., u believes to speak with v.

Both algorithms should immediately abort the handling of the current input if a cryptographic command does not yield the desired result, e.g., if a decryption fails. For readability we omitted this in the algorithm descriptions; instead we impose the following convention on both algorithms.

Convention 1. *If* $\mathsf{M}_u^{\mathsf{NS}}$ *receives* $\downarrow$ *as the answer of the cryptographic library to a command, then* $\mathsf{M}_u^{\mathsf{NS}}$ *aborts the execution of the current algorithm, except for the command types* $\mathsf{list\_proj}$ *or* $\mathsf{send\_i}$.

We refer to Step i of Algorithm j as Step $j.i$.

Algorithm 2 Evaluation of Network Inputs in $\mathsf{M}_u^{\mathsf{NS}}$

```
Input: $(v, u, \mathsf{i}, m^{\mathsf{hnd}})$ at $\mathsf{out}_u?$ with $v \in \{1, \ldots, n\} \setminus \{u\}$.
 1: $c^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(m^{\mathsf{hnd}}, 1)$
 2: $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(ske_u^{\mathsf{hnd}}, c^{\mathsf{hnd}})$
 3: $x_i^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$ for $i = 1, 2, 3$.
 4: if $x_1^{\mathsf{hnd}} \neq \downarrow \wedge x_2^{\mathsf{hnd}} \neq \downarrow \wedge x_3^{\mathsf{hnd}} = \downarrow$ then {First Message is input}
 5:    $x_2 \leftarrow \mathsf{retrieve}(x_2^{\mathsf{hnd}})$.
 6:    if $x_2 \neq v$ then
 7:       Abort
 8:    end if
 9:    $n_u^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
10:    $Nonce_{u,v} := Nonce_{u,v} \cup \{n_u^{\mathsf{hnd}}\}$.
11:    $u^{\mathsf{hnd}} \leftarrow \mathsf{store}(u)$.
12:    $l_2^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$.
13:    $c_2^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_2^{\mathsf{hnd}})$.
14:    $m_2^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_2^{\mathsf{hnd}})$.
15:    $\mathsf{send\_i}(v, m_2^{\mathsf{hnd}})$.
16: else if $x_1^{\mathsf{hnd}} \neq \downarrow \wedge x_2^{\mathsf{hnd}} \neq \downarrow \wedge x_3^{\mathsf{hnd}} \neq \downarrow$ then {Second Message is input}
17:    $x_3 \leftarrow \mathsf{retrieve}(x_3^{\mathsf{hnd}})$.
18:    if $x_3 \neq v \vee x_1^{\mathsf{hnd}} \not\in Nonce_{u,v}$ then
19:       Abort
20:    end if
21:    $l_3^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_2^{\mathsf{hnd}})$.
22:    $c_3^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_3^{\mathsf{hnd}})$.
23:    $m_3^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_3^{\mathsf{hnd}})$.
24:    $\mathsf{send\_i}(v, m_3^{\mathsf{hnd}})$.
25: else if $x_1^{\mathsf{hnd}} \in Nonce_{u,v} \wedge x_2^{\mathsf{hnd}} = x_3^{\mathsf{hnd}} = \downarrow$ then {Third Message is input}
26:    Output $(\mathsf{ok}, v)$ at $\mathsf{EA\_out}_u!$.
27: end if
```

7.2 Overall Framework and Adversary Model

When protocol machines such as $\mathsf{M}_u^{\mathsf{NS}}$ for certain users $u \in \{1, \ldots, n\}$ are defined, there is no guarantee that all these machines are correct. A trust model determines for what subsets $\mathcal{H}$ of $\{1, \ldots, n\}$ we want to guarantee anything; in our case this is essentially for all subsets: We aim at entity authentication between u and v whenever $u, v \in \mathcal{H}$ and thus whenever $\mathsf{M}_u^{\mathsf{NS}}$ and $\mathsf{M}_v^{\mathsf{NS}}$ are correct. Incorrect machines disappear and are replaced by the adversary. Each set of potential correct machines together with its user interface constitute a structure, and the set of these structures is called the system, cf. Section 2.2. Recall further that when considering the security of a structure, an arbitrary probabilistic machine H is connected to the user interface to represent all users, and an arbitrary machine A is connected to the remaining free ports (typically the network) and to H to represent the adversary, see Fig. 5. In polynomial-time security proofs, H and A are polynomial-time.

This setting implies that any number of concurrent protocol runs with both honest participants and the adversary are considered because H and A can arbitrarily interleave protocol start inputs $(\mathsf{new\_prot}, v)$ with the delivery of network messages.

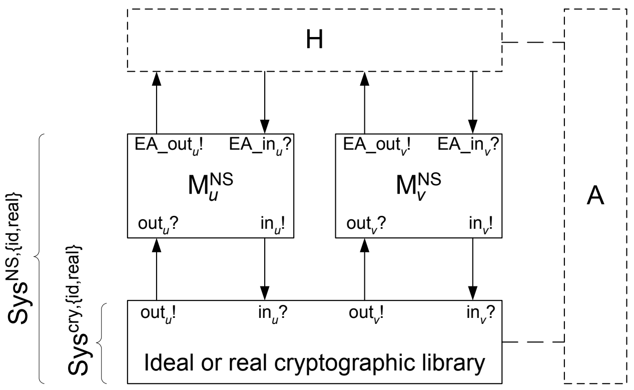


Fig. 5. Overview of the Needham-Schroeder-Lowe Ideal System

For a set $\mathcal{H}$ of honest participants, the user interface of the ideal and real cryptographic library is the port set $S_{\mathcal{H}}^{\mathsf{cry}} := \{\mathsf{in}_u?, \mathsf{out}_u! \mid u \in \mathcal{H}\}$. This is where the Needham-Schroeder machines input their cryptographic commands and obtain results and received messages. In the ideal case this interface is served by just one machine $\mathsf{TH}_{\mathcal{H}}$ called trusted host which essentially administrates Dolev-Yao-style terms under the handles. In the real case, the same interface is served by a set $\hat{M}_{\mathcal{H}}^{\mathsf{cry}} := \{\mathsf{M}_{u,\mathcal{H}}^{\mathsf{cry}} \mid u \in \mathcal{H}\}$ of real cryptographic machines. The corresponding systems are called $Sys^{\mathsf{cry,id}} := \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}^{\mathsf{cry}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$ and $Sys^{\mathsf{cry,real}} := \{(\hat{M}_{\mathcal{H}}^{\mathsf{cry}}, S_{\mathcal{H}}^{\mathsf{cry}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$.

The user interface of the Needham-Schroeder machines or any other entity authentication protocol is $S_{\mathcal{H}}^{\mathsf{EA}} := \{\mathsf{EA\_in}_u?, \mathsf{EA\_out}_u! \mid u \in \mathcal{H}\}$. The ideal and real Needham-Schroeder-Lowe systems serving this interface differ only in the cryptographic library. With $\hat{M}_{\mathcal{H}}^{\mathsf{NS}} := \{\mathsf{M}_u^{\mathsf{NS}} \mid u \in \mathcal{H}\}$, they are $Sys^{\mathsf{NS,id}} := \{(\hat{M}_{\mathcal{H}}^{\mathsf{NS}} \cup \{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}^{\mathsf{EA}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$ and $Sys^{\mathsf{NS,real}} := \{(\hat{M}_{\mathcal{H}}^{\mathsf{NS}} \cup \hat{M}_{\mathcal{H}}^{\mathsf{cry}}, S_{\mathcal{H}}^{\mathsf{EA}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$.

7.3 Initial State

We have assumed in the algorithms that each Needham-Schroeder machine $\mathsf{M}_u^{\mathsf{NS}}$ already has a handle ske_u^{hnd} to its own secret encryption key and handles $pke_{v,u}^{\mathsf{hnd}}$ to the corresponding public keys of every participant v. The cryptographic library can also represent key generation and distribution by normal commands. Formally, this assumption means that for each participant u two entries of the following form are added to D where $\mathcal{H} = \{u_1, \ldots, u_m\}$:

$$(ske_u, type := \mathsf{ske}, arg := (), hnd_u := ske_u^{\mathsf{hnd}}, len := 0);$$
$$(pke_u, type := \mathsf{pke}, arg := (), hnd_{u_1} := pke_{u,u_1}^{\mathsf{hnd}}, \ldots,$$
$$hnd_{u_m} := pke_{u,u_m}^{\mathsf{hnd}}, hnd_{\mathsf{a}} := pke_{u,\mathsf{a}}^{\mathsf{hnd}}, len := \mathsf{pke\_len}^*(k)).$$

Here ske_u and pke_u are two consecutive natural numbers and $\mathsf{pke\_len}^*$ is the length function for public keys. Treating the secret key length as 0 is a technicality in [12] and will not matter here. Further, each machine $\mathsf{M}_u^{\mathsf{NS}}$ contains the bitstring u denoting its identity, and the family $(Nonce_{u,v})_{v\in\{1,\dots,n\}}$ of initially empty sets of (nonce) handles.

7.4 On Polynomial Runtime

In order to be valid users of the real cryptographic library, the machines $\mathsf{M}_u^{\mathsf{NS}}$ have to be polynomial-time. We therefore define that each machine $\mathsf{M}_u^{\mathsf{NS}}$ maintains explicit polynomial bounds on the accepted message lengths and the number of inputs accepted at each port. As this is done exactly as in the cryptographic library, we omit the rigorous write-up.

8 The Security Property

Our security property states that an honest participant v only successfully terminates a protocol with an honest participant u if u has indeed started a protocol with v, i.e., an output (ok, u) at $\mathsf{EA\_out}_v!$ can only happen if there was a prior input $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$. This property and also the actual protocol does not consider replay attacks, i.e., a user v could successfully terminate a protocol with u multiple times while u started a protocol with v only once. However, this can easily be avoided as follows: If $\mathsf{M}_u^{\mathsf{NS}}$ receives a message from v containing one of its own nonces, it additionally removes this nonce from the corresponding set, i.e., it removes x_1^{hnd} from $Nonce_{u,v}$ after Steps 2.20 and 2.25. Proving freshness given this change and mutual authentication is useful future work, but better done once the proof has been automated. Warinschi proves these properties [72]. The even stronger property of matching conversations from [19] that he also proves makes constraints on events within the system, not only at the interface. We thus regard it as an overspecification in an approach based on abstraction.

Integrity properties in the underlying model are formally sets of traces at the user interfaces of a system, i.e., here at the port sets $S_{\mathcal{H}}^{\mathsf{EA}}$. Intuitively, an integrity property *Req* contains the "good" traces at these ports. A trace is a sequence of sets of events. We write an event $p?m$ or $p!m$, meaning that message m occurs at in- or output port p. The t-th step of a trace r is written r_t; we speak of the step at time t. The integrity requirement Req^{EA} for the Needham-Schroeder-Lowe protocol is defined as follows, meaning that if v believes to speak with u at time t_1, then there exists a past time t_0 where u started a protocol with v:

Definition 3. *(Entity Authentication Requirement) A trace r is contained in Req^{EA} if for all $u, v \in \mathcal{H}$:*

$$\begin{aligned}&\forall t_1 \in \mathbb{N}\colon \mathsf{EA\_out}_v!(\mathsf{ok}, u) \in r_{t_1}\\ \Rightarrow\ &\exists t_0 < t_1\colon \mathsf{EA\_in}_u?(\mathsf{new\_prot}, v) \in r_{t_0}. \qquad \diamondsuit\end{aligned}$$

The notion of a system *Sys* fulfilling an integrity property *Req* essentially comes in two flavors [4]. *Perfect fulfillment*, $Sys \models^{\mathsf{perf}} Req$, means that the integrity property holds for all traces arising in runs of *Sys* (a well-defined notion from the underlying model [65]). *Computational fulfillment*, $Sys \models^{\mathsf{poly}} Req$, means that the property only holds for polynomially bounded users and adversaries, and that a negligible error probability is permitted. Perfect fulfillment implies computational fulfillment.

The following theorem captures the security of the Needham-Schroeder-Lowe protocol; we prove it in the rest of the paper.

Theorem 2. *(Security of the Needham-Schroeder-Lowe Protocol) For the Needham-Schroeder-Lowe systems from Section 7.2 and the integrity property of Definition 3, we have* $Sys^{\mathsf{NS,id}} \models^{\mathsf{perf}} Req^{\mathsf{EA}}$ *and* $Sys^{\mathsf{NS,real}} \models^{\mathsf{poly}} Req^{\mathsf{EA}}$. □

9 Proof of the Cryptographic Realization from the Idealization

As discussed in the introduction, the idea of our approach is to prove Theorem 2 for the protocol using the ideal Dolev-Yao-style cryptographic library. Then the result for the real system follows automatically. As this paper is the first instantiation of this argument, we describe it in detail.

The notion that a system Sys_1 securely implements another system Sys_2 reactive simulatability (recall the introduction), is written $Sys_1 \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys_2$ (in the computational case). The main result of [12] is therefore

$$Sys^{\mathsf{cry,real}} \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys^{\mathsf{cry,id}}. \tag{1}$$

Since $Sys^{\mathsf{NS,real}}$ and $Sys^{\mathsf{NS,id}}$ are compositions of the same protocol with $Sys^{\mathsf{cry,real}}$ and $Sys^{\mathsf{cry,id}}$, respectively, the composition theorem of [65] and (1) imply

$$Sys^{\mathsf{NS,real}} \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys^{\mathsf{NS,id}}. \tag{2}$$

Showing the theorem's preconditions is easy since the machines $\mathsf{M}_u^{\mathsf{NS}}$ are polynomial-time (see Section 7.4). Finally, the integrity preservation theorem from [4] and (2) imply for every integrity requirement Req that

$$(Sys^{\mathsf{NS,id}} \models^{\mathsf{poly}} Req) \Rightarrow (Sys^{\mathsf{NS,real}} \models^{\mathsf{poly}} Req). \tag{3}$$

Hence if we prove $Sys^{\mathsf{NS,id}} \models^{\mathsf{perf}} Req^{\mathsf{EA}}$, we immediately obtain $Sys^{\mathsf{NS,real}} \models^{\mathsf{poly}} Req^{\mathsf{EA}}$.

10 Proof in the Ideal Setting

This section contains the proof of the ideal part of Theorem 2: We prove that the Needham-Schroeder-Lowe protocol implemented with the ideal, Dolev-Yao-style cryptographic library perfectly fulfills the integrity requirement Req^{EA}. The proof idea is to go backwards in the protocol step by step, and to show that a specific output always requires a specific prior input. For instance, when user v successfully terminates a protocol with user u, then u has sent the third protocol message to v; thus v has sent the second protocol message to u; and so on. The main challenge in this proof was to find suitable invariants on the state of the ideal Needham-Schroeder-Lowe system.

We start by formulating the invariants and then prove the overall entity authentication requirement from the invariants. Finally we prove the invariants, after describing detailed state transitions of the ideal cryptographic library as needed in that proof.

10.1 Invariants

This section contains invariants of the system $Sys^{\mathsf{NS,id}}$, which are used in the proof of Theorem 2. The first invariants, *correct nonce owner* and *unique nonce use*, are easily proved and essentially state that handles contained in a set $Nonce_{u,v}$ indeed point to entries of type nonce, and that no nonce is in two such sets. The next two invariants, *nonce secrecy* and *nonce-list secrecy*, deal with the secrecy of certain terms. They are mainly needed to prove the last invariant, *correct list owner*, which establishes who created certain terms.

– *Correct Nonce Owner.* For all $u \in \mathcal{H}, v \in \{1, \ldots, n\}$ and $x^{\mathsf{hnd}} \in Nonce_{u,v}$, we have $D[hnd_u = x^{\mathsf{hnd}}].type = \mathsf{nonce}$.
– *Unique Nonce Use.* For all $u, v \in \mathcal{H}$, all $w, w' \in \{1, \ldots, n\}$, and all $j \leq size$: If $D[j].hnd_u \in Nonce_{u,w}$ and $D[j].hnd_v \in Nonce_{v,w}$, then $(u, w) = (v, w')$.

Nonce secrecy states that the nonces exchanged between honest users u and v remain secret from all other users and from the adversary, i.e., that the other users and the adversary have no handles to such a nonce:

– *Nonce Secrecy.* For all $u, v \in \mathcal{H}$ and all $j \leq size$: If $D[j].hnd_u \in Nonce_{u,v}$ then $D[j].hnd_w = \downarrow$ for all $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$.

Similarly, the invariant *nonce-list secrecy* states that a list containing such a nonce can only be known to u and v. Further, it states that the identity fields in such lists are correct for Needham-Schroeder-Lowe messages. Moreover, if such a list is an argument of another entry, then this entry is an encryption with the public key of u or v.

– *Nonce-List Secrecy.* For all $u, v \in \mathcal{H}$ and all $j \leq size$ with $D[j].type = \mathsf{list}$: Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ for $i = 1, 2, 3$. If $D[x_i^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$ then:
  a) $D[j].hnd_w = \downarrow$ for all $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$.
  b) If $D[x_{i+1}^{\mathsf{ind}}].type = \mathsf{data}$, then $D[x_{i+1}^{\mathsf{ind}}].arg = (u)$.
  c) For all $k \leq size$ we have $j \in D[k].arg$ only if $D[k].type = \mathsf{enc}$ and $D[k].arg[1] \in \{pke_u, pke_v\}$.

The invariant *correct list owner* states that certain protocol messages can only be constructed by the "intended" users. For instance, if a database entry is structured like the cleartext of a first protocol message, i.e., it is of type list, its first argument belongs to the set $Nonce_{u,v}$, and its second argument is non-cryptographic, i.e., of type data, then it has been created by user u. Similar statements exist for the second and third protocol message.

– *Correct List Owner.* For all $u, v \in \mathcal{H}$ and all $j \leq size$ with $D[j].type = \mathsf{list}$: Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].hnd_u$ for $i = 1, 2$.
  a) If $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ and $D[x_2^{\mathsf{ind}}].type = \mathsf{data}$, then $D[j]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.4.
  b) If $D[x_1^{\mathsf{ind}}].type = \mathsf{nonce}$ and $x_{2,u}^{\mathsf{hnd}} \in Nonce_{u,v}$, then $D[j]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 2.12.
  c) If $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ and $x_2^{\mathsf{ind}} = \downarrow$, then $D[j]$ was created by $\mathsf{M}_v^{\mathsf{NS}}$ in Step 2.21.

This invariant is key for proceeding backwards in the protocol. For instance, if v terminates a protocol with user u, then v must have received a third protocol message. *Correct list owner* implies that this message has been generated by u. Now u only constructs such a message if it received a second protocol message. Applying the invariant two more times shows that u indeed started a protocol with v. The proof described below will take care of the details. Formally, the invariance of the above statements is captured in the following lemma.

Lemma 1. *The statements* correct nonce owner, unique nonce use, nonce secrecy, nonce-list secrecy, *and* correct list owner *are invariants of* $Sys^{\mathsf{NS},\mathsf{id}}$*, i.e., they hold at all times in all runs of* $\{\mathsf{M}_u^{\mathsf{NS}} \mid u \in \mathcal{H}\} \cup \{\mathsf{TH}_\mathcal{H}\}$ *for all* $\mathcal{H} \subseteq \{1, \ldots, n\}$. □

The proof is postponed to Section 10.4.

10.2 Entity Authentication Proof

To increase readability, we partition the proof into several steps with explanations in between. Assume that $u, v \in \mathcal{H}$ and that $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) to its user, i.e., a protocol between u and v has terminated successfully. We first show that this implies that $\mathsf{M}_v^{\mathsf{NS}}$ has received a message corresponding to the third protocol step, i.e., of the form that allows us to apply *correct list owner* to show that it was created by $\mathsf{M}_v^{\mathsf{NS}}$. The following property of $\mathsf{TH}_\mathcal{H}$ proven in [12] will be useful in this proof to show that properties proven for one time also hold at another time.

Lemma 2. *In the ideal cryptographic library* $Sys^{\mathsf{cry},\mathsf{id}}$*, the only modifications to existing entries* x *in* D *are assignments to previously undefined attributes* $x.hnd_u$ *(except for counter updates in entries for signature keys, which we do not have to consider here).* □

Proof. (Ideal part of Theorem 2) Assume that $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) at $\mathsf{EA\_out}_v!$ for $u, v \in \mathcal{H}$ at time t_4. By definition of Algorithms 1 and 2, this can only happen if there was an input $(u, v, \mathsf{i}, m_v^{3\ \mathsf{hnd}})$ at $\mathsf{out}_v?$ at a time $t_3 < t_4$. Here and in the sequel we use the notation of Algorithm 2, but we distinguish the variables from its different executions by a superscript indicating the number of the (claimed) received protocol message, here ^{3}, and give handles an additional subscript for their owner, here v.

The execution of Algorithm 2 for this input must have given $l_v^{3\ \mathsf{hnd}} \neq \downarrow$ in Step 2.2, since it would otherwise abort by Convention 1 without creating an output. Let $l^{3^{\mathsf{ind}}} := D[hnd_v = l_v^{3\ \mathsf{hnd}}].ind$. The algorithm further implies $D[l^{3^{\mathsf{ind}}}].type = \mathsf{list}$. Let $x_i^{3^{\mathsf{ind}}} := D[l^{3^{\mathsf{ind}}}].arg[i]$ for $i = 1, 2$ at the time of Step 2.3. By definition of $\mathsf{list\_proj}$ and since the condition of Step 2.25 is true immediately after Step 2.3, we have

$$x_{1,v}^{3\ \mathsf{hnd}} = D[x_1^{3^{\mathsf{ind}}}].hnd_v \text{ at time } t_4 \tag{4}$$

and

$$x_{1,v}^{3\ \mathsf{hnd}} \in Nonce_{v,u} \wedge x_2^{3^{\mathsf{ind}}} = \downarrow \text{ at time } t_4, \tag{5}$$

since $x_{2,v}^{3\ \mathsf{hnd}} = \downarrow$ after Step 2.3 implies $x_2^{3^{\mathsf{ind}}} = \downarrow$.

This first part of the proof shows that $\mathsf{M}_v^{\mathsf{NS}}$ has received a list corresponding to a third protocol message. Now we apply *correct list owner* to the list entry $D[l^{3^{\mathsf{ind}}}]$ to show that this entry was created by $\mathsf{M}_u^{\mathsf{NS}}$. Then we show that $\mathsf{M}_u^{\mathsf{NS}}$ only generates such an entry if it has received a second protocol message. To show that this message contains a nonce from v, as needed for the next application of *correct list owner*, we exploit the fact that v accepts the same value as its nonce in the third message, which we know from the first part of the proof.

Proof (cont'd with 3rd message). Equations (4) and (5) are the preconditions for Part c) of *correct list owner*. Hence the entry $D[l^{3^{\mathsf{ind}}}]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 2.21.

This algorithm execution must have started with an input $(w, u, \mathsf{i}, m_u^{2^{\mathsf{hnd}}})$ at $\mathsf{out}_u?$ at a time $t_2 < t_3$ with $w \neq u$. As above, we conclude $l_u^{2^{\mathsf{hnd}}} \neq \downarrow$ in Step 2.2, set $l^{2^{\mathsf{ind}}} := D[hnd_u = l_u^{2^{\mathsf{hnd}}}].ind$, and obtain $D[l^{2^{\mathsf{ind}}}].type = \mathsf{list}$. Let $x_i^{2^{\mathsf{ind}}} := D[l^{2^{\mathsf{ind}}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3. As the condition of Step 2.16 is true immediately afterwards, we obtain $x_{i,u}^{2^{\mathsf{hnd}}} \neq \downarrow$ for $i \in \{1, 2, 3\}$. The definition of list_proj and Lemma 2 imply

$$x_{i,u}^{2^{\mathsf{hnd}}} = D[x_i^{2^{\mathsf{ind}}}].hnd_u \text{ for } i \in \{1, 2, 3\} \text{ at time } t_4. \tag{6}$$

Step 2.18 ensures $x_3^2 = w$ and $x_{1,u}^{2^{\mathsf{hnd}}} \in Nonce_{u,w}$. Thus *correct nonce owner* implies

$$D[x_1^{2^{\mathsf{ind}}}].type = \mathsf{nonce}. \tag{7}$$

Now we exploit that $\mathsf{M}_u^{\mathsf{NS}}$ creates the entry $D[l^{3^{\mathsf{ind}}}]$ in Step 2.21 with the input $\mathsf{list}(x_{2,u}^{2^{\mathsf{hnd}}})$. With the definitions of list and list_proj this implies $x_2^{2^{\mathsf{ind}}} = x_1^{3^{\mathsf{ind}}}$. Thus Equations (4) and (5) imply

$$D[x_2^{2^{\mathsf{ind}}}].hnd_v \in Nonce_{v,u} \text{ at time } t_4. \tag{8}$$

We have now shown that $\mathsf{M}_u^{\mathsf{NS}}$ has received a list corresponding to the second protocol message. We apply *correct list owner* to show that $\mathsf{M}_v^{\mathsf{NS}}$ created this list, and again we can show that this can only happen if $\mathsf{M}_v^{\mathsf{NS}}$ received a suitable first protocol message. Further, the next part of the proof shows that $w = v$ and thus $\mathsf{M}_u^{\mathsf{NS}}$ got the second protocol message from $\mathsf{M}_v^{\mathsf{NS}}$, which remained open in the previous proof part.

Proof (cont'd with 2nd message). Equations (6) to (8) are the preconditions for Part b) of *correct list owner*. Thus the entry $D[l^{2^{\mathsf{ind}}}]$ was created by $\mathsf{M}_v^{\mathsf{NS}}$ in Step 2.12. The construction of this entry in Steps 2.11 and 2.12 implies $x_3^2 = v$ and hence $w = v$ (using the definitions of store and retrieve, and list and list_proj). With the results from before Equation (7) and Lemma 2 we therefore obtain

$$x_3^2 = v \wedge x_{1,u}^{2^{\mathsf{hnd}}} \in Nonce_{u,v} \text{ at time } t_4. \tag{9}$$

The algorithm execution where $\mathsf{M}_v^{\mathsf{NS}}$ creates the entry $D[l^{2^{\mathsf{ind}}}]$ must have started with an input $(w', v, \mathsf{i}, m_v^{1^{\mathsf{hnd}}})$ at $\mathsf{out}_v?$ at a time $t_1 < t_2$ with $w' \neq v$. As above, we conclude $l_v^{1^{\mathsf{hnd}}} \neq \downarrow$ in Step 2.2, set $l^{1^{\mathsf{ind}}} := D[hnd_v = l_v^{1^{\mathsf{hnd}}}].ind$, and obtain

$D[l^{1^{\mathsf{ind}}}].type = \mathsf{list}$. Let $x_i^{1^{\mathsf{ind}}} := D[l^{1^{\mathsf{ind}}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3. As the condition of Step 2.4 is true, we obtain $x_{i,v}^{1^{\mathsf{hnd}}} \neq \downarrow$ for $i \in \{1, 2\}$. Then the definition of $\mathsf{list\_proj}$ and Lemma 2 yield

$$x_{i,v}^{1^{\mathsf{hnd}}} = D[x_i^{1^{\mathsf{ind}}}].hnd_v \text{ for } i \in \{1, 2\} \text{ at time } t_4. \tag{10}$$

When $\mathsf{M}_v^{\mathsf{NS}}$ creates the entry $D[l^{2^{\mathsf{ind}}}]$ in Step 2.12, its input is $\mathsf{list}(x_{1,v}^{1^{\mathsf{hnd}}}, n_v^{\mathsf{hnd}}, v^{\mathsf{hnd}})$. This implies $x_1^{1^{\mathsf{ind}}} = x_1^{2^{\mathsf{ind}}}$ (as above). Thus Equations (6) and (9) imply

$$D[x_1^{1^{\mathsf{ind}}}].hnd_u \in Nonce_{u,v} \text{ at time } t_4. \tag{11}$$

The test in Step 2.6 ensures that $x_2^1 = w' \neq \downarrow$. This implies $D[x_2^{1^{\mathsf{ind}}}].type = \mathsf{data}$ by the definition of $\mathsf{retrieve}$, and therefore with Lemma 2,

$$D[x_2^{1^{\mathsf{ind}}}].type = \mathsf{data} \text{ at time } t_4. \tag{12}$$

We finally apply *correct list owner* again to show that $\mathsf{M}_u^{\mathsf{NS}}$ has generated this list corresponding to a first protocol message. We then show that this message must have been intended for user v, and thus user u has indeed started a protocol with user v.

Proof. (cont'd with 1st message) Equations (10) to (12) are the preconditions for Part a) of *correct list owner*. Thus the entry $D[l^{1^{\mathsf{ind}}}]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.4. The construction of this entry in Steps 1.3 and 1.4 implies $x_2^1 = u$ and hence $w' = u$.

The execution of Algorithm 1 must have started with an input $(\mathsf{new\_prot}, w'')$ at $\mathsf{EA\_in}_u?$ at a time $t_0 < t_1$. We have to show $w'' = v$. When $\mathsf{M}_u^{\mathsf{NS}}$ creates the entry $D[l^{1^{\mathsf{ind}}}]$ in Step 1.4, its input is $\mathsf{list}(n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$ with $n_u^{\mathsf{hnd}} \neq \downarrow$. Hence the definition of $\mathsf{list\_proj}$ implies $D[x_1^{1^{\mathsf{ind}}}].hnd_u = n_u^{\mathsf{hnd}} \in Nonce_{u,w}$. With Equation (11) and *unique nonce use* we conclude $w'' = v$.

In a nutshell, we have shown that for all times t_4 where $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) at $\mathsf{EA\_out}_v!$, there exists a time $t_0 < t_4$ such that $\mathsf{M}_u^{\mathsf{NS}}$ receives an input $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$ at time t_0. This proves Theorem 2.

10.3 Command Evaluation by the Ideal Cryptographic Library

This section contains the definition of the cryptographic commands used for modeling the Needham-Schroeder-Lowe protocol, and the local adversary commands that model the extended capabilities of the adversary as far as needed to prove the invariants. Recall that we deal with top levels of Dolev-Yao-style terms, and that commands typically create a new term with its index, type, arguments, handles, and length functions, or parse an existing term. We present the full definitions of the commands, but the reader can ignore the length functions, which have names x_len. Note that we already defined the commands for generating a nonce and for public-key encryption in Section 3.2, hence we do not repeat them here.

Each input c at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$ should be a list $(cmd, x_1, \dots, x_j)$ with cmd from a fixed list of commands and certain parameter domains. We usually write it $y \leftarrow cmd(x_1, \ldots, x_j)$ with a variable y designating the result that $\mathsf{TH}_{\mathcal{H}}$ returns at $\mathsf{out}_u!$. The algorithm $i^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$ (with side effect) denotes that $\mathsf{TH}_{\mathcal{H}}$ determines a handle i^{hnd} for user u to an entry $D[i]$: If $i^{\mathsf{hnd}} := D[i].hnd_u \neq \downarrow$, it returns that, else it sets and returns $i^{\mathsf{hnd}} := D[i].hnd_u := curhnd_u{+}{+}$. On non-handles, it is the identity function. The function $\mathsf{ind2hnd}_u^*$ applies $\mathsf{ind2hnd}_u$ to each element of a list.

In the following definitions, we assume that a cryptographic command is input at port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$. First, we describe the commands for storing and retrieving data via handles.

- *Storing:* $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$, for $m \in \{0,1\}^{\mathsf{max\_len}(k)}$.
  If $i := D[type = \mathsf{data} \wedge arg = (m)].ind \neq \downarrow$ then return $m^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$. Otherwise if $\mathsf{data\_len}^*(\mathsf{len}(m)) > \mathsf{max\_len}(k)$ return $\downarrow$. Else set $m^{\mathsf{hnd}} := curhnd_u{+}{+}$ and
  $$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{data}, arg := (m), \\ hnd_u := m^{\mathsf{hnd}}, len := \mathsf{data\_len}^*(\mathsf{len}(m))).$$
- *Retrieval:* $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.
  $m := D[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].arg[1]$.

Next we describe list creation and projection. Lists cannot include secret keys of the public-key systems (entries of type ske, sks) because no information about those must be given away.

- *Generate a list:* $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, \ldots, x_j^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.
  Let $x_i := D[hnd_u = x_i^{\mathsf{hnd}}].ind$ for $i = 1, \ldots, j$. If any $D[x_i].type \in \{\mathsf{sks}, \mathsf{ske}\}$, set $l^{\mathsf{hnd}} := \downarrow$.If $l := D[type = \mathsf{list} \wedge arg = (x_1, \ldots, x_j)].ind \neq \downarrow$, then return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$. Otherwise, set $length := \mathsf{list\_len}^*(D[x_1].len, \ldots, D[x_j].len)$ and return $\downarrow$ if $length > \mathsf{max\_len}(k)$. Else set $l^{\mathsf{hnd}} := curhnd_u{+}{+}$ and
  $$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{list}, arg := (x_1, \ldots, \\ x_j), hnd_u := l^{\mathsf{hnd}}, len := length).$$
- *i-th projection:* $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $1 \leq i \leq \mathsf{max\_len}(k)$.
  If $D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].arg = (x_1, \ldots, x_j)$ with $j \geq i$, then $x^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(x_i)$, otherwise $x^{\mathsf{hnd}} := \downarrow$.

Further, we used a command for decrypting a list.

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.
  Let $sk := D[hnd_u = sk^{\mathsf{hnd}} \wedge type = \mathsf{ske}].ind$ and $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$. Return $\downarrow$ if $c = \downarrow$ or $sk = \downarrow$ or $pk := D[c].arg[1] \neq sk + 1$ or $l := D[c].arg[2] = \downarrow$. Else return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.

From the set of local adversary commands, which capture additional commands for the adversary at port $\mathsf{in_a}?$, we only describe the command adv_parse. It allows the adversary to retrieve all information that we do not explicitly require to be hidden. This command returns the type and usually all the abstract arguments of a value (with indices replaced by handles), except in the case of ciphertexts. About the remaining local adversary commands, we only need to know that they do not output handles to already existing entries of type list or nonce.

– *Parameter retrieval:* $(type, arg) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$.
  Let $m := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].ind$ and $type := D[m].type$. In most cases, set $arg := \mathsf{ind2hnd}^*_{\mathsf{a}}(D[m].arg)$. (Recall that this only transforms arguments in $\mathcal{INDS}$.) The only exception is for $type = \mathsf{enc}$ and $D[m].arg$ of the form (pk, l) (a valid ciphertext) and $D[pk-1].hnd_{\mathsf{a}} = \downarrow$ (the adversary does not know the secret key); then $arg := (\mathsf{ind2hnd_a}(pk), D[l].len)$.

We finally describe the command that allows an adversary to send messages on insecure channels. In the command, the adversary sends list l to v, pretending to be u.

– $\mathsf{adv\_send\_i}(u, v, l^{\mathsf{hnd}})$, for $u \in \{1, \ldots, n\}$ and $v \in \mathcal{H}$ at port $\mathsf{in_a}?$.
  Let $l^{\mathsf{ind}} := D[hnd_{\mathsf{a}} = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l^{\mathsf{ind}} \neq \downarrow$, output $(u, v, \mathsf{i}, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at $\mathsf{out}_v!$.

10.4 Proof of the Invariants

We start with the proof of *correct nonce owner.*

*Proof (*Correct nonce owner*).* Let $x^{\mathsf{hnd}} \in Nonce_{u,v}$ for $u \in \mathcal{H}$ and $v \in \{1, \ldots, n\}$. By construction, x^{hnd} has been added to $Nonce_{u,v}$ by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 or Step 2.10. In both cases, x^{hnd} has been generated by the command $\mathsf{gen\_nonce}()$ at some time t, input at port $\mathsf{in}_u?$ of $\mathsf{TH}_{\mathcal{H}}$. Convention 1 implies $x^{\mathsf{hnd}} \neq \downarrow$, as $\mathsf{M}_u^{\mathsf{NS}}$ would abort otherwise and not add x^{hnd} to the set $Nonce_{u,v}$. The definition of $\mathsf{gen\_nonce}$ then implies $D[hnd_u = x^{\mathsf{hnd}}] \neq \downarrow$ and $D[hnd_u = x^{\mathsf{hnd}}].type = \mathsf{nonce}$ at time t. Because of Lemma 2 this also holds at all later times $t' > t$, which finishes the proof.

The following proof of *unique nonce use* is quite similar.

*Proof (*Unique Nonce Use*).* Assume for contradiction that both $D[j].hnd_u \in Nonce_{u,w}$ and $D[j].hnd_v \in Nonce_{v,w'}$ at some time t. Without loss of generality, let t be the first such time and let $D[j].hnd_v \notin Nonce_{v,w'}$ at time $t-1$. By construction, $D[j].hnd_v$ is thus added to $Nonce_{v,w'}$ at time t by Step 1.2 or Step 2.10. In both cases, $D[j].hnd_v$ has been generated by the command $\mathsf{gen\_nonce}()$ at time $t-1$. The definition of $\mathsf{gen\_nonce}$ implies that $D[j]$ is a new entry and $D[j].hnd_v$ its only handle at time $t-1$, and thus also at time t. With *correct nonce owner* this implies $u = v$. Further, $Nonce_{v,w'}$ is the only set into which the new handle $D[j].hnd_v$ is put at times $t-1$ and t. Thus also $w = w'$. This is a contradiction.

In the following, we prove *correct list owner*, *nonce secrecy*, and *nonce-list secrecy* by induction. Hence we assume that all three invariants hold at a particular time t in a run of the system, and show that they still hold at time $t+1$.

*Proof (*Correct list owner*).* Let $u, v \in \mathcal{H}$, $j \leq size$ with $D[j].type = \mathsf{list}$. Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].hnd_u$ for $i = 1, 2$ and assume that $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ for $i = 1$ or $i = 2$ at time $t+1$.

The only possibilities to violate the invariant *correct list owner* are that (1) the entry $D[j]$ is created at time $t+1$ or that (2) the handle $D[j].hnd_u$ is created at time $t+1$ for an entry $D[j]$ that already exists at time t or that (3) the handle $x_{i,u}^{\mathsf{hnd}}$ is added to $Nonce_{u,v}$

at time $t+1$. In all other cases the invariant holds by the induction hypothesis and Lemma 2.

We start with the third case. Assume that $x_{i,u}^{\mathsf{hnd}}$ is added to $Nonce_{u,v}$ at time $t+1$. By construction, this only happens in a transition of $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 and Step 2.10. However, here the entry $D[x_i^{\mathsf{ind}}]$ has been generated by the command gen_nonce input at $\mathsf{in}_u?$ at time t, hence x_i^{ind} cannot be contained as an argument of an entry $D[j]$ at time t. Formally, this corresponds to the fact that D is *well-formed*, i.e., index arguments of an entry are always smaller than the index of the entry itself; this has been shown in [12]. Since a transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify entries in $\mathsf{TH}_{\mathcal{H}}$, this also holds at time $t+1$.

For proving the remaining two cases, assume that $D[j].hnd_u$ is created at time $t+1$ for an already existing entry $D[j]$ or that $D[j]$ is generated at time $t+1$. Because both can only happen in a transition of $\mathsf{TH}_{\mathcal{H}}$, this implies $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ already at time t, since transitions of $\mathsf{TH}_{\mathcal{H}}$ cannot modify the set $Nonce_{u,v}$. Because of $u, v \in \mathcal{H}$, *nonce secrecy* implies $D[x_i^{\mathsf{ind}}].hnd_w \neq \downarrow$ only if $w \in \{u, v\}$. Lists can only be constructed by the basic command list, which requires handles to all its elements. More precisely, if $w \in \mathcal{H} \cup \{\mathsf{a}\}$ creates an entry $D[j']$ with $D[j'].type = \mathsf{list}$ and $(x'_1, \ldots, x'_k) := D[j].arg$ at time $t+1$ then $D[x'_i].hnd_w \neq \downarrow$ for $i = 1, \ldots, k$ already at time t. Applied to the entry $D[j]$, this implies that either u or v have created the entry $D[j]$.

We now only have to show that the entry $D[j]$ has been created by u in the claimed steps. This can easily be seen by inspection of Algorithms 1 and 2. We only show it in detail for the first part of the invariant; it can be proven similarly for the remaining two parts.

Let $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ and $D[x_2^{\mathsf{ind}}].type = \mathsf{data}$. By inspection of Algorithms 1 and 2 and because $D[j].type = \mathsf{list}$, we see that the entry $D[j]$ must have been created by either $\mathsf{M}_u^{\mathsf{NS}}$ or $\mathsf{M}_v^{\mathsf{NS}}$ in Step 1.4. (The remaining list generation commands either only have one element, which implies $x_2^{\mathsf{ind}} = \downarrow$ and hence $D[x_2^{\mathsf{ind}}].type \neq \mathsf{data}$, or we have $D[x_2^{\mathsf{ind}}].type = \mathsf{nonce}$ by construction.) Now assume for contradiction that the entry $D[j]$ has been generated by $\mathsf{M}_v^{\mathsf{NS}}$. This implies that also the entry $D[x_1^{\mathsf{ind}}]$ has been newly generated by the command gen_nonce input at $\mathsf{in}_v?$. However, only $\mathsf{M}_u^{\mathsf{NS}}$ can add a handle to the set $Nonce_{u,v}$ (it is the local state of $\mathsf{M}_u^{\mathsf{NS}}$), but every nonce that $\mathsf{M}_u^{\mathsf{NS}}$ adds to the set $Nonce_{u,v}$ is newly generated by the command gen_nonce input by $\mathsf{M}_u^{\mathsf{NS}}$ by construction. This implies $x_{1,u}^{\mathsf{hnd}} \notin Nonce_{u,v}$ at all times, which yields a contradiction to $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ at time $t+1$. Hence $D[j]$ has been created by user u.

*Proof (*Nonce secrecy*).* Let $u, v \in \mathcal{H}$, $j \leq size$ with $D[j].hnd_u \in Nonce_{u,v}$, and $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$ be given. Because of *correct nonce owner*, we know that $D[j].type = \mathsf{nonce}$. The invariant could only be affected if (1) the handle $D[j].hnd_u$ is put into the set $Nonce_{u,v}$ at time $t+1$ or (2) if a handle for w is added to the entry $D[j]$ at time $t+1$.

For proving the first case, note that the set $Nonce_{u,v}$ is only extended by a handle n_u^{hnd} by $\mathsf{M}_u^{\mathsf{NS}}$ in Steps 1.2 and 2.10. In both cases, n_u^{hnd} has been generated by $\mathsf{TH}_{\mathcal{H}}$ at time t since the command gen_nonce was input at $\mathsf{in}_u?$ at time t. The definition of gen_nonce immediately implies that $D[j].hnd_w = \downarrow$ at time t if $w \neq u$. Moreover, this also holds at time $t+1$ since a transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify handles in $\mathsf{TH}_{\mathcal{H}}$, which finishes the claim for this case.

For proving the second case, we only have to consider those commands that add handles for w to entries of type nonce. These are only the commands $\mathsf{list\_proj}$ or $\mathsf{adv\_parse}$ input at $\mathsf{in}_w?$, where $\mathsf{adv\_parse}$ has to be applied to an entry of type list, since only entries of type list can have arguments which are indices to nonce entries. More precisely, if one of the commands violated the invariant there would exist an entry $D[i]$ at time t such that $D[i].type = \mathsf{list}$, $D[i].hnd_w \neq \downarrow$ and $j \in (x_1^{\mathsf{ind}}, \ldots, x_m^{\mathsf{ind}}) := D[i].arg$. However, both commands do not modify the set $Nonce_{u,v}$, hence we have $D[j].hnd_u \in Nonce_{u,v}$ already at time t. Now *nonce secrecy* yields $D[j].hnd_w = \downarrow$ at time t and hence also at all times $< t$ because of Lemma 2. This implies that the entry $D[i]$ must have been created by either u or v, since generating a list presupposes handles for all elements (cf. the previous proof). Assume without loss of generality that $D[i]$ has been generated by u. By inspection of Algorithms 1 and 2, this immediately implies $j \in (x_1^{\mathsf{ind}}, x_2^{\mathsf{ind}})$, since handles to nonces only occur as first or second element in a list generation by u. Because of $j \in D[i].arg[1, 2]$ and $D[j].hnd_u \in Nonce_{u,v}$ at time t, *nonce-list secrecy* for the entry $D[i]$ implies that $D[i].hnd_w = \downarrow$ at time t. This yields a contradiction.

*Proof (*Nonce-list secrecy*).* Let $u, v \in \mathcal{H}$, $j \leq size$ with $D[j].type = \mathsf{list}$. Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].hnd_u$ for $i = 1, 2$, and $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$. Let $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ for $i = 1$ or $i = 2$.

We first show that the invariant cannot be violated by adding the handle $x_{i,u}^{\mathsf{hnd}}$ to $Nonce_{u,v}$ at time $t + 1$. This can only happen in a transition of $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 or 2.10. As shown in the proof of *correct list owner*, the entry $D[x_i^{\mathsf{ind}}]$ has been generated by $\mathsf{TH}_{\mathcal{H}}$ at time t. Since D is well-formed, this implies that $x_i^{\mathsf{ind}} \notin D[j].arg$ for all entries $D[j]$ that already exist at time t. This also holds for all entries at time $t + 1$, since the transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify entries of $\mathsf{TH}_{\mathcal{H}}$. This yields a contradiction to $x_i^{\mathsf{ind}} = D[j].arg[i]$. Hence we now know that $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ already holds at time t.

Part a) of the invariant can only be affected if a handle for w is added to an entry $D[j]$ that already exists at time t. (Creation of $D[j]$ at time t with a handle for w is impossible as above because that presupposes handles to all arguments, in contradiction to *nonce secrecy*.) The only commands that add new handles for w to existing entries of type list are $\mathsf{list\_proj}$, $\mathsf{decrypt}$, $\mathsf{adv\_parse}$, $\mathsf{send\_i}$, and $\mathsf{adv\_send\_i}$ applied to an entry $D[k]$ with $j \in D[k].arg$. *Nonce-list secrecy* for the entry $D[j]$ at time t then yields $D[k].type = \mathsf{enc}$. Thus the commands $\mathsf{list\_proj}$, $\mathsf{send\_i}$, and $\mathsf{adv\_send\_i}$ do not have to be considered any further. Moreover, *nonce-list secrecy* also yields $D[k].arg[1] \in \{pke_u, pke_v\}$. The secret keys of u and v are not known to $w \notin \{u, v\}$, formally $D[hnd_w = ske_u^{\mathsf{hnd}}] = D[hnd_w = ske_v^{\mathsf{hnd}}] = \downarrow$; this corresponds to the invariant *key secrecy* of [12]. Hence the command $\mathsf{decrypt}$ does not violate the invariant. Finally, the command $\mathsf{adv\_parse}$ applied to an entry of type enc with unknown secret key also does not give a handle to the cleartext list, i.e., to $D[k].arg[2]$, but only outputs its length.

Part b) of the invariant can only be affected if the list entry $D[j]$ is created at time $t + 1$. (By well-formedness, the argument entry $D[x_{i+1}^{\mathsf{ind}}]$ cannot be created after $D[j]$.) As in Part a), it can only be created by a party $w \in \{u, v\}$ because other parties have no handle to the nonce argument. Inspection of Algorithms 1 and 2 shows that this can only happen in Steps 1.4 and 2.12, because all other commands list have only one argument, while our preconditions imply $x_2^{\mathsf{ind}} \neq \downarrow$.

- If the creation is in Step 1.4, the preceding Step 1.2 implies $D[x_1^{\mathsf{ind}}].hnd_w \in Nonce_{w,w}$ for some w' and Step 1.3 implies $D[x_2^{\mathsf{ind}}].type = \mathsf{data}$. Thus the preconditions of Part b) of the invariant can only hold for $i = 1$, and thus $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$. Now *unique nonce use* implies $u = w$. Thus Steps 1.3 and 1.4 yield $D[x_2^{\mathsf{ind}}].arg = (u)$.
- If the creation is in Step 2.12, the preceding steps 2.10 and 2.11 imply that the preconditions of Part b) of the invariant can only hold for $i = 2$. Then the precondition, Step 2.10, and *unique nonce use* imply $u = w$. Finally, Steps 2.11 and 2.12 yield $D[x_3^{\mathsf{ind}}].arg = (u)$.

Part c) of the invariant can only be violated if a new entry $D[k]$ is created at time $t + 1$ with $j \in D[k].arg$ (by Lemma 2 and well-formedness). As $D[j]$ already exists at time t, *nonce-list secrecy* for $D[j]$ implies $D[j].hnd_w = \downarrow$ for $w \notin \{u, v\}$ at time t. We can easily see by inspection of the commands that the new entry $D[k]$ must have been created by one of the commands list and encrypt (or by sign, which creates a signature), since entries newly created by other commands cannot have arguments that are indices of entries of type list. Since all these commands entered at a port $\mathsf{in}_z?$ presuppose $D[j].hnd_z \neq \downarrow$, the entry $D[k]$ is created by $w \in \{u, v\}$ at time $t + 1$. However, the only steps that can create an entry $D[k]$ with $j \in D[k].arg$ (with the properties demanded for the entry $D[j]$) are Steps 1.5, 2.13, and 2.22. In all these cases, we have $D[k].type = \mathsf{enc}$. Further, we have $D[k].arg[1] = pke_w$ where w' denotes w's current believed partner. We have to show that $w' \in \{u, v\}$.

- Case 1: $D[k]$ is created in Step 1.5. By inspection of Algorithm 1, we see that the precondition of this proof can only be fulfilled for $i = 1$. Then $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$ and $D[x_1^{\mathsf{ind}}].hnd_w \in Nonce_{w,w}$ and *unique nonce use* imply $w' = v$.
- Case 2: $D[k]$ is created in Step 2.13, and $i = 2$. Then $D[x_2^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$ and $D[x_2^{\mathsf{ind}}].hnd_w \in Nonce_{w,w}$ and *unique nonce use* imply $w' = v$.
- Case 3: $D[k]$ is created in Step 2.13, and $i = 1$. This execution of Algorithm 2 must give $l^{\mathsf{hnd}} \neq \downarrow$ in Step 2.2, since it would otherwise abort by Convention 1. Let $l^{\mathsf{ind}} := D[hnd_w = l^{\mathsf{hnd}}].ind$. The algorithm further implies $D[l^{\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{0\,\mathsf{ind}} := D[l^{\mathsf{ind}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3, and let $x_{i,w}^{0\,\mathsf{hnd}}$ be the handles obtained in Step 2.3. As the algorithm does not abort in Steps 2.5 and 2.7, we have $D[x_2^{0\,\mathsf{ind}}].type = \mathsf{data}$ and $D[x_2^{0\,\mathsf{ind}}].arg = (w')$.
  Further, the reuse of $x_{1,w}^{0\,\mathsf{hnd}}$ in Step 2.12 implies $x_1^{0\,\mathsf{ind}} = x_1^{\mathsf{ind}}$. Together with the precondition $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$, the entry $D[l^{\mathsf{ind}}]$ therefore fulfills the conditions of Part b) of *nonce-list secrecy* with $i = 1$. This implies $D[x_2^{0\,\mathsf{ind}}].arg = (u)$, and thus $w' = u$.
- Case 4: $D[k]$ is created in Step 2.22. With Step 2.21, this implies $x_2^{\mathsf{ind}} = \downarrow$ and thus $i = 1$. As in Case 3, this execution of Algorithm 2 must give $l^{\mathsf{hnd}} \neq \downarrow$ in Step 2.2, we set $l^{\mathsf{ind}} := D[hnd_w = l^{\mathsf{hnd}}].ind$, and we have $D[l^{\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{0\,\mathsf{ind}} := D[l^{\mathsf{ind}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3, and let $x_{i,w}^{0\,\mathsf{hnd}}$ be the handles obtained in Step 2.3. As the algorithm does not abort in Steps 2.17 and 2.19, we have $D[x_3^{0\,\mathsf{ind}}].type = \mathsf{data}$ and $D[x_3^{0\,\mathsf{ind}}].arg = (w')$.
  Further, the reuse of $x_{2,w}^{0\,\mathsf{hnd}}$ in Step 2.21 implies $x_2^{0\,\mathsf{ind}} = x_1^{\mathsf{ind}}$. Together with the precondition $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$, the entry $D[l^{\mathsf{ind}}]$ therefore fulfills the con-

dition of Part b) of *nonce-list secrecy* with $i = 2$. This implies $D[x_3^{0^{\mathsf{ind}}}].arg = (u)$, and thus $w' = u$.

Hence in all cases we obtained $w' = u$, i.e., the list containing the nonce was indeed encrypted with the key of an honest participant.

11 Conclusion

We have shown that an ideal cryptographic library, which constitutes a slightly extended Dolev-Yao model, is sound with respect to the commonly accepted cryptographic definitions under arbitrary active attacks and in arbitrary protocol environments. The abstraction is deterministic and does not contain any cryptographic objects, hence it is abstract in the sense needed for theorem provers. Sound means that we can implement the abstraction securely in the cryptographic sense, so that properties proved for the abstraction carry over to the implementation without any further work. We provided one possible implementation whose security is based on provably secure cryptographic systems. We already showed that the library can be extended in a modular way by adding symmetric authentication [13] and symmetric encryption [9].

This soundness of the cryptographic library now allows for a meaningful analysis of protocol properties on the abstract level. We demonstrated this with a proof of the well-known Needham-Schroeder-Lowe public-key protocol. Further, the abstractness of the library makes such an analysis accessible for formal verification techniques. As many protocols commonly analyzed in the literature can be expressed with our library, this enables the first formal, machine-aided verification of these protocols which is not only meaningful for Dolev-Yao-like abstractions, but whose security guarantees are equivalent to the security of the underlying cryptography. This bridges the up-to-now missing link between cryptography and formal methods for arbitrary attacks.

Acknowledgments

We thank Anupam Datta, Martin Hirt, Dennis Hofheinz, Paul Karger, Ralf Küsters, John Mitchell, Jörn Müller-Quade, Andre Scedrov, Matthias Schunter, Victor Shoup, Michael Steiner, Rainer Steinwandt, Dominique Unruh for interesting discussions.

References

1. M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.
2. M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.
3. R. Anderson and R. Needham. Robustness principles for public key protocols. In *Advances in Cryptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 1995.

4. M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 675–686. Springer, 2003.
5. M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Proc. 11th Symposium on Formal Methods Europe (FME 2002)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2002.
6. M. Backes and B. Pfitzmann. Computational probabilistic non-interference. In *Proc. 7th European Symposium on Research in Computer Security (ESORICS)*, volume 2502 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.
7. M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 1–12, 2003. Full version in IACR Cryptology ePrint Archive 2003/121, Jun. 2003, `http://eprint.iacr.org/`.
8. M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. 24th IEEE Symposium on Security & Privacy*, pages 140–152, 2003.
9. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, 2004. Full version in IACR Cryptology ePrint Archive 2004/059, Feb. 2004, `http://eprint.iacr.org/`.
10. M. Backes and B. Pfitzmann. Relating symbolic and cryptographic key secrecy. In *Proc. 26th IEEE Symposium on Security & Privacy*, 2005. Extended version in IACR Cryptology ePrint Archive 2004/300.
11. M. Backes, B. Pfitzmann, M. Steiner, and M. Waidner. Polynomial fairness and liveness. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 160–174, 2002.
12. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/`.
13. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2003. Extended version in IACR Cryptology ePrint Archive 2003/145, Jul. 2003, `http://eprint.iacr.org/`.
14. M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. `http://eprint.iacr.org/`.
15. D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.
16. G. Bella, F. Massacci, and L. C. Paulson. The verification of an industrial payment protocol: The SET purchase phase. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 12–20, 2002.
17. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.
18. M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in ssh: Provably fixing the ssh binary packet protocol. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 1–11, 2002.
19. M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1994.

20. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.
21. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.
22. R. Canetti. A unified framework for analyzing security of protocols. IACR Cryptology ePrint Archive 2000/067, Dec. 2000. `http://eprint.iacr.org/`.
23. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001. Extended version in Cryptology ePrint Archive, Report 2000/67, `http://eprint.iacr.org/`.
24. R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 209–218, 1998.
25. R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). Cryptology ePrint Archive, Report 2004/334, 2004. `http://eprint.iacr.org/`.
26. R. Cramer and I. Damgård. Secure signature schemes based on interactive protocols. In *Advances in Cryptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 297–310. Springer, 1995.
27. R. Cramer and I. Damgård. New generation of secure and practical RSA-based signatures. In *Advances in Cryptology: CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 173–185. Springer, 1996.
28. R. Cramer and V. Shoup. Practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.
29. R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. In *Proc. 6th ACM Conference on Computer and Communications Security*, pages 46–51, 1999.
30. Z. Dang and R. Kemmerer. Using the ASTRAL model checker for cryptographic protocol analysis. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. `http://dimacs.rutgers.edu/Workshops/Security/`.
31. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.
32. Y. Desmedt and K. Kurosawa. How to break a practical mix and design a new one. In *Advances in Cryptology: EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2000.
33. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
34. B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Proc. International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1997.
35. D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).
36. R. Gennaro, S. Halevi, and T. Rubin. Secure hash-and-sign signatures without the random oracle. In *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 1999.
37. O. Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Advances in Cryptology: CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110. Springer, 1986.

38. S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.
39. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
40. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.
41. J. D. Guttman, F. J. Thayer Fabrega, and L. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 186–195, 2001.
42. J. Herzog. *Computational Soundness of Formal Adversaries*. PhD thesis, MIT, 2002.
43. J. Herzog, M. Liskov, and S. Micali. Plaintext awareness via key registration. In *Advances in Cryptology: CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 548–564. Springer, 2003.
44. M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
45. R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 372–381, 2003.
46. R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.
47. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.
48. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 25th IEEE Symposium on Security & Privacy*, pages 71–85, 2004.
49. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proc. 5th ACM Conference on Computer and Communications Security*, pages 112–121, 1998.
50. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–135, 1995.
51. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.
52. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 18–30, 1997.
53. C. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proc. 4th European Symposium on Research in Computer Security (ESORICS)*, volume 1146 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 1996.
54. S. Micali and P. Rogaway. Secure computation. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404. Springer, 1991.
55. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.
56. J. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 725–733, 1998.
57. J. Mitchell, M. Mitchell, A. Scedrov, and V. Teague. A probabilistic polynominal-time process calculus for analysis of cryptographic protocols (preliminary report). *Electronic Notes in Theoretical Computer Science*, 47:1–31, 2001.

58. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. 18th IEEE Symposium on Security & Privacy*, pages 141–151, 1997.
59. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(21):993–999, 1978.
60. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *Proc. 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.
61. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
62. B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic security of reactive systems. Presented at the *DERA/RHUL Workshop on Secure Architectures and Information Flow*, 1999, Electronic Notes in Theoretical Computer Science (ENTCS), March 2000. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm`.
63. B. Pfitzmann and M. Waidner. How to break and repair a "provably secure" untraceable payment system. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 1992.
64. B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254, 2000. Extended version (with Matthias Schunter) IBM Research Report RZ 3206, May 2000, `http://www.semper.org/sirene/publ/PfSW1_00ReactSimulIBM.ps.gz`.
65. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001. Extended version of the model (with Michael Backes) IACR Cryptology ePrint Archive 2004/082, `http://eprint.iacr.org/`.
66. C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer, 1992.
67. P. Rogaway. Authenticated-encryption with associated-data. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 98–107, 2002.
68. S. Schneider. Verifying authentication protocols with CSP. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–17, 1997.
69. P. Syverson. A new look at an old protocol. *Operation Systems Review*, 30(3):1–4, 1996.
70. F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 19th IEEE Symposium on Security & Privacy*, pages 160–171, 1998.
71. D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.
72. B. Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. In *Proc. 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 248–262, 2003.
73. A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

Model-Based Security Engineering with UML

Jan Jürjens*

Dep. of Informatics, TU Munich, Germany
`http://www4.in.tum.de/~juerjens`

Abstract. Developing security-critical systems is difficult and there are many well-known examples of security weaknesses exploited in practice. Thus a sound methodology supporting secure systems development is urgently needed.

Our aim is to aid the difficult task of developing security-critical systems in a formally based approach using the notation of the Unified Modeling Language. We present the extension UMLsec of UML that allows one to express security-relevant information within the diagrams in a system specification. UMLsec is defined in form of a UML profile using the standard UML extension mechanisms. In particular, the associated constraints give criteria to evaluate the security aspects of a system design, by referring to a formal semantics of a simplified fragment of UML. We explain how these constraints can be formally verified against the dynamic behavior of the specification using automated theorem provers for first-order logic. This formal security verification can also be extended to C code generated from the specifications.

1 Introduction

Modern society and modern economies rely on infrastructures for communication, finance, energy distribution, and transportation. These infrastructures depend increasingly on networked information systems. Attacks against these systems can threaten the economical or even physical well-being of people and organizations. Due to the widespread interconnection of information systems, attacks can be waged anonymously and from a safe distance. Many security incidents have been reported, sometimes with potentially quite severe consequences.

Many problems with security-critical systems arise from the fact that their developers do not always have a strong background in computer security. This is problematic since in practice, security is compromised most often not by breaking mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used. Security mechanisms cannot be "blindly" inserted into a security-critical system, but the overall system development must take security aspects into account.

* This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author(s).

Furthermore, sometimes security mechanisms (such as security protocols) cannot be used off-the-shelf, but have to be designed specifically to satisfy given requirements (see [GHJW03] for an example). Such mechanisms are notoriously hard to design correctly, even for experts, as many examples of protocols designed by experts that were later found to contain flaws show.

Enforcing security requirements is intrinsically subtle, because one has to take into account the interaction of the system with motivated adversaries that act independently. Risks are very hard to calculate because of the possibility to quickly distribute new information to exploit vulnerabilities, for example across the Internet.

Any support to aid secure systems development is thus dearly needed. In particular, it would be desirable to consider security aspects already in the design phase, before a system is actually implemented, since removing security flaws in the design phase, as opposed to patching fielded systems, saves cost and time, reduces security risks and increases customer confidence.

This has motivated a significant amount of successful research into using formal methods for secure systems development. However, part of the difficulty of security-critical systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

The Unified Modeling Language (UML, [UML01], the de facto industry-standard in object-oriented modeling) offers an interesting opportunity for high-quality secure systems development that is feasible in an industrial context.

– As the de facto standard in industrial modeling, a large number of developers is trained in UML.
– Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.

Also, because of its expressitivity and the formal foundation of the UML fragment under consideration, UML gives an interesting theoretical basis for research into open problems in the foundations of security, such as the composability and consistency of the various formalized security requirements. Because our underlying formal system model is largely independent from UML specifics, it provide a suitable platform for such investigations also independently from UML.

To exploit this opportunity, however, some challenges remain which include the following:

– Extending the UML to be able to conveniently express security requirements within a UML specification.
– Defining a formal execution semantics for a sufficient simplified fragment of UML as a basis for the formalization of behavioral security requirements.
– Providing automated tool-support for a formal security verification of UML specifications against the security requirements.

The present work reports on research towards overcoming these challenges.

To support using UML for secure systems development, we define the extension UMLsec of the UML. Various recurring security requirements (such as secrecy, integrity, authenticity and others), as well as assumptions on the security of the system environment, are offered as stereotypes and tags by the UMLsec definition. These can be included in UML diagrams firstly to keep track of the information. Using the associated constraints that refer to a formal semantics of the used simplified fragment of UML, the properties can be used to evaluate diagrams of various kinds and to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy. One can also ensure that the requirements are actually met by the given UML specification of the system. This way we can encapsulate knowledge on prudent security engineering and thereby make it available to developers which may not be specialized in security. One can also go further by checking whether the constraints associated with the UMLsec stereotypes are fulfilled in a given specification, if desired by performing an automated formal security verification using automated theorem provers for first order logic or model-checkers.

Due to space restrictions, we can only present a partial overview on the work. A complete acccount, with many more examples and industrial applications, and the necessary background on distributed system security, can be found in [Jür04].

We explain how to formally evaluate UML specifications for security requirements in Sect. 2. We introduce a fragment of the UMLsec notation in Sect. 3 and explain the various stereotypes with examples. We describe how to use automated theorem provers for first-order logic to verify UML specifications against seurity requirements in Sect. 4. We point to further work applying these analyses to the source-code level in Sect. 5. We report on an industrial application to biometric authentication systems in Sect. 6.

2 Security Evaluation of UML Diagrams

A UMLsec diagram is essentially a UML diagram where security properties and requirements are inserted as stereotypes with tags and constraints, although certain restrictions apply to enable automated formal verification. UML[1] offers three main "light-weight" language extension mechanisms: stereotypes, tagged values, and constraints [UML01]. Stereotypes define new types of modeling elements extending the semantics of existing types or classes in the UML metamodel. Their notation consists of the name of the stereotype written in double angle brackets , attached to the extended model element. This model element is then interpreted according to the meaning ascribed to the stereotype. One way of explicitly defining a property is by attaching a tagged value to a model element. A tagged value is a name-value pair, where the name is referred to as the *tag*. The corresponding notation is $\{tag = value\}$ with the tag name *tag* and a corresponding *value* to be assigned to the tag. If the value is of type Boolean, one

[1] In the following, we consider the UML version 1.5 current at the time of writing; the transition to the upcoming version 2.0 only has a limited impact on the things we discuss here.

usually omits $\{tag = \mathit{false}\}$, and writes $\{tag\}$ instead of $\{tag = \mathit{true}\}$. Another way of adding information to a model element is by attaching logical *constraints* to refine its semantics (for example written in first-order predicate logic).

To construct an extension of the UML one collects the relevant definitions of stereotypes, tagged values, and constraints into a so-called profile [UML01]. For UMLsec, we give validation rules that evaluate a model with respect to listed security requirements. Many security requirements are formulated regarding the behavior of a system in interaction with its environment (in particular, with potential adversaries). To verify these requirements, we use the formal semantics defined in Sect. 2.1.

2.1 Outline of Formal Semantics

For some of the constraints used to define the UMLsec extensions we need to refer to a precisely defined semantics of behavioral aspects, because verifying whether they hold for a given UML model may be mathematically non-trivial. Firstly, the semantics is used to define these constraints in a mathematically precise way. Secondly, we have developed mechanical tool support for analyzing UML specifications for security requirements using model-checkers and automated theorem provers for first-order logic [JS04, JS05, Jür05a]. For this, a precise definition of the meaning of the specifications is necessary. For security analysis, the security-relevant information from the security-oriented stereotypes is then incorporated (cf. Sect. 2.3).

Because of space restrictions, we cannot recall our formal semantics here completely. Instead, we define precisely and explain the interfaces of this semantics that we need here to define the UMLsec profile. More details on the formal semantics of a simplified fragment of UML and on previous and related work in this area can be found in [Jür02, Jür04]. The semantics is defined formally using so-called *UML Machines*, which is an extension of Mealy automata with UML-type communication mechanisms. It includes the following kinds of diagrams:

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *object diagrams*.

Statechart diagrams (or *state diagrams*) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

There is another kind of diagrams, the use case diagrams, which describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed. We will not use it in the following. Additionally to sequence diagrams, there are *collaboration diagrams*, which present similar information. Also, there are *component diagrams*, presenting part of the information contained in deployment diagrams.

The used fragment of UML is simplified to keep automated formal verification that is necessary for some of the more subtle security requirements feasible. Note that in our approach we identify system objects with UML objects, which is suitable for our purposes. Also, as with practically all analysis methods, also in the real-time setting [Wat02], we are mainly concerned with instance-based models. Although, simplified, our choice of a subset of UML is reasonable for our needs, as we have demonstrated in several industrial case-studies (some of which are documented in [Jür04]).

The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. It

– models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and the parameters employed in them,
– provides passing of messages with their parameters between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
– allows in principle whole specification documents to be based on a formal foundation.

In particular, we can compose subsystems by including them into other subsystems.

Objects, and more generally system components, can communicate by exchanging messages. These consist of the message name, and possibly arguments to the message (which will be assumed to be elements of the set **Exp** defined in Sect. 2.2). Message names may be prefixed with object or subsystem instance names. Each object or component may receive messages received in an input queue and release messages to an output queue.

In our model, every object or subsystem instance O has associated multi-sets inQu_O and outQu_O (*event queues*). Our formal semantics models sending a message $msg = op(exp_1, \ldots, exp_n) \in \mathbf{Events}$ from an object or subsystem instance S to an object or subsystem instance R as follows:

(1) S places the message $R.msg$ into its multi-set outQu_S.
(2) A scheduler distributes the messages from out-queues to the intended in-queues (while removing the message head); in particular, $R.msg$ is removed from outQu_S and msg added to inQu_R.
(3) R removes msg from its in-queue and processes its content.

In the case of operation calls, we also need to keep track of the sender to allow sending return signals. This way of modeling communication allows for a very flexible treatment; for example, we can modify the behavior of the scheduler to take account of knowledge on the underlying communication layer (for example regarding security issues, see Sect. 2.3).

At the level of single objects, behavior is modeled using statecharts, or (in special cases such as protocols) possibly as using sequence diagrams. The internal activities contained at states of these statecharts can again be defined each as a statechart, or alternatively, they can be defined directly using UML Machines.

Using subsystems, one can then define the behavior of a system component C by including the behavior of each of the objects or components directly contained in C, and by including an activity diagram that coordinates the respective activities of the various components and objects.

Thus for each object or component C of a given system, our semantics defines a function $[\![C]\!]()$ which

– takes a multi-set I of input messages and a component state S and
– outputs a set $[\![C]\!](I,S)$ of pairs (O,T) where O is a multi-set of output messages and T the new component state (it is a *set* of pairs because of the non-determinism that may arise)

together with an *initial state* S_0 of the component.

Specifically, the behavioral semantics $[\![D]\!]()$ of a statechart diagram D models the run-to-completion semantics of UML statecharts. As a special case, this gives us the semantics for activity diagrams. Any sequence diagram $\mathcal{S}$ gives us the behavior $[\![\mathcal{S}.C]\!]()$ of each contained component C.

Subsystems group together diagrams describing different parts of a system: a system component $\mathcal{C}$ given by a subsystem $\mathcal{S}$ may contain subcomponents $\mathcal{C}_1,\ldots,\mathcal{C}_n$. The behavioral interpretation $[\![\mathcal{S}]\!]()$ of $\mathcal{S}$ is defined as follows:

(1) It takes a multi-set of input events.
(2) The events are distributed from the input multi-set and the link queues connecting the subcomponents and given as arguments to the functions defining the behavior of the intended recipients in $\mathcal{S}$.
(3) The output messages from these functions are distributed to the link queues of the links connecting the sender of a message to the receiver, or given as the output from $[\![\mathcal{S}]\!]()$ when the receiver is not part of $\mathcal{S}$.

When performing security analysis, after the last step, the adversary model may modify the contents of the link queues in a certain way explained in Sect. 2.3.

2.2 Modeling Cryptography

We introduce some sets to be used in modeling cryptographic data in a UML specification and its security analysis.

We assume a set **Keys** with a partial injective map $(\ )^{-1} : \mathbf{Keys} \rightarrow \mathbf{Keys}$. The elements in its domain (which may be public) can be used for encryption and for verifying signatures, those in its range (usually assumed to be secret) for decryption and signing. We assume that every key is either an encryption or decryption key, or both: Any key k satisfying $k^{-1} = k$ is called *symmetric*, the others are called *asymmetric*. We assume that the numbers of symmetric and asymmetric keys are both infinite. We fix infinite sets **Var** of *variables* and **Data** of *data values*. We assume that **Keys**, **Var**, and **Data** are mutually disjoint and that $\mathbf{ASMNames} \cup \mathbf{MsgNm} \subseteq \mathbf{Data}$. **Data** may also include *nonces* and other secrets.

The *algebra of cryptographic expressions* **Exp** is the quotient of the term algebra generated from the set $\mathbf{Var} \cup \mathbf{Keys} \cup \mathbf{Data}$ with the operations

- $\_ :: \_$ (concatenation),
- $\mathbf{head}(\_)$ and $\mathbf{tail}(\_)$,
- $\{\_\}_\_$ (encryption)
- $\mathcal{D}ec_\_(\_)$ (decryption)
- $\mathcal{S}ign_\_(\_)$ (signing)
- $\mathcal{E}xt_\_(\_)$ (extracting from signature)
- $\mathcal{H}ash(\_)$ (hashing)

by factoring out the equations

- $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ (for $K \in \mathbf{Keys}$),
- $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ (for $K \in \mathbf{Keys}$),
- and the usual laws regarding concatenation, **head**(), and **tail**():
  - $(E_1 :: E_2) :: E_3 = E_1 :: (E_2 :: E_3)$ for all $E_1, E_2, E_3 \in \mathbf{Exp}$,
  - $\mathbf{head}(E_1 :: E_2) = E_1$ and $\mathbf{tail}(E_1 :: E_2) = \mathbf{tail}(E_2)$ for all expressions $E_1, E_2 \in \mathbf{Exp}$ such that there exist no E, E' with $E_1 = E :: E'$.

We write $\mathbf{fst}(E) \overset{\text{def}}{=} \mathbf{head}(E)$, $\mathbf{snd}(E) \overset{\text{def}}{=} \mathbf{head}(\mathbf{tail}(E))$, and $\mathbf{thd}(E) \overset{\text{def}}{=} \mathbf{head}(\mathbf{tail}(\mathbf{tail}(E)))$ for each $E \in \mathbf{Exp}$.

This symbolic model for cryptographic operations implies that we assume cryptography to be perfect, in the sense that an adversary cannot "guess" an encrypted value without knowing the decryption key. Also, we assume that one can detect whether an attempted decryption is successful. See for example [AJ01] for a formal discussion of these assumptions.

Based on this formalization of cryptographical operations, important conditions on security-critical data (such as freshness, secrecy, integrity) can then be formulated at the level of UML diagrams in a mathematically precise way (see Sect. 3).

In the following, we will often consider *subalgebras* of **Exp**. These are subsets of **Exp** which are closed under the operations used to define **Exp** (such as concatenation, encryption, decryption etc.). For each subset E of **Exp** there exists a unique smallest (wrt. subset inclusion) **Exp**-subalgebra containing E, which we call **Exp**-*subalgebra generated by* E. Intuitively, it can be constructed from E by iteratively adding all elements in **Exp** reachable by applying the operations used to define **Exp** above. It can be seen as the knowledge one can obtain from a given set E of data by iteratively applying publicly available operations to it (such as concatenation and encryption etc.) and will be used to model the knowledge an attacker may gain from a set E of data obtained for example by eavesdropping on Internet connections.

2.3 Security Analysis of UML Diagrams

Our modular UML semantics allows a rather natural modeling of potential adversary behavior. We can model specific types of adversaries that can attack different parts of the system in a specified way. For example, an attacker of type *insider* may be able to intercept the communication links in a company-wide local area network. We model the actual behavior of the adversary by defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class. This way of reasoning allows an intuitive formulation of many security properties. Since the actual verification is rather indirect this way, we also give alternative intrinsic ways of defining security properties below, which are more manageable, and show that they are equivalent to the earlier ones.

Thus for a security analysis of a given UMLsec subsystem specification $\mathcal{S}$, we need to model potential adversary behavior. We model specific types of adversaries that can attack different parts of the system in a specified way. For this we assume a function $\mathsf{Threats}_A(s)$ which takes an *adversary type* A and a stereotype s and returns a subset of $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ (*abstract threats*). These functions arise from the specification of the physical layer of the system under consideration using deployment diagrams, as explained in Sect. 3. For a link l in a deployment diagram in $\mathcal{S}$, we then define the set $\mathsf{threats}_A^{\mathcal{S}}(l)$ of *concrete threats* to be the smallest set satisfying the following conditions:

If each node n that l is contained in[2] carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$ then:

- If l carries a stereotype s with $\mathsf{delete} \in \mathsf{Threats}_A(s)$ then $\mathsf{delete} \in \mathsf{threats}_A^{\mathcal{S}}(l)$.
- If l carries a stereotype s with $\mathsf{insert} \in \mathsf{Threats}_A(s)$ then $\mathsf{insert} \in \mathsf{threats}_A^{\mathcal{S}}(l)$.
- If l carries a stereotype s with $\mathsf{read} \in \mathsf{Threats}_A(s)$ then $\mathsf{read} \in \mathsf{threats}_A^{\mathcal{S}}(l)$.
- If l is connected to a node that carries a stereotype t with $\mathsf{access} \in \mathsf{Threats}_A(t)$ then $\{\mathsf{delete}, \mathsf{insert}, \mathsf{read}\} \subseteq \mathsf{threats}_A^{\mathcal{S}}(l)$.

The idea is that $\mathsf{threats}_A^{\mathcal{A}}(x)$ specifies the *threat scenario* against a component or link x in the UML Machine System $\mathcal{A}$ that is associated with an adversary type A. On the one hand, the threat scenario determines, which data the adversary can obtain by *accessing* components, on the other hand, it determines, which actions the adversary is permitted by the threat scenario to apply to the concerned links. delete means that the adversary may delete the messages in the

[2] Note that nodes and subsystems may be nested one in another.

corresponding link queue, read allows him to read the messages in the link queue, and insert allows him to insert messages in the link queue.

Then we model the actual behavior of an adversary of type A as a *type A adversary machine.* This is a state machine which has the following data:

- a control state $\mathsf{control} \in \mathsf{State}$,
- a set of *current adversary knowledge* $\mathcal{K} \subseteq \mathbf{Exp}$, and
- for each possible control state $c \in \mathsf{State}$ and set of knowledge $K \subseteq \mathbf{Exp}$, we have
  - a set $\mathsf{Delete}_{c,K}$ which may contain the name of any link l with $\mathsf{delete} \in \mathsf{threats}_A^{\mathcal{S}}(l)$
  - a set $\mathsf{Insert}_{c,K}$ which may contain any pair (l, E) where l is the name of a link with $\mathsf{insert} \in \mathsf{threats}_A^{\mathcal{S}}(l)$, and $E \in \mathcal{K}$, and
  - a set $\mathsf{newState}_{c,k} \subseteq \mathsf{State}$ of states.

The machine is executed from a specified initial state $\mathsf{control} := \mathsf{control}_0$ with a specified *initial knowledge* $\mathcal{K} := \mathcal{K}_0$ iteratively, where each iteration proceeds according to the following steps:

(1) The contents of all link queues belonging to a link l with $\mathsf{read} \in \mathsf{threats}_A^{\mathcal{S}}(l)$ are added to $\mathcal{K}$.
(2) The content of any link queue belonging to a link $l \in \mathsf{Delete}_{\mathsf{control},\mathcal{K}}$ is mapped to $\emptyset$.
(3) The content of any link queue belonging to a link l is enlarged with all expressions E where $(l, E) \in \mathsf{Insert}_{\mathsf{control},\mathcal{K}}$.
(4) The next control state is chosen non-deterministically from the set $\mathsf{newState}_{\mathsf{control},\mathcal{K}}$.

The set $\mathcal{K}_0$ of initial knowledge contains all data values v given in the UML specification under consideration for which each node n containing v carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$. In a given situation, $\mathcal{K}_0$ may also be specified to contain additional data (for example, public encryption keys).

Note that an adversary A able to remove all values sent over the link l (that it, $\mathsf{delete}_l \in \mathsf{threats}_A^{\mathcal{S}}(l)$) may not be able to selectively remove a value e with known meaning from l: For example, the messages sent over the Internet within a virtual private network are encrypted. Thus, an adversary who is unable to break the encryption may be able to delete all messages undiscrimatorily, but not a single message whose meaning would be known to him.

To evaluate the security of the system with respect to the given type of adversary, we then define the *execution of the subsystem $\mathcal{S}$ in presence of an adversary of type A* to be the function $[\![\mathcal{S}]\!]_A()$ defined from $[\![\mathcal{S}]\!]()$ by applying the modifications from the adversary machine to the link queues as a fourth step in the definition of $[\![\mathcal{S}]\!]()$ as follows:

(4) The type A adversary machine is applied to the link queues as detailed above.

Thus after each iteration of the system execution, the adversary may non-deterministically change the contents of link queues in a way depending on the level of physical security as described in the deployment diagram (see Sect. 3).

There are results which simplify the analysis of the adversary behavior defined above, which are useful for developing mechanical tool support, for example to check whether the security properties secrecy and integrity (see below) are provided by a given specification. These are beyond the scope of the current paper and can be found in [Jür04].

One possibility to specify security requirements is to define an idealized system model where the required security property evidently holds (for example, because all links and components are guaranteed to be secure by the physical layer specified in the deployment diagram), and to prove that the system model under consideration is behaviorally equivalent to the idealized one, using a notion of behavioral equivalence of UML models. This is explained in detail in [Jür04].

In the following subsection, we consider alternative ways of specifying the important security properties secrecy and integrity which do not require one to explicitly construct such an idealized system and which are used in the remaining parts of this paper.

2.4 Important Security Properties

The formal definitions of the two main security properties secrecy and integrity considered in this section follow the standard approach of [DY83] and are defined in an intuitive way by incorporating the attacker model.

Secrecy. The formalization of secrecy used in the following relies on the idea that a process specification preserves the secrecy of a piece of data d if the process never sends out any information from which d could be derived, even in interaction with an adversary. More precisely, d is leaked if there is an adversary of the type arising from the given threat scenario that does not initially know d and an input sequence to the system such that after the execution of the system given the input in presence of the adversary, the adversary knows d (where "knowledge", "execution" etc. have to be formalized). Otherwise, d is said to be kept secret.

Thus we come to the following definition.

Definition 1. *We say that a subsystem $\mathcal{S}$* preserves the secrecy *of an expression E from adversaries of type A if E never appears in the knowledge set $\mathcal{K}$ of A during execution of $[\![\mathcal{S}]\!]_A()$.*

This definition is especially convenient to verify if one can give an upper bound for the set of knowledge $\mathcal{K}$, which is often possible when the security-relevant part of the specification of the system $\mathcal{S}$ is given as a sequence of command schemata of the form *await event e – check condition g – output event e'* (for example when using UML sequence diagrams or statecharts for specifying security protocols, see Sect. 3).

Note that this formalization of secrecy is relatively "coarse" in that it may not prevent implicit information flow, but it is comparatively easy to verify and seems to be sufficient in practice [Aba00].

Examples

- The system that sends the expression $\{m\}_K :: K \in \mathbf{Exp}$ over an unprotected Internet link does not preserve the secrecy of m or K against attackers eavesdropping on the Internet, but the system that sends $\{m\}_K$ (and nothing else) does, assuming that it preserves the secrecy of K against attackers eavesdropping on the Internet.
- The system that receives a key K encrypted with its public key over a dedicated communication link and sends back $\{m\}_K$ over the link does not preserve the secrecy of m against attackers eavesdropping on and inserting messages on the link, but does so against attackers that cannot insert messages on the link.

Integrity. The property integrity can be formalized similarly: If during the execution of the system, a system variable gets assigned a value intially only known to the adversary, then the adversary must have caused this variable to contain the value. In that sense the integrity of the variable is violated. (Note that with this definition, integrity is also viewed as violated if the adversary as an honest participant in the interaction is able to change the value, so the definition may have to be adapted in certain circumstances; this is, however, typical for formalizations of security properties.) Thus we say that a system preserves the integrity of a variable v if there is no adversary A such that at some point during the execution of the system with A, v has a value i_0 that is initially known only to A.

Definition 2. *We say that a subsystem $\mathcal{S}$* preserves the integrity *of an attribute a from adversaries of type A with initial knowledge $\mathcal{K}_0$ if during execution of $[\![\mathcal{S}]\!]_A()$, the attribute a never takes on a value appearing in $\mathcal{K}_0$ but not in the specification $\mathcal{S}$.*

The idea of this definition is that $\mathcal{S}$ preserves the integrity of a if no adversary can make a take on a value initially only known to him, in interaction with $\mathcal{A}$. Intuitively, it is the "opposite" of secrecy, in the sense that secrecy prevents the flow of information from protected sources to untrusted recipients, while integrity prevents the flow of information in the other direction. Again, it is a relatively simple definition, which may however not prevent implicit flows of information.

Secure Information Flow. We explain an alternative way of specifying secrecy and integrity like requirements, which gives protection also against partial flow of information, but can be more difficult to deal with, especially when handling with encryption (for which further refinements of the notion are possible, but not needed in the following).

Given a set of messages H and a sequence $\boldsymbol{m}$ of event multi-sets, we write

– $\boldsymbol{m}^H$ for the sequence of event multi-sets derived from those in $\boldsymbol{m}$ by deleting all events the message names of which are *not* in H, and
– $\boldsymbol{m}_H$ for the sequence of event multi-sets derived from those in $\boldsymbol{m}$ by deleting all events the message names of which *are* in H.

Definition 3. *Given a subsystem* $\mathcal{S}$ *and a set of* high *messages* H*, we say that*

– A prevents down-flow *with respect to* H *if for any two sequences* $\boldsymbol{i}, \boldsymbol{j}$ *of event multi-sets and any two output sequences* $\boldsymbol{o} \in [\![\mathcal{S}]\!]_A(\boldsymbol{i})$ *and* $\boldsymbol{p} \in [\![\mathcal{S}]\!]_A(\boldsymbol{j})$*,* $\boldsymbol{i}_H = \boldsymbol{j}_H$ *implies* $\boldsymbol{o}_H = \boldsymbol{p}_H$ *and*
– A prevents up-flow *with respect to* H *if for any two sequences* $\boldsymbol{i}, \boldsymbol{j}$ *of event multi-sets and any two output sequences* $\boldsymbol{o} \in [\![\mathcal{S}]\!]_A(\boldsymbol{i})$ *and* $\boldsymbol{p} \in [\![\mathcal{S}]\!]_A(\boldsymbol{j})$*,* $\boldsymbol{i}^H = \boldsymbol{j}^H$ *implies* $\boldsymbol{o}^H = \boldsymbol{p}^H$ *and*

Intuitively, to prevent down-flow means that outputting a *non-high* (or *low*) message does not depend on *high* inputs (this can be seen as a secrecy requirement for messages marked as high). Conversely, to prevent up-flow means that outputting a *high* value does not depend on *low* inputs (this can be seen as an integrity requirement for messages marked as high).

This notion of *secure information flow* is a generalization of the original notion of noninterference for deterministic systems in [GM82] to system models that are non-deterministic because of underspecification, see [Jür04] for a more detailed discussion.

3 The UMLsec Extension

In Fig. 1 we give some of the stereotypes from UMLsec, together with their tags and constraints. For space reasons, we can only recall part of the UMLsec notation; a complete account can be found in [Jür04]. The constraints are only referred to in the table and formulated (in plain mathematical language) and explained in detail in the remainder of the section. Fig. 2 gives the corresponding tags (which are all DataTags). Note that some of the stereotypes on subsystems refer to stereotypes on model elements contained in the subsystems. For example, the constraint of the data security stereotype refers to contained objects stereotyped as critical (which in turn have tags {secrecy}). The relations between the elements of the tables are explained below in detail.

We explain the stereotypes and tags given in Figures 1 and 2. The constraints are parameterized over the adversary type with respect to which the security requirements should hold; we thus fix an adversary type A to be used in the following. By their nature, some of the constraints can be enforced at the level of abstract syntax (such as secure links), while others refer to the formal definitions in Sect. 2.3 (such as data security). Note that even checking the latter can be mechanized given appropriate tool-support, as explained in Sect. 4.

We give short examples for usage of the stereotypes. To keep the presentation concise, we sometimes give only those fragments of (instances of) subsystems

Stereotype	Base Class	Tags	Constraints	Description
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link,node			LAN connection
wire	link			wire
smart card	node			smart card node
POS device	node			POS device
issuer node	node			issuer node
secure links	subsystem		dependency security matched by links	enforces secure communication links
secrecy	dependency			assumes secrecy
integrity	dependency			assumes integrity
high	dependency			high sensitivity
secure dependency	subsystem		call , send respect data security	structural interaction data security
critical	object, subsystem	secrecy, integrity, high fresh		critical object
no down-flow	subsystem		prevents down-flow	information flow
no up-flow	subsystem		prevents up-flow	information flow
data security	subsystem		provides secrecy, integrity, freshness	basic datasec requirements
fair exchange	subsystem	start,stop	after start eventually reach stop	enforce fair exchange
provable	subsystem	action, cert	action is non-deniable	non-repudiation requirement
guarded access	subsystem		guarded objects accessed through guards	access control using guard objects
guarded	object	guard		guarded object

Fig. 1. UMLsec stereotypes

that are essential to the stereotype in question. Also, we omit presenting the formal semantics and proving the stated properties formally, since the examples are just for illustration.

Internet, encrypted, LAN, wire, smart card, POS device, issuer node These stereotypes on links (resp. nodes) in deployment diagrams denote the corresponding requirements on communication links (resp. system nodes). We require that each link or node carries at most one of these stereotypes. For each adversary type A, we have a function $\mathsf{Threats}_A(s)$ from each stereotype

$$s \in \{ \ \mathsf{wire} \ , \ \mathsf{encrypted} \ , \ \mathsf{LAN} \ , \ \mathsf{smart\ card} \ , \\ \mathsf{POS\ device} \ , \ \mathsf{issuer\ node} \ , \ \mathsf{Internet} \ \}$$

to a set of strings $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ under the following conditions:

Tag	Stereotype	Type	Multipl.	Description
secrecy	critical	String	*	secrecy of data
integrity	critical	String	*	integrity of data
high	critical	String	*	high-level message
fresh	critical	String	*	fresh data
start	fair exchange	String	*	start states
stop	fair exchange	String	*	stop states
action	provable	String	*	provable action
cert	provable	String	*	certificate
guard	guarded	String	1	guard object

Fig. 2. UMLsec tags

– for a node stereotype s, we have $\mathsf{Threats}_A(s) \subseteq \{\mathsf{access}\}$ and
– for a link stereotype s, we have $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$.

Thus $\mathsf{Threats}_A(s)$ specifies which kinds of actions an adversary of type A can apply to node or links stereotyped s. This way we can evaluate UML specifications using the approach explained in Sect. 2.1. We make use of this for the constraints of the remaining stereotypes of the profile.

Examples for threat sets associated with some common adversary types are given in Figures 3 and 4.

Figure 3 gives the *default* attacker, which represents an outsider adversary with modest capability. This kind of attacker is able to read and delete the messages on an Internet link and to insert messages. On an encrypted Internet link (for example a Virtual Private Network), the attacker can delete the messages (without knowing their encrypted content), but not to read the (plaintext) messages or to insert messages (that are encrypted with the right key). Of course, this assumes that the encryption is set up in a way such that the adversary does not get hold of the secret key. The default attacker is assumed not to have direct access to the Local Area Network (LAN) and therefore not be able to eavesdrop on those connections[3], nor on wires connecting security-critical devices (for example, a smart card reader and a display in a point-of-sale (POS) device). Also, smart cards are assumed to be tamper-resistant against default attackers (although they may not be against more sophisticated attackers [And01]). Also, the default attacker is not able to access POS devices or card issuer systems.

Figure 4 defines the *insider* attacker (in the context of an electronic purse system). As an insider, the attacker may access the encrypted Internet link (the assumption is that insiders know the corresponding key) and the local system components.

Secure links. This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency

[3] With more sophistication, even an external adversary may be able to access local connections, but this is assumed to be beyond "default" capabilities.

Stereotype	$\mathsf{Threats}_{default}()$
Internet	{delete, read, insert}
encrypted	{delete}
LAN	$\emptyset$
wire	$\emptyset$
smart card	$\emptyset$
POS device	$\emptyset$
issuer node	$\emptyset$

Fig. 3. Threats from the *default* attacker

Stereotype	$\mathsf{Threats}_{insider}()$
Internet	{delete, read, insert}
encrypted	{delete, read, insert}
LAN	{delete, read, insert}
wire	{delete, read, insert}
smart card	$\emptyset$
POS device	{access}
issuer node	{access}

Fig. 4. Threats from the insider attacker *card issuer*

d with stereotype $s \in \{$ secrecy , integrity , high $\}$ between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype t such that

– in the case of $s =$ high , we have $\mathsf{Threats}_A(t) = \emptyset$,
– in the case of $s =$ secrecy , we have read $\notin \mathsf{Threats}_A(t)$, and
– in the case of $s =$ integrity , we have insert $\notin \mathsf{Threats}_A(t)$.

Example. In Fig. 5, given the *default* adversary type, the constraint for the stereotype secure links is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not give the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated, because the dependency carries the stereotype secrecy , but for the stereotype Internet of corresponding link we have read $\in \mathsf{Threats}_{default}(Internet)$.

Secrecy, Integrity, High. These stereotypes, which may label dependencies in static structure or component diagrams, denote dependencies that are supposed to provide the respective security requirement for the data that is sent along them as arguments or return values of operations or signals. These stereotypes are used in the constraint for the stereotype secure links .

Secrecy. call or send dependencies in object or component diagrams stereotyped secrecy are supposed to provide secrecy for the data that is sent along

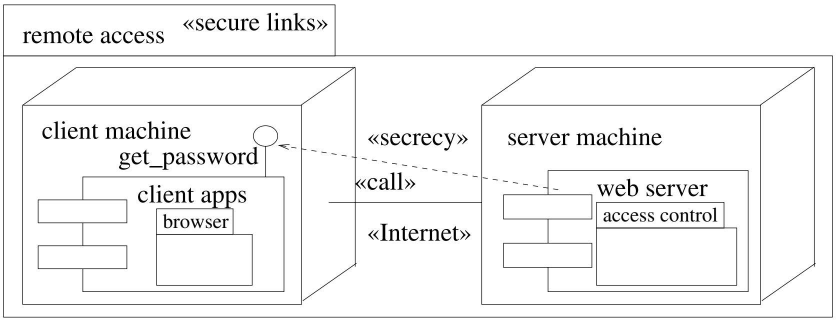


Fig. 5. Example *secure links* usage

them as arguments or return values of operations or signals. This stereotype is used in the constraint for the stereotype secure links .

Secure Dependency. This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the call and send dependencies between objects or subsystems respect the security requirements on the data that may be communicated along them, as given by the tags {secrecy}, {integrity}, and {high} of the stereotype critical . More exactly, the constraint enforced by this stereotype is that if there is a call or send dependency from an object (or subsystem) C to an interface I of an object (or subsystem) D then the following conditions are fulfilled.

– For any message name n in I, n appears in the tag {secrecy} (resp. {integrity} resp. {high}) in C if and only if it does so in D.
– If a message name in I appears in the tag {secrecy} (resp. {integrity} resp. {high}) in C then the dependency is stereotyped secrecy (resp. integrity resp. high).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

Example. Figure 6 shows a key generation subsystem stereotyped with the requirement secure dependency . The given specification violates the constraint for this stereotype, since Random generator and the call dependency do not provide the security levels for random() required by Key generator. More precisely, the constraint is violated, because the message *random* is required to be of high level by Key generator (by the tag {high} in Key generator), but it is not guaranteed to be high level by Random generator (in fact there are no high messages in Random generator and so the tag {high} is missing).

Critical. This stereotype labels objects or subsystem instances containing data that is critical in some way, which is specified in more detail using the corresponding tags. These tags are {secrecy}, {integrity}, {fresh}, and {high}. The values

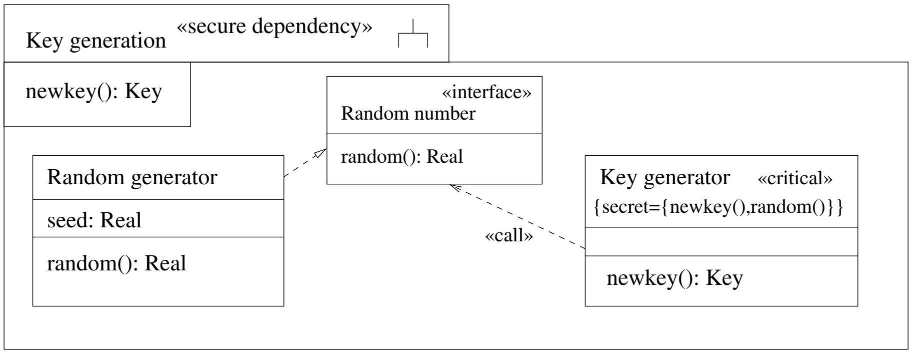


Fig. 6. Key generation subsystem

of the first two are the names of expressions or variables (that is, attributes or message arguments) of the current object the secrecy (resp. integrity) of which is supposed to be protected. The tag {fresh} has data that should be freshly generated as its value. These requirements are enforced by the constraint of the stereotype data security which labels (instances of) subsystems that contain critical objects (see there for an explanation). The tag {high} has the names of messages as values that are supposed to be protected with respect to secure information flow, as enforced by the stereotypes no down − flow and no up − flow .

No Down-Flow. This stereotype of subsystems enforces secure information flow by making use of the associated tag {high}. According to the no down − flow constraint, the stereotyped subsystem prevents down-flow with respect to the messages and their return messages specified in {high}, as defined in Definition 3.

Example. The example in Fig. 7 shows a bank account data object that allows its secret balance to be read using the operation rb() (whose return value is also secret) and written using wb(x). If the balance is over 10000, the object is in a state ExtraService, otherwise in NoExtraService. The state of the object can be queried using the operation rx(). The data object is supposed to be prevented from indirectly leaking out any partial information about *high* data via non-high data, as specified by the stereotype no down − flow . For example, in a situation where government agencies can request information about the existence of bank accounts of a given person, but not their balance, it may be important that the *type* of the account allows no conclusion about its balance. The given specification violates the constraint associated with no down − flow , since partial information about the input of the high operation wb() is leaked out via the return value of the non-high operation rx(). To see how the underlying formalism captures the security flaw using Definition 3, it is sufficient to exhibit sequences $\boldsymbol{i}, \boldsymbol{j}$ of event multi-sets and output sequences $\boldsymbol{o} \in [\![A]\!](\boldsymbol{i})$ and $\boldsymbol{p} \in [\![A]\!](\boldsymbol{j})$ of the UML Machine A giving the behavior of the statechart, with $\boldsymbol{i}_H = \boldsymbol{j}_H$ and $\boldsymbol{o}_H \neq \boldsymbol{p}_H$, where H is the set

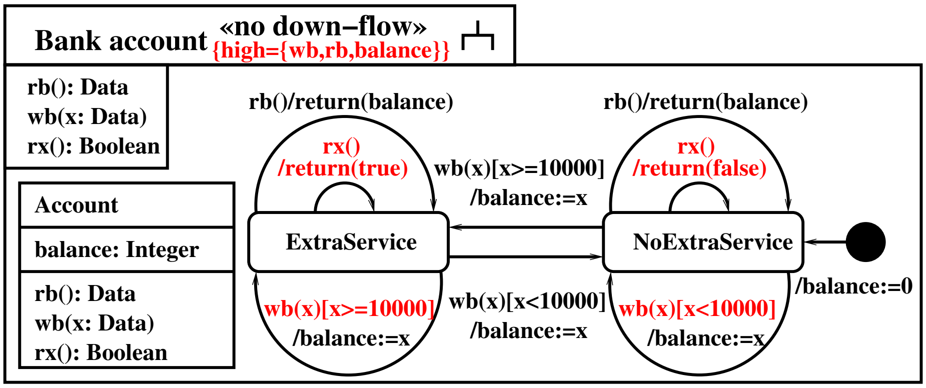


Fig. 7. Bank account data object

of high messages. Consider the sequences $\boldsymbol{i} \overset{\mathrm{def}}{=} (\{\!\!\{ wb(0) \}\!\!\}, \{\!\!\{ rx() \}\!\!\})$ and $\boldsymbol{j} \overset{\mathrm{def}}{=} (\{\!\!\{ wb(10000) \}\!\!\}, \{\!\!\{ rx() \}\!\!\})$. We have $\boldsymbol{i}_H = (\{\!\!\{ \}\!\!\}, \{\!\!\{ rx() \}\!\!\}) = \boldsymbol{j}_H$. From the definition of the behavioral semantics of statecharts sketched in Sect. 2.1, we can see that $\boldsymbol{o} \overset{\mathrm{def}}{=} (\{\!\!\{ \}\!\!\}, \{\!\!\{ return(false) \}\!\!\}) \in [\![A]\!](\boldsymbol{i})$ and $\boldsymbol{p} \overset{\mathrm{def}}{=} (\{\!\!\{ \}\!\!\}, \{\!\!\{ return(true) \}\!\!\}) \in [\![A]\!](\boldsymbol{j})$. But then $\boldsymbol{o}_H = (\{\!\!\{ \}\!\!\}, \{\!\!\{ return(false) \}\!\!\}) \neq (\{\!\!\{ \}\!\!\}, \{\!\!\{ return(true) \}\!\!\}) = \boldsymbol{p}_H$, as required.

Note that, while in the given example, it may be easy to see that the system does not satisfy the no down – flow constraint, it is in general not simple to establish that a system does satisfy this constraint, which is why in [Jür04] we provide the formal semantics sketched in Sect. 2.1.

Data security. This stereotype labeling (instances of) subsystems has the following constraint. The subsystem behavior respects the data security requirements given by the stereotypes critical and the associated tags contained in the subsystem, with respect to the threat scenario arising from the deployment diagram.

More precisely, the constraint is given by the following three conditions (of which the first two use the concepts of preservation of secrecy resp. integrity defined in Sect. 2.3).

secrecy. The subsystem preserves the secrecy of the data designated by the tag {secrecy} against adversaries of type A.

integrity. The subsystem preserves the integrity of the data designated by the tag {integrity} against adversaries of type A.

freshness. Within the subsystem $\mathcal{S}$ stereotyped data security the following condition holds for any subsystem instance or object model $\mathcal{D}$ stereotyped critical for any value *data* of the associated tag {fresh}: *data* occurs within $\mathcal{S}$ at most in
- the object model or subsystem instance model representing $\mathcal{D}$ in the static structure diagram contained in $\mathcal{S}$,
- the swim-lanes belonging to $\mathcal{D}$ in the activitiy diagram contained in $\mathcal{S}$,

– the statechart diagrams contained in $\mathcal{S}$ that model parts of the behavior of $\mathcal{D}$, or
– $\mathcal{D}$'s part of the connections in the sequence diagram contained in $\mathcal{S}$.

Note that it is enough for data to be listed with a security requirement in *one* of the objects or subsystem instances contained in the subsystem to be required to fulfill the above conditions.

Thus the properties of secrecy and integrity are taken relative to the type of adversary under consideration. In case of the default adversary, this is a principal external to the system; one may, however, consider adversaries that are part of the system under consideration, by giving the adversary access to the relevant system components (by defining $\mathsf{Threats}_A(s)$ to contain access for the relevant stereotype s). For example, in an e-commerce protocol involving customer, merchant and bank, one might want to say that the identity of the goods being purchased is a secret known only to the customer and the merchant (and not the bank). This can be formulated by marking the relevant data as "secret" and by performing a security analysis relative to the adversary model "bank" (that is, the adversary is given access to the bank component by defining the $\mathsf{Threats}()$ function in a suitable way).

The secrecy and integrity tags can be used for data values as well as variable and message names (as permitted by the definitions of secrecy and integrity in Sect. 2.3). Note that the adversary does not always have access to the input and output queues of the system (for example, if the system under consideration is part of a larger system it is connected through a secure connection). Therefore it may make sense to use the secrecy tag on variables that are assigned values received by the system; that is, effectively, one may require values that are received to be secret. Of course, the above condition only ensures that the component under consideration keeps the values received by the environment secret; additionally, one has to make sure that the environment (for example, the rest of the system apart from the component under consideration) does not make these values available to the adversary.

Example. The example in Fig. 8 shows the specification of a very simple security protocol. The sender requests the public key K together with the certificate $\mathcal{S}ign_{K_{CA}}(rcv :: K)$ certifying authenticity of the key from the receiver and sends the data d back encrypted under K (here $\{M\}_K$ is the encryption of the message M with the key K, $\mathcal{D}ec_K(C)$ is the decryption of the ciphertext C using K, $\mathcal{S}ign_K(M)$ is the signature of the message M with K, and $\mathcal{E}xt_K(S)$ is the extraction of the data from the signature using K). Assuming the *default* adversary type and by referring to the adversary model outlined in Sect. 2.3 and by using the formal semantics defined in [Jür04], one can establish that the secrecy of d is preserved. (Note that the protocol only serves as a simple example for the use of patterns, not to propose a new protocol of practical value.) Recall from Sect. 2.4 that the requirements {$\mathsf{secrecy}$} and {$\mathsf{integrity}$} refer to the type of adversary under consideration. In the case of the default adversary, in this example this is an adversary that has access to the Internet link between the two nodes only. It does not have direct access to any of the components in the specification (this

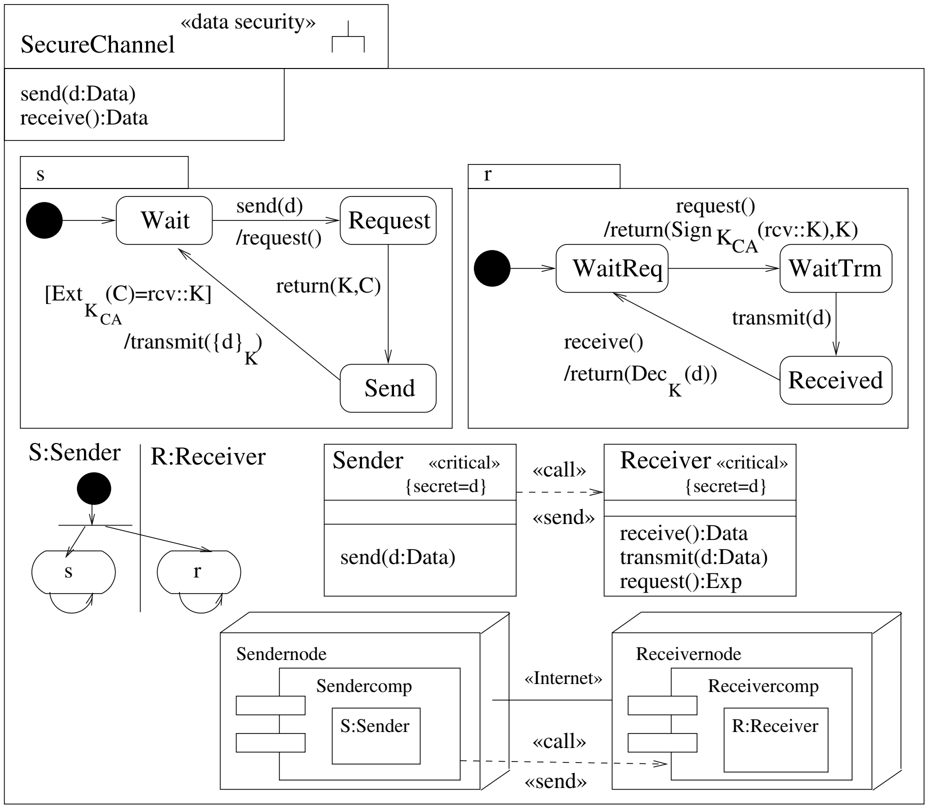


Fig. 8. Security protocol

would have to be specified explicitly using the Threats() function). In particular, the adversary to be considered here does not have access to the components R and S (if it would, then secrecy and integrity would fail because the adversary could read and modify the critical values directly as attributes of R and S).

Again, verifying that a given system satisfies the data security constraint is in general non-trivial, even for small specifications as the example above. We therefore provided tool-support for an automated formal verification to assist this task in Sect. 4. Note also that, while it is often possibly to use standard security protocols (such as SSL), which may already be verified, our work in industrial projects has shown that for a variety of reasons, self-designed protocols are still developed and used in industry (see for example [GHJW03]).

The stereotypes secure links , secure dependencies , and data security describe different conditions for ensuring secure data communication: secure links ensures that the security requirements on the communication dependencies between components are supported by the physical situation, relative to the adversary model under consideration. The stereotype secure dependencies ensures that the security requirements in different parts of a static structure diagram are consistent. Finally, data security ensures that security is enforced on the behavior level. – One could for example merge the conditions of secure links and secure dependencies to give one stereotype; we keep them separate to facilitate understanding and because one might like to use the stereotype secure dependencies in situations where no implementation diagram is present.

Fair Exchange. This stereotype of subsystems has associated tags {start} and {stop} taking names of states as values. The associated constraint requires that, whenever a {start} state in the contained activity diagram is reached, then eventually a corresponding {stop} state will be reached. This allows one to formalize the fair exchange requirement as explained in [Jür04]. This is formalized for a given subsystem $\mathcal{S}$ as follows. $\mathcal{S}$ fulfills the constraint of fair exchange if for every adversary *adv* of type A and every sequence of input event multi-sets $I_1, \ldots, I_n$, the following implication holds: For any state specified by {start} that the function associated with $\mathcal{S}$ reaches, it subsequently eventually reaches a state specified by {stop}.

Provable. A subsystem instance $\mathcal{S}$ may be labelled provable with associated tags {action}, and {cert}, to specify that $\mathcal{S}$ may output the expression $E \in \mathbf{Exp}$ given in {cert} (which serves as a proof that the action at state {action} was performed) only after the state having a name given in {action} is reached. Here the certificate in {cert} is assumed to be unique for each subsystem instance. This is formalized as follows: $\mathcal{S}$ fulfills the constraint if

- for each sequence of event multi-sets $I_1, \ldots, I_k$,
- for each adversary *adv* of type A, and
- for each sequence $(O_1, \ldots, O_k)$ output by $\mathcal{S}$ when executed with an adversary *adv* on input of $(I_1, \ldots, I_k)$,
- and if $(S_1, \ldots, S_k)$ is the corresponding sequence of executed states,

the following implication holds: If there exists an i such that the output O_i equals to the expression in {certificate}, then we have $j \leq i$ such that the state multi-set S_j contains the state specified by action. Again, more explanation can be found in [Jür04].

Example. Fig. 9 gives a subsystem instance describing the following situation: a customer buys a good from a business. The semantics of the stereotype fair exchange is, intuitively, that the actions listed in the tags {start} and {stop} should be linked in the sense that if the former is executed then eventually the latter will be. This would entail that, once the customer has paid, either the order is delivered to him by the due date, or he is able to reclaim the payment on that date. To avoid illegitimate repayment claims, one could employ the stereotype provable with regards to the state Pay, in order to make sure that the Reclaim payment action checks whether the Customer can provide a proof of payment.

Guarded Access. This stereotype of (instances of) subsystems is supposed to mean that each object in the subsystem that is stereotyped guarded can only

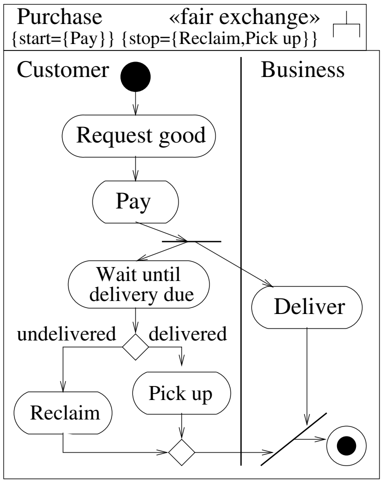


Fig. 9. Purchase activity diagram

be accessed through the objects specified by the tag {guard} attached to the guarded object. This way, one can define access control policies, similar the approach taken in the Java 2 security architecture. Formally, we assume that we have $name \notin \mathcal{K}_A^p$ for the adversary type A under consideration and each name *name* of an instance of a guarded object (that is, a reference is not publicly available), and that for each guarded object there is a statechart specification of an object whose name is given in the associated tag {guard}. This way, we model the passing of references. This is explained in detail in [Jür04].

3.1 Discussion

We shortly discuss the aspects of security covered by the UMLsec extension.

Security requirements. Formalizations of basic security requirements are provided via stereotypes, such as secrecy and integrity .

Threat scenarios. Threat scenarios are incorporated using the formal semantics and depending on the underlying physical layer via the sets $\mathsf{Threats}_{adv}(ster)$ of actions available to the adversary of kind *adv*.

Security concepts. We have shown how to incorporate security concepts such as tamper-resistant hardware (using threat scenarios, in this case).

Security mechanisms. As an example, we demonstrated modeling of the Java security architecture access control mechanisms.

Security primitives. Security primitives are either built in (such as encryption) or can be treated (such as security protocols).

Underlying physical security. This can be addressed as demonstrated by the stereotype secure link in deployment diagrams.

Security management. This can be considered in our approach by using activity diagrams (as in Fig. 9).

Additional domain knowledge has been incorporated regarding Java security and CORBA applications, as well as smart card security (see [Jür04] for more details).

Note that when adapting a modeling language to security requirements, one needs to make sure that the features used to express security properties on the design level actually map to system constructs on the implementation level which do provide these properties. Since we assume, for example, that attributes can only be accessed through the operations of an object, and that only the explicitly offered operations of a subsystem can be called from outside it, it is generally security-critical that this is enforced on the implementation level.

4 Formal Security Verification of UML Models

We present some work on automated formal verification of the security requirements expressed in the UMLsec notation. This tool-support is embedded in a framework supporting the construction of automated requirements analysis tools for UML diagrams. The framework is connected to industrial CASE tools using data integration with XMI [XMI02] and allows convenient access to this data and to the human user. In this chapter, we will, as an example for a usage of this framework, present verification routines to verify the constraints associated with the stereotypes of UMLsec using automated theorem provers (ATPs).

The goal is, in particular, that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes, in a way that allows them to concentrate on the verification logic (rather than on user interface issues).

The usage of the framework as illustrated in Fig. 10 proceeds as follows. The developer creates a model and stores it in the UML 1.5/XMI 1.2 file format.[4] The file is imported by the UML verification framework into the internal MDR repository. MDR is an XMI-specific data-binding library which directly provides a representation of an XMI file on the abstraction level of a UML model through Java interfaces (JMI). This allows the developer to operate directly with UML concepts, such as classes, statecharts, and stereotypes. It is part of the Netbeans project [Net03]. Each plug-in accesses the model through the JMI interfaces generated by the MDR library, they may receive additional textual input, and they may return both a UML model and textual output. The two exemplary analysis plug-ins proceed as follows: The static checker parses the model, verifies its static features, and delivers the results to the error analyzer. The dynamic checker translates the relevant fragments of the UML model into the automated theorem prover input language. The automated theorem prover is spawned by the UML framework as an external process; its results are delivered back to the error analyzer. The error analyzer uses the information received from the static checker and dynamic checker to produce a text report for the developer describing the problems found, and a modified UML model, where the errors

[4] This will be updated to UML 2.0 once the corresponding DTD has been officially released.

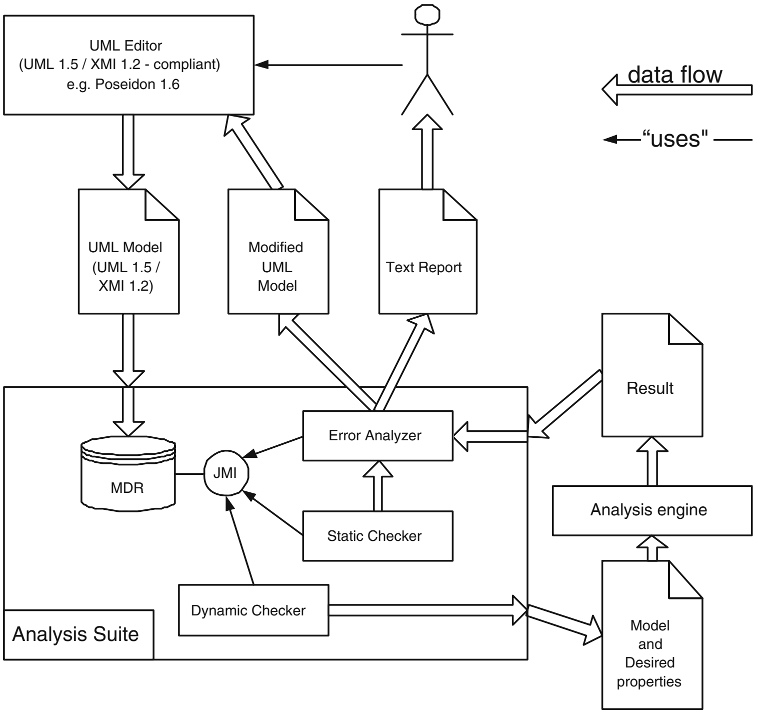


Fig. 10. UML verification framework: usage

found are visualized. Besides the automated theorem prover binding presented in this section there are other analysis plugins including a model-checker binding [JS04] and plugins for simulation and test-sequence generation.

The framework is designed to be extensible: advanced users can define stereotypes, tags, and first-order logic constraints which are then automatically translated to the automated theorem prover for verification on a given UML model. Similarly, new adversary models can be defined.

The user webinterface and the source code of the verification framework is accessible at [UML04].

4.1 Translating UMLsec Diagrams to First-Order Logic Formulas

We explain the automated translation of UMLsec diagrams to first-order logic (FOL) formulas which allows automated analysis of the diagrams using automated first-order logic theorem provers such as e-SETHEO [SW00, MIL$^+$97] or SPASS. More precisely, we assume that we are given a UML package containing the following kinds of diagrams: A deployment diagram specifies the physical layer of the system, such as system nodes and communication links, and the level of security it provides, using UMLsec stereotypes, such as Internet denoting an Internet communication link. From this, in the security analysis, the

$$\begin{aligned}\forall E_1, E_2.\ \big(&\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \\ &\Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big) \\ &\wedge\big(\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big) \\ &\wedge\big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)\big) \\ &\wedge\big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1)\big)\end{aligned}$$

Fig. 11. Some structural formulas

adversary model is generated in first-order logic who is able to control certain communication links. Secondly, a class diagram describes the data structure of the system, including the security requirements on the system data, for example using the UMLsec tags {secrecy}, {integrity} and {authenticity} which represent the respective requirements. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover. The package also contains diagrams specifying the intended behavior of the system, which may include an activity diagram coordinating the components or objects in the package, a sequence diagram specifying interaction between them by message exchange (making use of cryptographic operations using the notation from Sect. 2.2), and statecharts specifying the behavior of single components or objects. The behavioral specifications are compiled to first-order logic axioms giving an abstract interpretation of the system behavior suitable for security analysis. In the following, we explain this translation for sequence diagrams. It works similarly for statecharts and activity diagrams. The formalization automatically derives an upper bound for the set of knowledge the adversary can gain. For space restrictions, we can only present a simplified treatment and have to omit issues such as session key generation.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain secret as specified in the UMLsec model, one thus has to check whether one can derive $\mathsf{knows}(s)$. The set of predicates defined to hold for a given UMLsec specification is defined as follows.

For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption) is captured by the formula in Fig. 11.

For our purposes, a sequence diagram is essentially a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams. Connections are the arrows from the life-line of a source object to the life-line of a target object which are labeled with a message to be sent from the source to the target and a guard condition that has to be fulfilled.

Suppose we are given a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ in a sequence diagram with $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$, where the parameters arg_i of the guard and the message are variables which store the data values exchanged during the course of the protocol. Suppose that the connection l' is the next connection in the sequence diagram with $\mathsf{source}(l') = \mathsf{source}(l)$. For each such connection l, we define a predicate $\mathsf{PRED}(l)$ as in Fig. 12. If such a connection l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Fig. 12.

The formula formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. With this formalization, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulas defined by a protocol. This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means, that one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any practical examples. The advantage is that this approach is rather efficient (see Sect. 4.3 for some performance data).

For each object O in the sequence diagram, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where l is the first connection in the sequence diagram with $\mathsf{source}(l) = O$. The axioms in the overall first-order logic formula for a given sequence diagram are then the conjunction of the formulas representing the publicly known expressions, the formula in Fig. 11, and the conjunction of the formulas $\mathsf{PRED}(O)$ for each object O in the diagram. The conjecture, for which the ATP will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value s is to be kept secret, the conjecture is $\mathsf{knows}(s)$. An example is given in Sect. 4.2.

One can define a variation of the formula in Fig. 12 by joining all subformulas $\mathsf{PRED}(l), \mathsf{PRED}(l'), \ldots$ for connections $l, l', \ldots$ in the sequence diagram using the conjunction operator $\wedge$, instead of including the predicate $\mathsf{PRED}(l')$ for next connection l' in the conclusion of the implication in $\mathsf{PRED}(l)$. The effect is that the order of the connections in the sequence diagram is then ignored. This results in a more coarse abstract interpretation of the sequence diagram than that in Fig. 12, which may produce more false positives: allegedly insecure specifications which are in fact secure in reality, because there the order of the connection is in fact observed. However, in particular architectures the order of messages in the sequence diagram is in fact not enforced, and then this variation is useful. For example, this is the case for the industrial application project which we report on in Sect. 6.

$$\begin{aligned}
&\mathsf{PRED}(l) = \\
&\forall exp_1, \ldots, exp_n \big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \\
&\qquad\qquad \wedge\ cond(exp_1, \ldots, exp_n) \\
&\qquad\qquad \Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n)) \\
&\qquad\qquad\quad \wedge\ \mathsf{PRED}(l')\big)
\end{aligned}$$

Fig. 12. Connection predicate

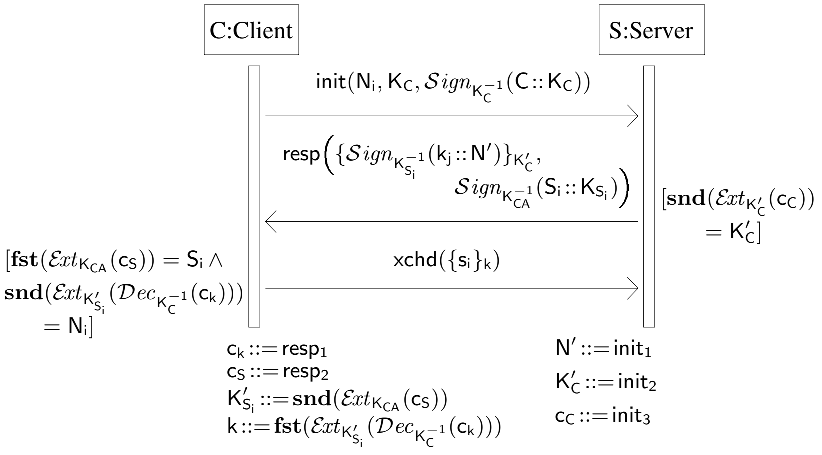

C:Client
S:Server
$\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$
$\mathsf{resp}\Big(\{\mathcal{S}ign_{\mathsf{K_{S_i}^{-1}}}(\mathsf{k_j} :: \mathsf{N'})\}_{\mathsf{K'_C}}, \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}(\mathsf{S_i} :: \mathsf{K_{S_i}})\Big)$
$[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K'_C}}(\mathsf{c_C})) = \mathsf{K'_C}]$
$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S})) = \mathsf{S_i} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K'_{S_i}}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k}))) = \mathsf{N_i}]$
$\mathsf{xchd}(\{\mathsf{s_i}\}_{\mathsf{k}})$
$\mathsf{c_k} ::= \mathsf{resp_1}$
$\mathsf{c_S} ::= \mathsf{resp_2}$
$\mathsf{K'_{S_i}} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))$
$\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K'_{S_i}}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$
$\mathsf{N'} ::= \mathsf{init_1}$
$\mathsf{K'_C} ::= \mathsf{init_2}$
$\mathsf{c_C} ::= \mathsf{init_3}$

Fig. 13. Variant of the TLS handshake

4.2 A Variant of the TLS Protocol

We will analyze a variant of the handshake protocol of TLS[5] examined in [Jür04] (note that this is not the variant of TLS in common use but a variant proposed at the conference IEEE Infocom 1999). To show applicability of our approach, we demonstrate the flaw from [Jür04], suggest a correction, and verify it. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys.

The central part of the specification of this protocol is shown in Fig. 13. Parts that have to be left out here are firstly a deployment diagram specifying that the two protocol participants client and server are connected by an Internet connection, using the UMLsec stereotype Internet . From this, in the security analysis, the adversary model who is able to control this communication link is generated. Secondly, there is a class diagram which includes various security requirements on the protocol data as UMLsec tags {secrecy}, {integrity} and {authenticity}. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover. Most importantly, the value s which is exchanged encrypted in the last message of the protocol is required to remain secret.

Depicted in Fig. 13, the protocol proceeds as we explain in the following. The client C initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$ to the server S. Suppose that the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K_C}}(\mathsf{c_C}))=\mathsf{K'_C}]$ holds, where $\mathsf{K'_C} ::= \mathsf{init_2}$ and $\mathsf{c_C} ::= \mathsf{init_3}$, that is, the key $\mathsf{K_C}$ contained in the signature matches the one transmitted in the clear. Then S sends the message

[5] TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).

```
input_formula(protocol,axiom,(
 ![Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1] : (
 % C <-> Attacker
(      ( ( true & true )
      => knows(conc( n,  conc( k_c,  sign(conc(c, k_c)), inv(k_c)) ) ))
     & ( ( knows(Resp_1) & knows(Resp_2)
     &equal(fst(ext(Resp_2,k_ca)),s)&equal(snd(ext(dec(Resp_1,inv(k_c)),
                snd(ext(Resp_2, k_ca)))), n ) )
=>knows(symenc(s,fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))
)))
 &  % S <-> Attacker
(      ( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
       & equal( snd(ext(Init_3, Init_2)), Init_2 ) )
  => knows(conc(enc(sign(conc(kgen(Init_2),Init_1),inv(k_s)),Init_2),
                sign(conc(s, k_s), inv(k_ca) ) ))
     & ( ( knows(Xchd_1) & true )
        => true
)) )  ) )).
```

Fig. 14. Protocol part of translation to TPTP

$\mathsf{resp}\big(\{\mathcal{S}ign_{\mathsf{K}_\mathsf{S}^{-1}}(\mathsf{k_j} :: \mathsf{N'})\}_{\mathsf{K_C}}, \mathcal{S}ign_{\mathsf{K}_{\mathsf{CA}}^{-1}}(\mathsf{S} :: \mathsf{K_S})\big)$ back to C, where $\mathsf{N'} ::= \mathsf{init_1}$. Now suppose that the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{S_i}}}(\mathcal{D}ec_{\mathsf{K}_\mathsf{C}^{-1}}(\mathsf{c_k})))=\mathsf{N_i}]$$

holds, where $\mathsf{c_S} ::= \mathsf{resp_1}$, $\mathsf{c_k} ::= \mathsf{resp_2}$, and $\mathsf{K'_{S_i}} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))$, that is, the certificate is actually for S and the correct nonce is returned. Then C sends $\mathsf{xchd}(\{\mathsf{s_i}\}_\mathsf{k})$ to S, where $\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{S_i}}}(\mathcal{D}ec_{\mathsf{K}_\mathsf{C}^{-1}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

The main part of the result of the transformation to the e-SETHEO input format TPTP is the protocol definition given in Fig. 14. We have to omit the formulas representing the initial adversary knowledge and the effect of message recombination on the intruder's knowledge predicate knows. The TPTP notation is the de-facto input notation for first-order logic automated theorem provers [SS01], supported, using existing converters, by a variety of provers including also Otter, SPASS, Vampire, and Waldmeister.

Note that in this notation, conjunction is written as &, and forall resp. exists quantification as ![X1, . . . , Xm] resp. ?[X1, . . . , Xm], where X1, . . . ,Xm are the quantified variables. Also, encryption, signature, and concatenation are represented respectively as binary functions enc, sign, and conc in TPTP. The private key belonging to the public key K is written as inv(K). Constants, such as the nonce N, have to be written in small letters in TPTP.

The protocol itself is expressed by a forall quantification over variables representing the arguments of messages which are transferred over the communication link. Here, the message variables Resp_1, and Resp_2 represent the messages received by the client. The message variables Init_1, Init_2, Init_3, and Xchd_1 stand

$$\begin{tikzcd}
C \arrow[r, "N::K_C::\mathcal{S}ign_{K_C^{-1}}(C::K_C)"] & A \arrow[r, "N::K_A::\mathcal{S}ign_{K_A^{-1}}(C::K_A)"] & S \\
C & A \arrow[l, "\{\mathcal{S}ign_{K_S^{-1}}(K::N)\}_{K_C}::\mathcal{S}ign_{K_{\mathsf{CA}}^{-1}}(S::K_S)"'] & S \arrow[l, "\{\mathcal{S}ign_{K_S^{-1}}(K::N)\}_{K_A}::\mathcal{S}ign_{K_{\mathsf{CA}}^{-1}}(S::K_S)"'] \\
C \arrow[r, "\{s\}_K"] & A \arrow[r, "\{s\}_K"] & S
\end{tikzcd}$$

Fig. 15. Attack Visualization: Man-in-the-middle

for the server receiving messages parts. The protocol example includes three messages (cf. Fig. 13), of which the first and third are sent by the client and the second by the server. Each message is expressed by a single implication in the main formula. Therefore three implications occur in Fig. 14 (of which the second is nested in the first). The first,

$$\mathsf{knows(n)}\ \&\ \mathsf{knows(k\_c)}\ \&\ \mathsf{knows(sign(conc(c,k\_c),inv(k\_c)))},$$

is the message sent from the client to the server. It has the precondition true because it is sent unconditionally without previous receipt of any other message. The postconditions of the implications include the messages sent over the communication channel.

4.3 Protocol Analysis with ATPs

We use the ATP e-SETHEO [SW00, MIL$^+$97] for verifying security protocols as a "black box": A TPTP input file is presented to the ATP and an output from the ATP is observed. No internal properties of or information from e-SETHEO is used. This allows one to use e-SETHEO interchangingly with any other ATP accepting TPTP as an input format (such as SPASS, Vampire and Waldmeister) when it may seem fit.

With respect to the security analysis described in Sect. 4.1, the results of the theorem prover have to be interpreted as follows: If the conjecture stating for example that the adversary may get to know the secret can be derived from the axioms which formalize the adversary model and the protocol specification, this means that there may be an attack against the protocol. We then use an attack generation machine programmed in Prolog to construct the attack (also contained in the analysis tool suite [UML04]). If the conjecture cannot be derived from the axioms, this constitutes a proof that the protocol is secure with respect to the security requirement formalized as the negation of the conjecture, because logical derivation is sound and complete with respect to semantic validity for first-order logic. Note that since first-order logic in general is undecidable, it can happen that the ATP is not able to decide whether a given conjecture can be derived from a given set of axioms. However, experience has shown that for a reasonable set of protocols and security requirements, our approach is in fact decidable.

In our example, e-SETHEO returns as an output that the conjection knows(s) can be derived from the defined rules (within one second). For this example the attack tracking tool needs a few seconds to produce the attack. The derived message flow diagram corresponding to a man-in-the-middle attack is depicted in Fig. 15.

We can fix this problem by substituting $K :: N$ in the second message (server to client) by $K :: N :: K_C$ and by including a check regarding this new message part at the client. Now the new version with the additional signature information about the client key k_c can be verified by the automated theorem prover approach. When e-SETHEO runs on the fixed version of the protocol it now gives back the result that the conjecture knows(s) cannot be derived from the axioms formalizing the protocol. Note that this result, which was delivered within a few seconds, means that the actually exists no such derivation, not just that the theorem prover is not able to find it. This means in particular that the attacker cannot gain the secret knowledge anymore. Note that this statement of course in itself is bound to the particular execution model and the formalizations of the security requirements used here. The security analysis may falsely claim that there may be an attack against the specified system, because of the optimizing abstractions used. This, however, has not so far surfaced as a limitation in practical applications.

5 Source Code Analysis

In recent work, we have applied our analysis techniques explained in the previous section to the security verification of cryptographic protocols implemented in C, making use of control flow graphs generated from the source code. This way, one can find security weaknesses which may have been introduced during the (manual or automated) transition from specifications to code. Such security weaknesses may be introduced not only by programming mistakes, but also because some security-relevant details are abstracted away on the specification level. For space restrictions, details have to be omitted but can be found in [Jür05c, Jür05b]. A link between the specification layer and the source code layer has been established in the context of aspect-oriented development in [JH05].

6 Industrial Case-Study: Biometric Authentication

We applied our methods and tools in an industrial application project with a major German company. The goal of the project was the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource, for example by opening a door or letting someone log into a computer system. In this system, a user carries his biometric reference data on a personal smart-card. To gain access, he inserts the smart-card in the card reader and delivers a fresh biometric sample at the biometric sensor, for example a finger-print reader. Since the communication links between the

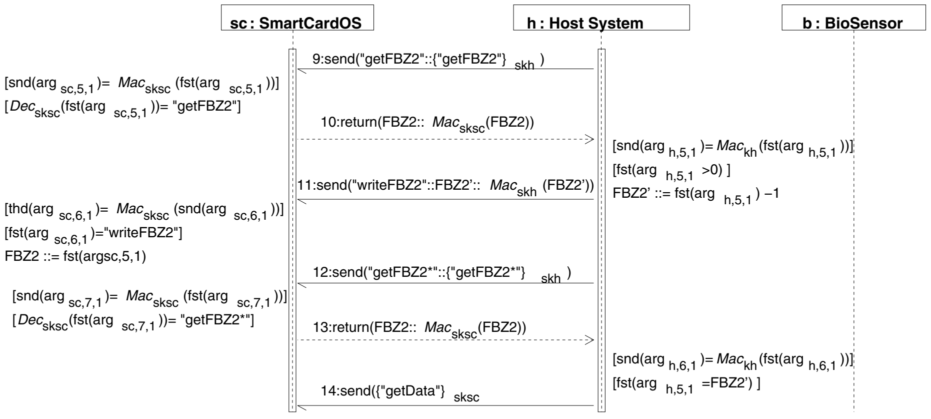

sc : SmartCardOS
h : Host System
b : BioSensor
9:send("getFBZ2"::{"getFBZ2"} skh)
[snd(arg sc,5,1)= Mac sksc (fst(arg sc,5,1))]
[Dec sksc(fst(arg sc,5,1))= "getFBZ2"]
10:return(FBZ2:: Mac sksc(FBZ2))
[snd(arg h,5,1)= Mac kh(fst(arg h,5,1))]
[fst(arg h,5,1 >0)]
FBZ2' ::= fst(arg h,5,1) −1
11:send("writeFBZ2"::FBZ2':: Mac skh (FBZ2'))
[thd(arg sc,6,1)= Mac sksc (snd(arg sc,6,1))]
[fst(arg sc,6,1)="writeFBZ2"]
FBZ2 ::= fst(argsc,5,1)
12:send("getFBZ2*"::{"getFBZ2*"} skh)
[snd(arg sc,7,1)= Mac sksc (fst(arg sc,7,1))]
[Dec sksc(fst(arg sc,7,1))= "getFBZ2*"]
13:return(FBZ2:: Mac sksc(FBZ2))
[snd(arg h,6,1)= Mac kh(fst(arg h,6,1))]
[fst(arg h,5,1 =FBZ2')]
14:send({"getData"} sksc

Fig. 16. Excerpt from biometric authentication protocol

host system (containing the bio-sensor), the card reader, and the smart-card are physically vulnerable, the system needs to make use of a cryptographic protocol to protect this communication. Because the correct design of such protocols and the correct use within the surrounding system is very difficult, our method was chosen to support the development of the biometric authentication system.

Within the project, the system was specified using UML diagrams: a deployment diagram describing the architecture, a class diagram defining the data structure, an activity diagram specifying the general workflow, and a sequence diagram giving a detailed specification of the cryptographic protocol. A fragment of the sequence diagram is shown in Fig. 16.

In the next step, this specification was enriched with security-relevant information, according to the UMLsec extension. This includes specifying the level of security provided by the physical layer of the system in the deployment diagram, and formulating security goals on the execution of the system and on the protection of particular data values in the activity and class diagrams.

Then the security of the protocol was analyzed using the automated tool support described in the previous section. The analysis is done with respect to the threat model which is derived from a deployment diagram of the system and the security goals contained in the class diagram, as explained in the previous sections. This way, it turned out that the protocol in fact contains a vital flaw. To prevent an attack where an attacker simply repeatedly tries to match a forged biometric sample, for example, using an artificial finger, with a forged or stolen smart-card, the protocol contains a misuse counter which is decreased from an initial value of 3 to 0, when the card will be disabled. The attack which was found using our tools enables the attacker to prevent the misuse counter from being updated, thereby enabling a brute-force attack.

The relevant part of the attack is displayed in Fig. 17. The attacker is assumed to control the communication between the smart-card and the host system, which is realistic since it is not protected by physical means. He chooses to act as a relay between the smart-card and the host system, until the host

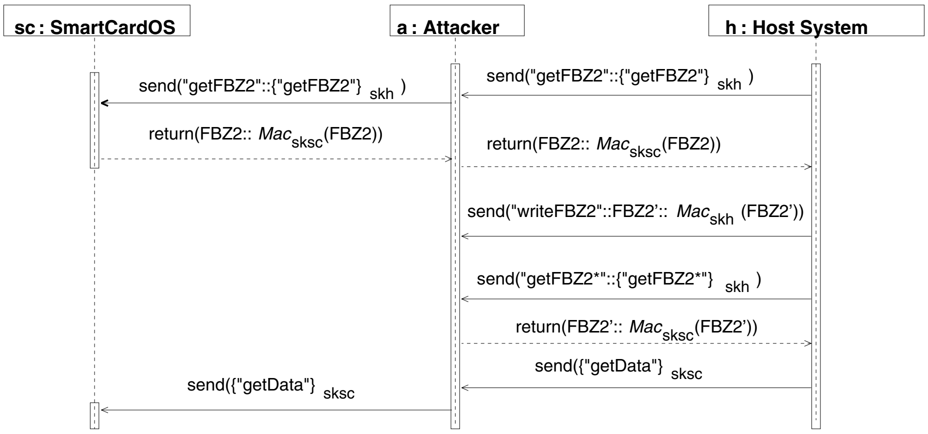

sc : SmartCardOS
a : Attacker
h : Host System
send("getFBZ2"::{"getFBZ2"} skh)
send("getFBZ2"::{"getFBZ2"} skh)
return(FBZ2:: Mac sksc (FBZ2))
return(FBZ2:: Mac sksc (FBZ2))
send("writeFBZ2"::FBZ2':: Mac skh (FBZ2'))
send("getFBZ2*"::{"getFBZ2*"} skh)
return(FBZ2':: Mac sksc (FBZ2'))
send({"getData"} sksc
send({"getData"} sksc

Fig. 17. Attack against biometric authentication protocol

system signals the smart-card to decrease its misuse counter FBZ2 by sending it the message writeFBZ2 which contains the new, decreased value $\mathsf{FBZ2}'$ that the smart-card should assign to its counter. This message is simply dropped by the attacker. Note that it is possible to simply drop the message although the integrity of the message is protected using a Message Authentication Code (MAC) in its third argument $\mathsf{Mac}_{\mathsf{skh}}(\mathsf{FBZ2})$. Here skh is a secret key of the host system, which in a correct protocol is supposed to be equal to the secret key sksc of the smart-card. One should note here that the smart-card does not keep an internal state of the protocol execution history. This means that it accepts any of the messages in the protocol at any point. Therefore, after dropping the message telling the smart-card to decrease its misuse counter, the protocol can simply proceed with the next message from the host system, which is again forwarded by the adversary to the smart-card. This problem had already been detected by our tools at an earlier version of the protocol. To fix it, the protocol was extended with the message getFBZ2 by which the host system tries to make sure that the misuse counter has actually been decreased, as shown in Fig. 16. The return value then expected by the host system from the smart-card is the misuse counter FBZ2, protected in its integrity by also sending the MAC $\mathsf{MAC}_{\mathsf{sksc}}(\mathsf{FBZ2})$, which is supposed to be correctly decreased to give the value FBZ'. Unfortunately, this value had already been sent in the previous message writeFBZ2, since the keys skh and sksc are supposed to be the same, so the adversary only needs to replay the value from that message to the host system.

Based on our findings, the protocol was corrected by using a different one of the coding modes suggested in the specification that makes sure that the return message from the getFBZ2 message cannot be a replay of earlier messages, using a freshly generated random value. This corrected version of the protocol is currently subject to ongoing analysis using our tools.

Since UML was used in the development of this system anyhow, the only extra effort needed was to extend the UML diagrams with the security-critical information describing the level of physical security, and the security goals to be achieved, as explained above. Considering the gain from using our methods, namely detecting several mistakes in various versions of the protocol, and making sure that the final version is correct, this modest extra effort seems to be worthwhile. In conclusion, experiences from this industrial application have been quite positive.

7 Related Work

So far, there seems to be no comparable approach which allows one to include a comparable variety of security requirements in a UML specification which is then, based on a formal semantics, formally verified for these requirements using tools such as automated theorem provers and model-checkers, and which comes with a transition to the source code level where automated formal verification can also be applied.

There has, however, been a substantial amount of work regarding some of the topics we address here (for example formal verification of security-critical systems or secure systems development with UML). A detailed comparison with related work has to be omitted here for space reasons, but can be found in [Jür04]. Many related approaches can also be found in the CSDUML workshop series [JFFuCH04].

8 Conclusion and Future Perspectives

We gave an overview over the extension UMLsec of UML for secure systems development, in the form of a UML profile using the standard UML extension mechanisms. Recurring security requirements are written as stereotypes, the associated constraints ensure the security requirements on the level of the formal semantics, by referring to the threat scenario also given as a stereotype. Thus one may evaluate UML specifications to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy.

At the hand of small examples, we demonstrated how to use UMLsec to model security requirements, threat scenarios, security concepts, security mechanisms, security primitives, underlying physical security, and security management.

As demonstrated, UMLsec can be used to encapsulate established rules on prudent security engineering, also by applying security patterns, and thereby makes them available to developers not specialized in security. While UML was developed to model object-oriented systems, one can also use UMLsec to analyze systems that are not object-oriented (assuming that the underlying assumptions, such as controlled access to data, are ensured).

We also explained how to analyse the UMLsec diagrams against security requirements with respect to their dynamic behavior, using automated theorem provers for first-order logic. We briefly reported on further work to apply this formal security verification to the source code level of implementations derived from the UMLsec specifications.

The definition and evolvement of the UMLsec notation has been based on experiences from in industrial application projects. We reported on the use of UMLsec and its tool-support in one such application, the formal security verification of a biometric authentication system, where several security weaknesses were found and corrected using our approach during its development.

For space restrictions, we could only present a brief overview over a fragment of UMLsec. The complete notation with many more examples and applications can be found in [Jür04].

Although there exists a solid core UMLsec notation now, together with automated formal verification tools, which has proven its usefulness in several industrial application projects, there are a number of interesting open foundational and practical questions still to consider. Because our underlying formal system model is largely independent from UML specifics, it provide a suitable platform for such investigations also independently from UML. For example, one could use the UMLsec framework to formally explore ...

- ... which security properties are preserved under which conditions when composing or decomposing systems in modular components,
- ... the consistency of different security properties expressed as stereotypes when appearing in combination,
- ... which security properties are preserved under which conditions when refining a specification to a more detailed specification or eventually to the source-code,
- ... how to achieve a coherent level of security throughout the abstraction levels of a computing system,
- ... how to evaluate propose standards such as secure reference architectures,
- ... how security requirements can be achieved in the presence of other non-functional requirements such as dependability.

Acknowledgements

The research summarized in this chapter has benefitted from the help of too many people to include here; they are listed in [Jür04].

References

[Aba00] M. Abadi. Security protocols and their properties. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.

[AJ01] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software (4th International Symposium, TACS 2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 82–94. Springer-Verlag, 2001.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, New York, 2001.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[GHJW03] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security-protocols: A bank application. In *SAFECOMP 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy (S&P)*, pages 11–20. IEEE Computer Society, 1982.

[JFFuCH04] J. Jürjens, E.B. Fernandez, R.B. France, and B. Rumpe und C. Heitmeyer. Critical systems development using modelling languages (CSDUML'04): Current development and future challenges (report on the third international workshop). In N. Jardin Nunes, B. Selic, A. Silva, and A. Toval, editors, *UML Modeling Languages and Applications. UML 2004 Satellite Activities, Lisbon, Portugal, October 11–15, 2004, Revised Selected Papers*, volume 3297 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[JH05] J. Jürjens and S.H. Houmb. Dynamic secure aspect modeling with UML: From models to code. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[JS04] J. Jürjens and P. Shabalin. Automated verification of UMLsec models for security requirements. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2004 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425. Springer-Verlag, 2004.

[JS05] J. Jürjens and P. Shabalin. Tools for secure systems development with UML: Security analysis with ATPs. In *FASE 2005*, Lecture Notes in Computer Science, Edinburgh, 2-10 April 2005. Springer-Verlag.

[Jür02] J. Jürjens. Formal semantics for interacting UML subsystems. In B. Jacobs and A. Rensink, editors, *5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 29–44. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2002.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[Jür05a] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[Jür05b] J. Jürjens. Understanding security goals provided by crypto-protocol implementations. In *21st International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005.

[Jür05c] J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*. IEEE Computer Society, 2005.

[MIL$^+$97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning (JAR)*, 18(2):237–246, 1997.

[Net03] Netbeans project. Open source.
Available from `http://mdr.netbeans.org`, 2003.

[SS01] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at `http://www.tptp.org`.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[UML04] UMLsec tool, 2002-04. Open-source. Accessible at http://www.umlsec.org.

[UML01] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at http : //www.omg.org/uml, September 2001.

[Wat02] B. Watson. The Real-time UML standard. In *Real-Time and Embedded Distributed Object Computing Workshop*. OMG, July 15–18 2002.

[XMI02] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification*, January 2002.

A Tutorial on Physical Security and Side-Channel Attacks

François Koeune[1,2] and François-Xavier Standaert[1]

[1] UCL Crypto Group, Place du Levant,
3. 1348 Louvain-la-Neuve, Belgium
fstandae@dice.ucl.ac.be
http://www.dice.ucl.ac.be/crypto/
[2] K2Crypt, Place Verte 60 box 2,
1348 Louvain-la-Neuve, Belgium
fkoeune@k2crypt.com
http://www.k2crypt.com/

Abstract. A recent branch of cryptography focuses on the physical constraints that a real-life cryptographic device must face, and attempts to exploit these constraints (running time, power consumption, ...) to expose the device's secrets. This gave birth to implementation-specific attacks, which often turned out to be much more efficient than the best known cryptanalytic attacks against the underlying primitive as an idealized object. This paper aims at providing a tutorial on the subject, overviewing the main kinds of attacks and highlighting their underlying principles.

1 Introduction and Objectives

A cryptographic primitive can be considered from two points of view: on the one hand, it can be viewed as an abstract mathematical object or black box (i.e. a transformation, possibly parameterized by a key, turning some input into some output); on the other hand, this primitive will *in fine* have to be implemented in a program that will run on a given processor, in a given environment, and will therefore present specific characteristics.

The first point of view is that of "classical" cryptanalysis; the second one is that of *physical security.* Physical attacks on cryptographic devices take advantage of implementation-specific characteristics to recover the secret parameters involved in the computation. They are therefore much less general – since it is specific to a given implementation – but often much more powerful than classical cryptanalysis, and are considered very seriously by cryptographic devices' implementors.

The goal of this paper is to provide the reader with a first tutorial on physical security. The paper will explore certain of the most important kinds of physical attacks, from direct data probing to electromagnetic analysis. However, the intention is not to make an exhaustive review of existing techniques, but rather to highlight the philosophy underlying the main attacks. So, this is not to be viewed a security manual, but as an introductory course in a specific branch of cryptography.

The authors did their best to keep the paper easy to read, giving a good understanding of the general principle of physical attacks. Strict formalism was sometimes sacrificed to the benefit of intuition, whereas many references were provided to guide the interested reader during his first steps in that fascinating and emerging subject.

Physical attacks usually proceed in two steps: an interaction phase, during which an attacker exploits some physical characteristic of a device (e.g. measures running time or current flow, inserts faults, ...) and an exploitation phase, analyzing this information in order to recover secret information. Although we will discuss the first phase, we will mostly focus on the second: once a "signal" has been obtained, how can we exploit this signal to expose a device's secrets?

1.1 Model

The context of a physical attack is the following: we consider a device capable of performing cryptographic operations (e.g. encryptions, signatures, ...) based on a secret key. This key is stored inside the device, and protected from external access. We assume that an attacker has the device at his disposal, and will be able to run it a number of times, possibly with input values of his choice. In addition, during the device's processing, he will be able to act on or measure some parameters related to the environment, the exact nature of which depends on the attack's context. This can for example be the device's running time, the surrounding electromagnetic field, or some way of inducing errors during the computation. The attacker has of course no direct access to the secret key.

Note that the expression "at disposal" might have various meanings: in some cases, it can be a complete gain of control, like for example by stealing an employee's identification badge during his lunch break, attacking it and then putting it back in place to go unnoticed. As another example, we would like to point out that there are situations where the owner of the device himself might be interested in attacking it, e.g. in the case of a pay-TV decoder chip. On the other hand, the control of the attacker on the device might be much more limited: he could for example be hidden behind the corner of the street when the genuine user is using his device, and monitoring electromagnetic radiations from a distance, or interrogating the device through a web interface, and monitoring the delay between request and answer.

Modern cryptography is driven by the well-known Kerckhoffs' assumption, which basically states that all the secret needed to ensure a system's security must be entirely gathered in the secret keys. In other words, we must assume that an attacker has perfect knowledge of the cryptographic algorithm, implementation details, ... The only thing that he does not know – and which is sufficient to guarantee security – is the value of the secret keys. We will adopt this point of view here, and consider that the attacker is familiar with the device under attack, and that recovering the secret keys is sufficient to allow him to build a pirated device with the same power and privileges as the original one.

1.2 Principle of Divide-and-Conquer Attacks

Most of the attacks we will discuss here are divide-and-conquer attacks. As the name says, divide and conquer attacks attempt to recover a secret key by parts. The idea is to find an observable characteristic that can be correlated with a partial key, small enough to make exhaustive search possible. The characteristic is used to validate the partial key, which can be established independently of the rest of the key. The process is then repeated with a characteristic that can be correlated to another part of the key, and so on until the full key is found, or the remaining unknown part is small enough to be in turn exhaustively searchable.

The word characteristic is intentionally imprecise. Such a characteristic can for example be a power consumption profile during a given time interval, a specific output structure after a fault, a probabilistic distribution of running times for an input subset, ...

Divide and conquer attacks can be iterative (that is, a part of the key must be discovered in order to be able to attack subsequent parts) or not (in which case all parts of the key can be guessed independently). Clearly, an iterative attack will be much more sensible to errors, as a wrong conclusion at some point will make subsequent guesses meaningless. This factor can sometimes be mitigated using an error detection/correction strategy [63].

2 Targets

For the sake of concreteness, principles will be illustrated in two contexts: smart cards (typically representing general purpose microprocessors with a fixed bus length) and FPGAs (typically representing application specific devices with parallel computing opportunities).

2.1 Smart Card

One of the most typical targets of side-channel attacks (and one often chosen in the literature) is the smart card. There are several reasons for this. First, these are devices dedicated to performing secure operations. They are also easy to get hold on: usually carried around in many places, small-sized, and an easy target for a pickpocket. In addition, smart cards are pretty easy to scrutinize: as a smart card depends on the reader it is inserted in in many ways (see below), running time, current, ... are easy to monitor. Finally, they are quite simple device, typically running one process at a time, and with a much simpler processor than a desktop PC or mainframe.

Basically, a smart card is a computer embedded in a safe. It consists of a (typically, 8-bit or 32-bit) processor, together with ROM, EEPROM, and a small amount of RAM, which is therefore capable of performing computations. The main goal of a smart card is to allow the execution of cryptographic operations, involving some secret parameter (the key), while not revealing this parameter to the outside world.

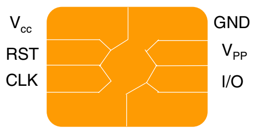


Fig. 1. Smart card chip and its connection points: supply voltage (V_{CC}), reset signal (RST), clock (CLK), ground connection (GND), input/output (I/O) and external voltage for programming (V_{PP}, generally not used)

This processor is embedded in a chip and connected to the outside world through eight wires, the role, use, position, . . . of which is normalized (Fig. 1). In addition to the input/output wires, the parts we will be the most interested in are the following.

Power supply: smart cards do not have an internal battery. The current they need is provided by the smart card reader. This will make the smart card's power consumption pretty easy to measure for an attacker with a rogue reader.

Clock: similarly, smart cards do not dispose of an internal clock either. The clock ticks must also be provided from the outside world. As a consequence, this will allow the attacker to measure the card's running time with very good precision.

Smart cards are usually equipped with protection mechanisms composed of a shield (the passivation layer), whose goal is to hide the internal behavior of the chip, and possibly sensors that react when the shield is removed, by destroying all sensitive data and preventing the card from functioning properly. This will be discussed further below.

We refer the interested reader to several very good books (e.g. [60]) for a deeper discussion of smart cards.

2.2 FPGA

In opposition to smart cards that are typical standardized general purpose circuits, FPGAs are a good example of circuits allowing application specific implementations. Fundamentally, both smart cards and FPGAs (and most processors and ASICs) share the same leakage sources and are consequently similar in terms of susceptibility against physical attacks. However, it is interesting to consider these two contexts for two reasons:

1. They are generally used for completely different applications: smart cards have limited computational power while FPGAs and ASICs are usually required for their ability to deal with high throughput.
2. FPGAs and ASICs allow parallel computing and have a more flexible architecture (as it is under control of the designer). This may affect their resistance against certain attacks.

More philosophically, we used these two contexts as an illustration (among others) that physical attacks are both general and specific: general because they rely

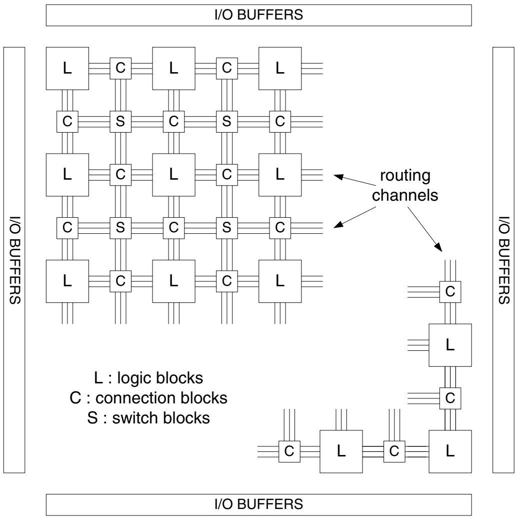


Fig. 2. FPGA: high level view

on physical principles that can be applied to any kind of device; specific because when it comes to their practical implementation, their efficiency depends on how well we can adapt these general principles to a particular target.

FPGAs usually contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, sometimes known as logic blocks, are connected using a set of routing resources that are also programmable (see Figure 2). They can be used to implement a variety of digital processing tasks. FPGAs allow the designer to determine the allocation and scheduling of the tasks in the circuit (*e.g.* to trade surface for speed), which is typically out of control of the smart card programmer.

Compared to ASICs, FPGAs present similar design opportunities, although some parts of reconfigurable circuits are not dedicated to the applications but to the reconfigurability management. In brief, FPGAs trade a little bit of the ASICs efficiency for more flexibility. Structural details on FPGAs are not necessary for the understanding of this survey, but can be useful for mounting attacks in practice. A good reference is [23].

2.3 Differences Between Platforms

Differences between platforms may affect the physical security at two distinct levels. First and practically, the devices may be based on different technologies and consequently have different physical behaviors. For example, certain side-channel attacks require to make predictions of the leakage. The prediction model may be different for different devices. Similarly, fault insertion techniques may have different effects on different technologies. In order to keep our discussions general, we will make abstraction of these possible technological differences and assume in the following that:

(1) Knowing the data handled by a device, it is possible to predict its leakage[1].
(2) Faults can be inserted in the device.

Second, as already mentioned, different platforms may have different architectures, leading to different computational paradigms. For example, smart cards are small processors where the data is managed sequentially. FPGA designs (i.e. the hardware counterpart of smart card programs) have a more flexible architecture and allow parallel computation. Such differences will be emphasized in the following sections.

3 Physical Attacks Classification

Physical attacks can be classified in many ways. The literature usually sorts them along two orthogonal axes.

Invasive vs. non-invasive: invasive attacks require depackaging the chip to get direct access to its inside components; a typical example of this is the connection of a wire on a data bus to see the data transfers. A non-invasive attack only exploits externally available information (the emission of which is however often unintentional) such as running time, power consumption, ... One can go further along this axis by distinguishing **local** and **distant** attacks: a local attack requires close – but external, i.e. non-invasive – proximity to the device under concern, for example by a direct connection to its power supply. As opposed, a distant attack can operate at a larger distance, for example by measuring electromagnetic field several meters (or hundreds of meters) away, or by interacting with the device through an internet connection.

Active vs. passive: active attacks try to tamper with the device's proper functioning; for example, fault-induction attacks will try to induce errors in the computation. As opposed, passive attacks will simply observe the device's behavior during its processing, without disturbing it.

Note that these two axes are well orthogonal: an invasive attack may completely avoid disturbing the device's behavior, and a passive attack may require a preliminary depackaging for the required information to be observable.

These attacks are of course not mutually exclusive: an invasive attack may for example serve as a preliminary step for a non-invasive one, by giving a detailed description of the chip's architecture that helps to find out where to put external probes.

As said in section 2.1, smart cards are usually equipped with protection mechanisms that are supposed to react to invasive attacks (although several

[1] Remark that most present devices are CMOS-based and their leakages are relatively simple and similar to predict. Typically, models based on the Hamming weight or the Hamming distance of the data processed were successfully applied to target respectively smart cards [50] and FPGAs [67]. On the other hand, technological solutions may also be considered to make the attacks harder. Such possibilities will be discussed in Section 9.

invasive attacks are nonetheless capable of defeating these mechanisms). On the other hand, it is worth pointing out that a non-invasive attack is completely undetectable: there is for example no way for a smart card to figure out that its running time is currently being measured. Other countermeasures will therefore be necessary.

The attacks we will consider belong to five major groups.

Probing attacks consist in opening a device in order to directly observe its internal parameters. These are thus invasive, passive attacks.

Fault induction attacks try to influence a device's behavior, in a way that will leak its secrets. The difficulty lies not so much in inducing a fault than in being able to recover secret parameters from the faulty result, and this is the question that will retain most of our attention. These attacks are by essence active, and can be either invasive or non-invasive.

The three last groups are usually denoted as *side-channel attacks*. Their basic idea is to passively observe some physical characteristic during the device's processing, and to use this "side-channel" to derive more information about the processed secret. They are thus passive, and typically non-invasive, although some exceptions exist.

Timing attacks exploit the device's running time.

Power analysis attacks focus on the device's electric consumption.

Electromagnetic analysis attacks measure the electromagnetic field surrounding the device during its processing.

In some sense, timing, power and electromagnetic analysis attacks can be viewed as an evolution in the dimension of the leakage space. Timing attacks exploit a single, scalar information (the running time) for each run. Power attacks provide a one-dimensional view of the device's behavior, namely instant power consumption at each time unit. With the possibility to move the sensor around the attacked device (or to use several sensors), electromagnetic analysis provide a 4-dimensional view: spatial position and time. This allows for example to separate the contributions of various components of the chip, and therefore to study them separately. Moreover, we will see that EM emanations consist of a multiplicity of signals, which can lead to even more information.

Finally, we believe there is a substantial difference between timing or Simple Power Analysis attacks and subsequent side-channel attacks (this will appear clearly in the next sections): timing attacks and Simple Power Analysis provide an indirect access to the data processed, via the observation of the operations performed. As opposed, power or electromagnetic analysis offer direct access to the data processed.

3.1 About the Cost...

From an economical point of view, invasive attacks are usually more expensive to deploy on a large scale, since they require individual processing of each attacked device. In this sense, non-invasive attacks constitute therefore a bigger menace for the smart card industry. According to [66], *"until now, invasive attacks involved a relatively high capital investment for lab equipment plus a moderate investment of effort for each individual chip attacked. Non-invasive attacks require only a moderate capital investment, plus a moderate investment of effort in designing an attack on a particular type of device; thereafter the cost per device attacked is low. [...] semi-invasive attacks can be carried out using very cheap and simple equipment."*

4 Probing

One natural idea when trying to attack a security device is to attempt to depackage it and observe its behavior by branching wires to data buses or observing memory cells with a microscope. These attacks are called probing attacks.

4.1 Measuring Phase

The most difficult part of probing attacks lies in managing to penetrate the device and access its internals. An useful tool for this purpose is a *probing station*. Probing stations consist of microscopes with micromanipulators attached for landing fine probes on the surface of the chip. They are widely used in the semiconductor manufacturing industry for manual testing of production-line samples, and can be obtained second-hand for under US$ 10 000.

To make observation easier, the attacker may try to slow down the clock provided to the chip, so that successive states are easily observable. An introduction on probing attacks can be found in [7], and a good overview of ways to depackage a card and probe its content is given in [44].

As we said before, smart cards are usually protected by a passivation layer, which is basically a shield covering the chip, in order to prevent from observing its behavior. In addition, some smart cards are equipped with detectors, for example in the form of additional metallization layers that form a sensor mesh above the actual circuit and that do not carry any critical signals. All paths of this mesh need to be continuously monitored for interruptions and short-circuits, and the smart card has to refuse processing and destroy sensitive data when an alarm occurs. Similarly, monitoring clock frequency and refusing to operate under abnormally low (or high) frequency should be done to protect the chip. Additional sensors (UV, light, ...) may also be placed.

Security is a permanent fight between attackers and countermeasure designers, and these protection means are not invulnerable. According to Anderson [7], "*the appropriate tool to defeat them is the* Focused Ion Beam Workstation *(FIB). This is a device similar to a scanning electron microscope, but it uses a beam of ions instead of electrons. By varying the beam current, it is possible to use it as a microscope or as a milling machine. By introducing a suitable gas, which is broken down by the ion beam, it is possible to lay down either conductors or insulators with a precision of a few tens of nanometers. Given a FIB, it is* *straightforward to attack a sensor mesh that is not powered up. One simply drills a hole through the mesh to the metal line that carries the desired signal, fills it up with insulator, drills another hole through the center of the insulator, fills it with metal, and plates a contact on top, which is easy to contact with a needle from the probing station*".

Better protection techniques, such as stronger passivation layers, that will make it difficult for the attacker to remove them without damaging the chip itself, are also developed. They complicate the attacker's task, but do not make it impossible yet. An interesting example, discussing how such a stronger passivation layer was defeated, can be found in [56].

4.2 Exploitation Phase

The most obvious target is of course the part of memory where secret keys are stored; similarly, in a software-based device, the attacker can also tape the data buses connecting memory to processor, as he knows that the secret key will of course be processed during the signature (or decryption), and hence transit through that wire. From our pedagogical point of view, this kind of attack is not extremely interesting: being able to access smart card internals might be a strong technical challenge (which is out of our scope), but exploiting this information is straightforward.

Things might get more difficult (and interesting) when only part of the information can be read (for example because technical constraints allow only to tape a part of the data bus, providing two bits of each transferred word), or when countermeasures are at stake, for example bus scrambling, which can be thought as some sort of lightweight encryption used for internal transfer and storage. However, we will not discuss these topics more in the detail here. We refer the interested reader to [31,27,32] for further information.

5 Fault Induction Attacks

When an electronic device stops working correctly, the most natural reaction is to get rid of it. This apparently insignificant habit may have deep impact in cryptography, where faulty computations are sometimes the easiest way to discover a secret key.

As a matter of fact, a recent and powerful cryptanalysis technique consists in tampering with a device in order to have it perform some erroneous operations, hoping that the result of that erroneous behavior will leak information about the secret parameters involved. This is the field of fault induction attacks.

5.1 Types of Faults

The faults can be characterized from several aspects.

Permanent vs. transient: as the name says, a permanent fault damages the cryptographic device in a permanent way, so that it will behave incorrectly in all future computations; such damage includes freezing a memory cell to a constant value, cutting a data bus wire, ... As opposed, with a transient fault, the device is disturbed during its processing, so that it will only perform fault(s) during that specific computation; examples of such disturbances are radioactive bombing, abnormally high or low clock frequency, abnormal voltage in power supply, ...

Error location: some attacks require the ability to induce the fault in a very specific location (memory cell); others allow much more flexibility;

Time of occurrence: similarly, some attacks require to be able to induce the fault at a specific time during the computation, while others do not;

Error type: many types of error may be considered, for example:
- flip the value of some bit or some byte,
- permanently freeze a memory cell to 0 or 1,
- induce (with some probability) flips in memory, but only in one direction (e.g. a bit can be flipped from 1 to 0, but not the opposite),
- prevent a jump from being executed,
- disable instruction decoder,
- ...

As can be guessed, the fault model has much importance regarding the feasibility of an attack. In fact, two types of papers can be found in the literature: the first type deals with the way to induce errors of a given type in current devices; the second basically assumes a (more or less realistic) fault model and deals with the way this model can be exploited to break a cryptosystem, without bothering with the way such faults can be induced in practice. These two types are of course complementary to determine the realism of a new attack and the potential weaknesses induced by a new fault induction method. From the viewpoint we took in this tutorial, we are mostly interested in the second aspect, i.e. how we can exploit faulty computations to recover secret parameters. However, let us first briefly consider the other aspect: fault induction methods.

5.2 Fault Induction Techniques

Faults are induced by acting on the device's environment and putting it in abnormal conditions. Many channels are available to the attacker. Let us review some of them.

Voltage: Unappropriate voltage might of course affect a device's behavior. For example, smart card voltages are defined by ISO standards: a smart card must be able to tolerate on the contact VCC a supply voltage between 4, 5V and 5, 5V, where the standard voltage is specified at 5V. Within this range the smart card must be able to work properly. However, a deviation of the external power supply, called spike, of much more than the specified 10% tolerance might cause problems for a proper functionality of the smart card. Indeed, it will most probably lead to a wrong computation result, provided that the smart card is still able to finish its computation completely.

Clock: Similarly, standards define a reference clock frequency and a tolerance around which a smart card must keep working correctly. Applying an abnormally high or low frequency may of course induce errors in the processing. Blömer and Seifert [10] note that "*a finely tuned clock glitch is able to completely change a CPU's execution behavior including the omitting of instructions during the executions of programs*". Note that, as opposed to the clock slowing down described in section 4, whose goal was to make internal state easier to observe, this clock variation may be very brief, in order to induce a single faulty instruction or to try to fool clock change detectors.

Temperature: Having the device process in extreme temperature conditions is also a potential way to induce faults, although it does not seem to be a frequent choice in nowadays attacks.

Radiations: Folklore often presents fault induction attacks as "microwave attacks" (the attacker puts the smart card into a microwave oven to have it perform erroneous computations). Although this is oversimplified, it is clear that correctly focused radiations can harm the device's behavior.

Light: Recently, Skorobogatov and Anderson [66] observed that illumination of a transistor causes it to conduct, thereby inducing a transient fault. By applying an intense light source (produced using a photoflash lamp magnified with a microscope), they were able to change individual bit values in an SRAM. By the same technique, they could also interfere with jump instructions, causing conditional branches to be taken wrongly.

Eddy current: Quisquater and Samyde [56] showed that eddy currents induced by the magnetic field produced by an alternating current in a coil could induce various effects inside a chip as for example inducing a fault in a memory cell, being RAM, EPROM, EEPROM or Flash (they could for example change the value of a pin code in a mobile phone card).

Several papers and books address the issue of fault induction techniques. We refer the reader to [7,5,6,29,30,46] and, for the last two techniques, to [66] and [56].

5.3 Cryptanalyses Based on Fault

Attack on RSA with CRT. Fault induction attack on RSA[2] with Chinese Remaindering Theorem (CRT) [12,37] is probably the most exemplary instance of fault induction attack: first, it is very easy to explain, even to a non-cryptologist; second, it is also easy to deploy, since only one fault induction *somewhere* in the computation – even with no precise knowledge of that fault's position – is enough to have it work; third, it is extremely powerful, as having one faulty computation performed is sufficient to completely break a signature device.

Implementations of RSA exponentiation often make use of the Chinese Remaindering Theorem to improve performance. Let m be the message to sign, $n = pq$ the secret modulus, d and e the secret and public exponents. Exponentiation process is described in Alg. 1.. Of course several values involved in this algorithm are constant and need not be recomputed every time.

[2] A short description of RSA can be found in Appendix A.

Algorithm 1. Chinese Remaindering Theorem

```
$m_p = m \bmod p$
$m_q = m \bmod q$
$d_p = d \bmod (p-1)$
$d_q = d \bmod (q-1)$
$x_p = m_p^{d_p} \bmod p$
$x_q = m_q^{d_q} \bmod q$
$s = \text{chinese}(x_p, x_q) = q(q^{-1} \bmod p)x_p + p(p^{-1} \bmod q)x_q \bmod n$
return s
```

Suppose an error occurs during the computation of either x_p or x_q (say x_p, to fix ideas, and denote by x'_p the incorrect result)[3]. It is easy to see that, with overwhelming probability, the faulty signature s' derived from x'_p and x_q will be such that

$$\begin{aligned} s'^e &\equiv m \bmod q, \\ s'^e &\not\equiv m \bmod p. \end{aligned}$$

Consequently, computing

$$\gcd(s'^e - m \bmod n, n)$$

will give the secret factor q with overwhelming probability. With this factorization of n, it is straightforward to recover the secret exponent d from e. As we see, having the cryptographic device perform one single faulty signature (without even the need to compare it to correct computations) is sufficient to be able to forge any number of signatures. Moreover, the type of fault is very general, and should therefore be fairly easy to induce.

Differential fault analysis. Shortly after the appearing of the fault attack against RSA, Biham and Shamir [9] showed that such attacks could be applied against block ciphers as well, by introducing the concept of *differential fault analysis* (DFA).

They demonstrated their attack against the Data Encryption Standard[4]. The fault model they assume is that of transient faults in registers, with some small probability of occurrence for each bit, so that during each encryption/decryption there appears a small number of faults (typically one) during the computation, and that each such fault inverts the value of one of the bits[5].

[3] Note that, since these two computations are by far the most complex part of the full signature process, inducing a transient fault at random time during the computation has great chance to actually affect one of these.

[4] A description of DES can be found in Appendix B.

[5] The authors claim that their model is the same as that of [12,37] but, in our opinion, this claim is misleading: whereas RSA's fault induction works provided *any* error occurs during the computation, DES's DFA requires that only one (or a very small number of) bit(s) is (are) affected by the error. This model is therefore much less general.

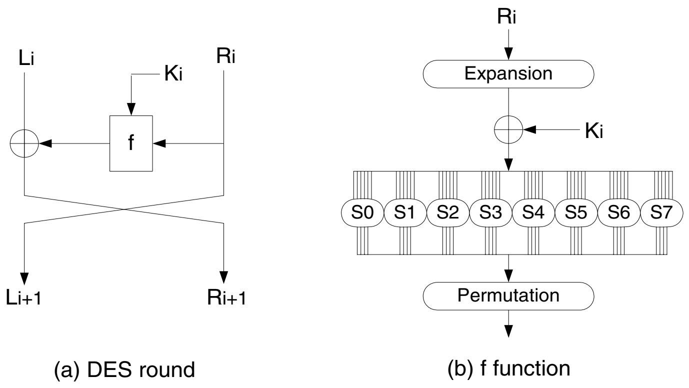


Fig. 3. Data Encryption Standard

The basic principle of their attack is that of a divide and conquer attack: suppose that we dispose of two results, one correct and one faulty, for the same input value. Suppose further that a (unique) fault has occurred, and that that fault affected one bit of R_{15} (that is, the right part of the input to the last round of DES – see Fig. 3(a), with $i = 15$). This bit will follow two paths through this last round. First, R_{15} will be copied into L_{16}, which will also have exactly one wrong bit. Second, it will be part of the input to one[6] S-box (Fig. 3(b)) and, being faulty, will induce a 4-bit error in this S-box output. This in turn will affect 4 bits of R_{16}. Noting that all further operations (IP^{-1}, P, $\ldots$) are deterministic and do not depend on unknown values, we can, from the final outputs (the right and the wrong) of DES, identify which bit of R_{15} was incorrect. We can also trace back the error as close as possible to the involved S-box. Taking the exclusive-or of the two outputs, we end up with a relationship of the form

$$S(R \oplus K) \oplus S(R \oplus F \oplus K) = X,$$

where S denotes the S-box under concern, R the part of interest of R_{15} (that we can reconstruct from the output), K the corresponding part of the key, and F the one-bit fault. Note that we do not know the S-box output, nor, of course, the key. The non-linearity of the S-box will help us: as a matter of fact, not all input pairs could have produced this output difference. So, we will simply perform an exhaustive search over all possible (that is, 2^6) key values, and discard all values which do not yield the expected difference. According to Biham and Shamir, only four possibilities remain on the average. Repeating the experiment, it is possible to confirm the correct 6-bit value, and then to attack other key parts.

To induce an error with the expected form and location, the attacker will repeat encryptions with device put under extreme conditions, and with same

[6] In fact, due to the expansion function, a single bit could affect two S-boxes. The argument is the same in this case, and we omit it for simplicity.

plaintext as input, until he observes a difference between ciphertexts with the expected pattern (one wrong bit in the output corresponding of L_{16} and four wrong bits in R_{16}). Choosing the time at which error is triggered to target the last rounds will of course be helpful.

Similar arguments can be used if the fault occurred in rounds 14 or 15. Using this technique, Biham and Shamir could recover a full DES key using between 50 and 200 messages. Note that triple-DES can be attacked in the same way.

It is important to remark at this point that fault induction significantly depends on the target device. While, due to the simplicity of the processor, it may be relatively easy to insert of fault during a specified computation in a smart card, large parallel designs (e.g. block ciphers implemented on FPGAs) may be more challenging.

Other results. Others fault models have also been considered, which allow pretty trivial attacks. Some authors, for example, consider a model in which memory cells can be flipped from one to zero (or from zero to one), but not the opposite. An obvious way to exploit this is to repeatedly induce faults on the key, until all its bits have been forced to zero (and producing some ciphertexts between each fault induction). The chain is then explored backwards, starting from the known (null) key, and guessing at each step which bits have been flipped; correct guesses are identified by comparison with the ciphertexts. An even simpler attack is that of [10], that additionally assumes that it is possible to choose the location of the flipped bit. In this case, the attack simply consists in forcing a key bit to zero and checking if the result is different from the one obtained without fault induction. If this is the case, conclude the key bit was 1, otherwise conclude 0.

Finally, several obvious ways to exploit very specific faults can easily be devised: for example, a fault that would affect a loop counter so that only two or three rounds of DES are executed would of course allow to break the scheme. Similarly, disabling the instruction decoder could have the effect that all instructions act as a NOP so the program counter cycles through the whole memory.

6 Timing Attack

6.1 Introduction

Usually the running time of a program is merely considered as a constraint, some parameter that must be reduced as much as possible by the programmer. More surprising is the fact that the running time of a cryptographic device can also constitute an *information channel*, providing the attacker with invaluable information on the secret parameters involved. This is the idea of *timing attack*. This idea was first introduced by Kocher [42].

In a timing attack, the information at the disposal of the attacker is a set of messages that have been processed by the cryptographic device and, for each of them, the corresponding running time. The attacker's goal is to recover the secret parameters (fig. 4). Remember that, as was said in section 2.1, the clock ticks

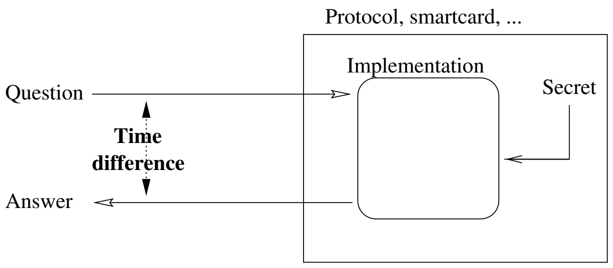


Fig. 4. The timing attack principle

are provided to the smart card by the terminal. Precise timing measurements are therefore easy to obtain.

6.2 Timing Attack Against RSA with Montgomery Reduction

The ideas suggested by Kocher were first practically applied by Dhem et al. against the RSA algorithm, implemented using Montgomery multiplication [25]. Although this attack is not optimal, its idea is pretty intuitive, and fits our tutorial purpose, so we will sketch it in this section.

The context is that of an RSA signature, and the goal of the attacker is to recover the secret exponent d. A common method to perform a modular exponentiation is the square and multiply algorithm (Alg. 2. [7]). This algorithm is mainly a sequence of modular multiplications and squares (which we will view as simple multiplications of a value by itself). When implemented in a scholar way, modular multiplications are time-consuming operations. Montgomery [52] proposed a clever way to speed-up these operations, by transferring them to a modulus which is better suited to the machine's internal structure.

Algorithm 2. Square and multiply

```
$x = m$
**for** $i = \omega - 2$ downto 0 **do**
  $x = x^2 \bmod n$
  **if** $d_i == 1$ **then**
    $x = x \cdot m \bmod n$
  **end if**
**end for**
return $x$
```

For simplicity, we will not describe Montgomery's algorithm in the detail here. For our purpose, it is sufficient to know that, for fixed modulus, the time for a Montgomery multiplication is constant, independently of the factors, except that, if the intermediary result of the multiplication is greater than the

[7] Here, $d_{\omega-1}$ denotes the most significant bit of d (which we assume to be equal to 1) and d_0 denotes the lsb.

modulus, an additional subtraction (called a reduction) has to be performed. In other words, this means that, depending on the input, the running time of a Montgomery multiplication can take two values, a "short one" (no final reduction) or a "long one" (final reduction needed). Of course, for given input values, we can predict whether the multiplication will be long or short.

The attack is an iterative divide and conquer attack: we will start by attacking the first unknown bit $d_{\omega-2}$, and, at each step of the attack, assume we know bits $d_{\omega-1} \dots d_{i+1}$ and recover bit d_i.

Let us begin by $d_{\omega-2}$. If this bit is equal to 1, the first loop pass in Alg 2. will involve a multiplication by m (line 5). As we have seen, this multiplication can be either long or short and, since no secret value is involved before this step, we can, for a given message m, determine whether this multiplication would be long or short. What we will do is partition our set of messages according to this criterion: all messages for which that first multiplication would be long will be put in subset A, and all messages for which that multiplication will be short will be put in subset B.

What is the influence of this step on the total running time? For all messages of subset A, the first pass in the loop has taken slightly more time than for all messages of subset B. What we expect is that this first pass will on average have a noticeable influence on the total running time, so that actual total running times for messages from subset A will be slightly longer. We cannot simulate further passes in the loop, since their behavior (and hence the evolution of x) depend on secret parameters $b_{\omega-3} \dots b_0$. So we will simply consider them as noise, hoping that their influence on messages from subset A will globally be the same than on subset B.

An important point in the attack is that the simulation and partition are based on the assumption that the first bit of the secret exponent is equal to 1. If this is not the case, then the conditional step we are targeting is not executed. Thus, our predictions of long or short operations will not correspond to any reality[8]. Our partition in sets A and B should then be roughly random, and there is no reason why we should observe any difference between the running times of subset A and B.

To conduct the attack, we will simply revert that argument: assuming $d_{\omega-2} = 1$, we build a partition as described above, and compare the average running times of the two subsets. If the running times are significantly different, we conclude that our assumption that $d_{\omega-2} = 1$ was true, otherwise, we conclude $d_{\omega-2} = 0$.

Once this bit has been determined, we have enough information to simulate the square and multiply until the second pass, and hence attack the next bit. Note that this does not require a new measurement set: we simply build a new partition of the same set.

[8] This argument is a bit too simplistic: in fact, the successive Montgomery multiplications of a square and multiply are not independent. Characterizing this fact allowed to greatly improve the attack's efficiency, but this falls outside the scope of this tutorial.

6.3 General Idea

The above argument can be generalized as follows. For a given algorithm involving a secret parameter, we view the global running time as a sum of random variables $T = \sum_{i=1}^{N} T_i$ corresponding to the various steps of the algorithm, and each individual execution of the algorithm as a realization of this random variable.

If we can – based on a key guess – simulate the computation and its associated running time up to step k, we can filter the measured running times by subtracting the parts corresponding to these steps. So, if input value m_j yielded a measured running time t_j and a simulated computation time t_j^{EST}, we estimate the remaining running time as $t_j^{REM} = t_j - t_j^{EST}$, corresponding to a realization of the random variable $T^k = \sum_{i=k+1}^{N} T_i$. If our key guess is correct, this filtering should reduce the variance (and the correct guess corresponds to the one that reduces variance the most).

We can generalize this further by characterizing the probabilities for an observation under the admissible hypotheses and the a priori distribution of these hypotheses, and deriving a decision strategy. The appropriate statistical tools for this purpose is maximum likelihood estimation or, better, optimal decision strategy. Schindler [62,63] applied an optimal decision strategy to the above scenario (square and multiply with Montgomery multiplication), and showed that this led to a drastic improvement in the attack's efficiency.

7 Power Analysis Attacks

In addition to its running time, the power consumption of a cryptographic device may provide much information about the operations that take place and the involved parameters. This is the idea of power analysis, first introduced by Kocher et al. in [43], that we describe in the context of a smart card implementation.

7.1 Measuring Phase

Measuring the power consumption of a smart card is a pretty easy task: as the clock ticks, the card's energy is also provided by the terminal, and can therefore easily be obtained. Basically, to measure a circuit's power consumption, a small (e.g., 50 ohm) resistor is inserted in series with the power or ground input. The voltage difference across the resistor divided by the resistance yields the current. Well-equipped electronics labs have equipment that can digitally sample voltage differences at extraordinarily high rates (over 1GHz) with excellent accuracy (less than 1% error). Devices capable of sampling at 20MHz or faster and transferring the data to a PC can be bought for less than US$ 400.

7.2 Simple Power Analysis

Simple Power Analysis (SPA) attempts to interpret the power consumption of a device and deduce information about the performed operations or involved

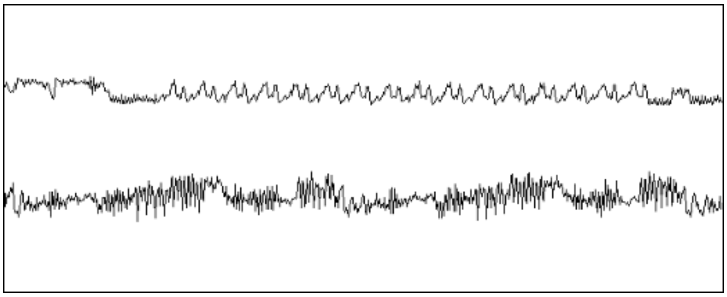


Fig. 5. SPA monitoring from a single DES operation performed by a typical smart card [43]. The upper trace shows the entire encryption operation, including the initial permutation, the 16 DES rounds, and the final permutation. The lower trace is a detailed view of the second and third rounds.

parameters. This is better illustrated by an example. Fig. 5, taken from the original description of power analysis, shows the consumption curve (named a *trace*) of a device performing a DES operation. It clearly shows a pattern that is repeated 16 times and corresponds to the 16 rounds of DES. The lower part of the figure is a detailed view of two rounds, providing more information about the round's substeps.

Of course, this information is not an attack in itself. Everybody knows that DES has 16 rounds, and knowing that a device is performing a DES encryption does not expose its secrets at all. According to our Kerckhoffs assumption, the cryptographic algorithm is known to the attacker anyway. However, there are cases in which this sequence of operations can provide useful information, mainly when the instruction flow depends on the data. For example, let us come back to exponentiation through the square and multiply algorithm. If the square operation is implemented differently than the multiply – a tempting choice, as this will allow specific optimizations for the square operation, resulting in faster code – and provided this difference results in different consumption patterns, then the power trace of an exponentiator directly yields the exponent's value. Similarly, some hardware multipliers behave differently when one of their operands is zero, which allows immediate return of the result, but exposes this input value. Generally speaking, all programs involving conditional branch operations depending on secret parameters are at risk.

However, power consumption may also depend on the *data* manipulated. This leads to a more general class of attacks that is investigated in the next section. So, SPA typically targets variable instruction flow, whereas Differential Power Analysis and its variants target data-dependence.

In practice, instruction flow exposing is a point where the security of smart cards and FPGAs may differ. On the one hand, sequential computing in smart cards involves that at one specific instant, only one of the operations is running, e.g. either square or multiply in our previous example. This makes it possible to distinguish operations by a simple visual inspection of the power traces. On the other hand, in FPGAs, parallel computation (if used in the design) prevents this visual inspection, as the power consumption will not only be the one of the targeted operation.

7.3 Differential Power Analysis

As we said, the idea of a divide and conquer attack is to compare some characteristic to the values being manipulated. One of the simplest comparison tools we can imagine is mean comparison, used in the original power analysis attack: Differential Power Analysis (DPA) [43]. As it is pretty intuitive, we will begin by studying it, in the context of DES encryption.

Here, the characteristic we will focus on is an arbitrary output bit b of an arbitrary S-box at the 16th round (say the first bit of S_1's output), and we will observe this characteristic through its associated power consumption. In fact, we have no idea of when this value is actually computed by the cryptographic device, nor do we know what its associated power consumption may look like. The only thing we know for sure is that, at some point in time, that value will be manipulated (computed, stored, ...) by the cryptographic device and that the power consumption of the device depends on this data. To expose that value, we will use a method very similar to the one we used for timing attack: partitioning and averaging.

Let us first summarize the idea. We will perform a large number of encryptions with variable inputs and record the corresponding consumption curves. Then, we will partition our set in two subsets: one, A, for which the target bit was equal to 0, the other, B for which the target bit was 1. In order to do this partitioning, we need to be able to estimate the value of bit b. We have access to the value R_{15} that entered the last round function, since it is equal to L_{16} (Fig. 3(a)). Looking at the round function in more detail (Fig. 3(b)), we see that six specific[9] bits of R_{15} will be XORed with six specific bits of the key, before entering the S-box. So, doing a guess on a 6-bit key value, we are able to predict the value of b. This means that we actually have 2^6 partitions of our measurements, one for each possible key.

We will then compute the average power consumption for messages in subset A, and subtract it from the average power consumption for messages in subset B (for each possible partition). Since all computations for subset A involved manipulating a 0 at a point where all computations for subset B involved manipulating a 1, we should observe notable differences at points in time where this value was actually manipulated, whereas we expect other points, which are unrelated to b, to behave as noise and be cancelled in subtraction. Of course, this will only be true provided our initial key guess is correct, and we will in fact use our observation as an oracle to validate this key guess: only the correct key guess (corresponding to the correct partition) should lead to significant differences.

[9] By specific we mean that they are chosen according to fixed permutation functions, so we can identify them independently of any key value.

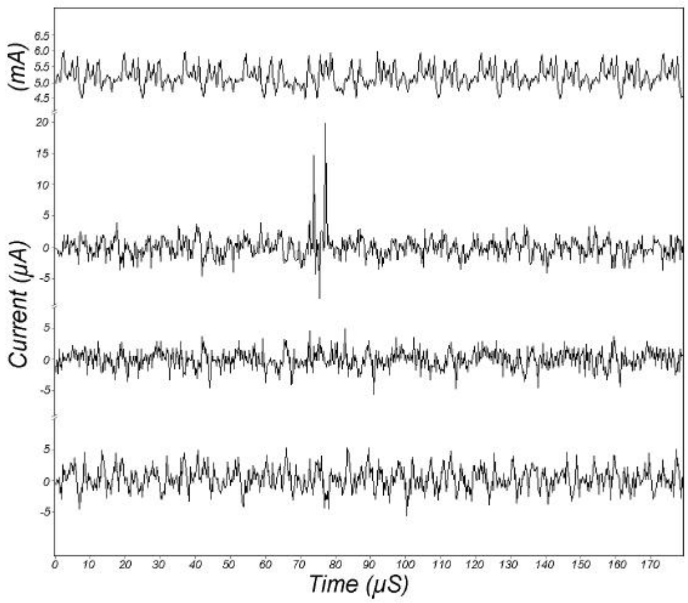


Fig. 6. DPA traces from [43]

Figure 6, taken from the original paper describing DPA, shows traces corresponding to correct and incorrect key guesses: the top trace is the average power consumption among the whole sample; the curve below shows the differential trace corresponding to the right key guess (and hence right partition), and the two bottom curves show differential traces for wrong key guesses. The spikes are clearly visible.

According to Messerges, such a difference of mean test is relevant because, at some point during a software DES implementation, the microprocessor needs to compute b. When this occurs or any time data containing this selection bit is manipulated, there will be a slight difference in the amount of power dissipated, depending on the values of these bits. What we just did is in fact building a *model* of the smart card behavior. The model is, in this case, pretty elementary, but this modelling phase is nonetheless an indispensable part of the attack.

Finally, once this 6-bit subkey has been determined, the attack goes on by focusing on other S-boxes and target bits.

7.4 Correlation Attack

Looking at the previous descriptions, it is easy to see that a DPA is far from making an optimal use of the sampled measurements. In this section, we describe two possible improvements.

A first improvement, usually denoted as the "multiple bit" DPA, comes when observing that the key guess performed in a DPA does not only allow to predict the bit b, but all the four output bits of S_1. As a consequence, one may separate the measurements in two sets according to these multiple bit values: one set corresponding to "*all zeroes*", the other one to "*all ones*". This improves the attack signal to noise ratio. However, as multiple bit attacks only consider the texts that give rise to "*all something*" values, it is suboptimal and a lot of texts (measurements) are actually not used. The correlation analysis allows solving this problem and constitutes a second improvement. Correlation Power Analysis usually hold in three steps.

– First, the attacker *predicts* the power consumption of the running device, at one specific instant, in function of certain secret key bits. For example, let us assume that the power consumption of the DES implementation depends of the Hamming weight of the data processed[10]. Then, the attacker could easily predict the value of S_1's output Hamming weight, for the 2^6 possible values of the key guess and N different input texts. This gives 2^6 possible predictions of the device power consumption and the result of this prediction is stored in a **prediction matrix**.
– Secondly, the attacker *measures* the power consumption of the running device, at the specific time where it processes the same input texts as during the prediction phase. The result of this measurement is stored in a **consumption vector**.
– Finally, the attacker *compares* the different predictions with the real, measured power consumption, using the correlation coefficient[11]. That is, he computes the correlation between the consumption vector and all the columns of the prediction matrix (corresponding to all the 2^6 key guesses). If the attack is successful, it is expected that only one value, corresponding to the correct key guess, leads to a high correlation coefficient.

Such an attack has been successfully applied to a variety of algorithms and implementations, e.g. in [13,55,67]. As an illustration, Figure 7 shows the result of a correlation attack against an implementation of the DES [68]. It is clearly observed that the correct key is distinguishable after 500 measurements.

To conclude this section, a number of points are important to notice:

– Correlation power analysis (as any power analysis attack targeting data dependencies) requires a power consumption model. The quality of this model

[10] As already mentioned, this is a typical model for the power consumed in smart cards.
[11] Let $M(i)$ denote the ith measurement data (*i.e.* the ith trace) and M the set of traces. Let $P(i)$ denote the prediction of the model for the *ith* trace and P the set of such predictions. Then we calculate:

$$C(M,P) = \frac{\mu(M \times P) - \mu(M) \times \mu(P)}{\sqrt{\sigma^2(M) \times \sigma^2(P)}} \tag{1}$$

where $\mu(M)$ denotes the mean of the set of traces M and $\sigma^2(M)$ its variance.

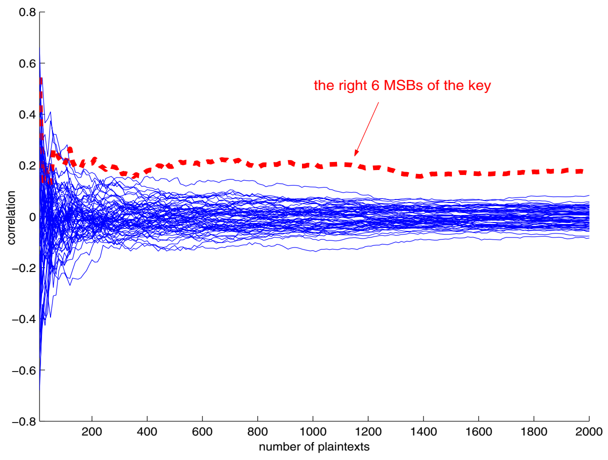


Fig. 7. A correlation attack against the Data Encryption Standard

has a strong influence on the attack efficiency. However, this leakage model does not mandatorily have to be known a priori by the attacker and can be built from measurements. This is the context of template attacks.

– Next to the leakage model, the synchronization of measurements is another typical issue in side-channel attacks. Fundamentally, the easiest synchronization is to perform the statistical test on the complete sampled data (this is typically done in DPA). However, when the statistical test becomes computationally intensive, such a technique may become cumbersome to deal with in practice. Some knowledge about the design or more advanced synchronization techniques (e.g. FFT-based) usually solve the problem.
– As SPA, DPA-like attacks are affected by parallel computing. This is easily seen when observing the DES design in Appendix B. Say we observe an 8-bit smart card implementation. When S_1's output is computed, the sampled power consumption relates to the four predicted S-box output bits and 4 other (unknown) bits. These unknown bits basically add some algorithmic noise to the measurements. Say we observe an FPGA implementation with all the S-boxes computed in parallel. Then, only 4 bits are predicted out of the 32 actually computed in the device. This means that the power consumption will be affected by more algorithmic noise. In general, as already mentioned in the context of fault insertion attacks, parallel computing improves security against certain physical attacks.
– Remark finally that the described correlation power analysis is still not optimal. Techniques based on a maximum likelihood detection of the correct key offer even better results. However, as the power consumption models in use (i.e. based on the Hamming weight, distance of the data processed) have a linear dependence, correlation provides a simple and efficient tool for attacking devices in practice.

8 EMA

Any movement of electric charges is accompanied by an electromagnetic field. *Electromagnetic attacks*, first introduced by Quisquater and Samyde [58], and further developed in [59,26] exploit this side channel by placing coils in the neighborhood of the chip and studying the measured electromagnetic field.

The information measured can be analyzed in the same way as power consumption (simple and differential electromagnetic analysis – SEMA and DEMA – or more advanced correlation attacks), but may also provide much more information and are therefore very useful, even when power consumption is available[12]. As a matter of fact, 3D positioning of coils might allow to obtain much more information from the device's components. Moreover, Agrawal et al. [2] showed that EM emanations consist of a multiplicity of signals, each leaking somewhat different information about the underlying computation. They sort the EM emanations in two main categories: direct emanations, i.e. emanations that result from intentional current flow, and unintentional emanations, caused by coupling effects between components in close proximity. According to them, unintentional emanations can prove much more useful that direct emanations. Moreover, some of them have substantially better propagation than direct emanations, which enables them to be observed without resorting to invasive attacks (and even, in some cases, to be carried out at pretty large distances - 15 feet! - which brings it to the field of tempest-like attacks [1]).

To summarize, EMA measurement phase is much more flexible and challenging that power measurement phase, and the provided information offers a wide spectrum of potential information. On the other hand, this information may be exploited using the same basic or advanced techniques as for power analysis, even if optimal decision models can be adapted to the specificities of EMA.

In essence, EMA is a non-invasive attack, as it consists in measuring the near field. However, this attack is made much more efficient by depackaging the chip first, to allow nearer measurements and to avoid perturbations due to the passivation layer.

9 Countermeasures

Countermeasures against physical attacks range among a large variety of solutions. However, in the present state of the art, no single technique allows to provide perfect security, even considering a particular attack only. Protecting implementations against physical attacks consequently intends to make the attacks harder. In this context, the implementation cost of a countermeasure is of primary importance and must be evaluated with respect to the additional security obtained. The exhaustive list of all possible solutions to protect cryptographic implementations from physical opponents would deserve a long survey

[12] One can of course imagine contexts in which power consumption cannot be obtained, but where it is possible to measure the radiated field, for example from a short distance.

in itself. In this section, we will only suggest a few exemplary proposals in order to illustrate that security can be added at different levels. We refer the reader to [57] for a more comprehensive (although slightly outdated) review of existing countermeasures.

The most direct way to prevent physical opponents is obviously to act at the physical/hardware level. A typical example related to probing is the addition of shields, conforming glues, or any process that makes the physical tempering of a device more complex. Detectors that react to any abnormal circuit behaviors (light detectors, supply voltage detectors, ...) may also be used to prevent probing and fault attacks [69]. With respect to side-channel attacks, simple examples of hardware protection are noise addition, or the use of detachable power supplies [65]. However, as detailed in [22,47], none of these methods provide a perfect solution.

Close to the physical level, technological solutions can also be considered. There exist circuit technologies that offer inherently better resistance against fault attacks (e.g. dual rail logic [72]) or side-channel attacks (e.g. any dynamic and differential logic style [45,70]). While these solutions may be very interesting in practice, because they can be combined with good performance, they do not offer any theoretical guarantee of security either. For example, dynamic and differential logic styles usually make the modelling of the power consumption more difficult (no simple prediction, smaller data dependencies of the leakage) but theoretically, such a model can always be obtained by the mean of template attacks (see next section).

Finally, most of the proposed techniques aim to counteract fault and side-channel attacks at the algorithmic level. Certain solutions, such as the randomization of the clock cycles, use of random process interrupts [48] or bus and memory encryption [14,27], may be used to increase the difficulty of successfully attacking a device, whatever physical attack is concerned. Most solutions however relate to one particular attack. With respect to faults, they mainly include different kinds of error detection/correction methods based on the addition of space or time redundancies [64,41,35,40]. The issue regarding these techniques is that their cost is frequently close to naive solutions such as duplication or repetition. Regarding side-channel attacks, a lot of countermeasures intend to hide (or mask) the leakage or to make it unpredictable. One could for example think about performing a "square and multiply always" algorithm (i.e. always perform a multiplication and discard the result if the operation was not necessary) for exponentiation, and using a similar "reduce always" method for Montgomery multiplication in order to counter timing attacks[13]. Another countermeasure consists in preventing the attacker from being able to make predictions about intermediary values, a simple example of which is Kocher's blinding [42]. Other typical examples are in [4,19,28]. Once again, such protections are not perfect and, in the case of block ciphers, for example, have been shown to be susceptible

[13] These are of course naive methods, which have been widely improved in the literature.

to higher-order side-channel attacks. Moreover, their implementation cost is also an issue.

In general, good security may nevertheless be obtained by a sound combination of these countermeasures. However, it is more an engineering process than a scientific derivation of provably secure schemes. Theoretical security against physical attacks is a large scope for further research.

10 More Advanced Attacks and Further Readings

Most advanced scenarios generally aim to improve the attacks' efficiency (i.e. to reduce the number of physical interactions required), to make them more general or to defeat certain countermeasures. We give here typical examples of such improvements that are also good directions for further readings.

For power and Electromagnetic analysis: (1) Multi-channel attacks [3] basically aim to improve the attacks' efficiency by making an optimal use of different available leakages, based on a maximum likelihood approach. (2) Template attacks [20] intend to remove the need for a leakage model and consequently make power and electromagnetic analysis more general. For this purpose, they act in two separate steps: (a) build a model based on sampled data, using a template of the target device (b) use maximum likelihood to determine the key whose leakage profile matches the best the one of the device under attack. Remark that templates inherently allow to defeat a large number of countermeasures. (3) Second-order attacks [51,71] finally target implementations in which the leakage has been masked. Contrary to first order attacks that aim to predict the power consumption of one specific operation at one specific instant during a cryptographic computations, higher-order techniques take advantage of correlations existing between multiple operations, possibly performed at different instants.

Another research lead is to explore the feasibility of an attack in a different context. For example, some authors demonstrated that timing attacks can also be conducted against a classical desktop computer, or even against a server accessed through a network connection [15,8]. Moreover, Bernstein argues that the complexity of a general-purpose CPU (multiple cache levels, pipelining, ...) might make timing attacks extremely difficult to prevent on such platforms.

Elliptic curve cryptography has been the field of a large number of publications regarding physical attacks. As elliptic curve operations are basically transpositions of classical operations in another mathematical structure, many of the aforementioned attacks can be transposed to this context too. On the other hand, this structure offers a degree of flexibility that allows several specific countermeasures to be designed in an efficient way (see [36,34,33,24,21], to only name a few).

11 Conclusion

In this paper, we presented a number of practical concerns for the security of cryptographic devices, from an intuitive point of view. It led us to cover recently proposed attacks and discuss their respective characteristics as well as possible countermeasures. The discussion clearly underlines that probing, side-channel or fault attacks constitute a very significant threat for actual designers. The generality of these attacks was suggested in the tutorial, as we mainly investigated algorithmic issues. However, it must be noted that the actual implementation of physical attacks (and more specifically, their efficiency) may be significantly platform-dependent. Moreover, the presented techniques usually require a certain level of practical knowledge, somewhat hidden in our abstract descriptions.

From an operational point of view, the present state of the art suggests that good actual security may be obtained by the sound combination of countermeasures. However, significant progresses are still required both in the practical setups for physical interactions with actual devices and in the theoretical understanding of the underlying phenomenons. The development of a theory of provable security is therefore a long term goal.

Acknowledgements

The authors are grateful to W. Schindler for useful comments.

References

1. *NSA tempest series*, Available at `http://cryptome.org/#NSA--TS`.
2. D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi, *The EM side channel*, in Kaliski et al. [38].
3. D. Agrawal, J. R. Rao, and P. Rohatgi, *Multi-channel attacks*, Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES) (Cologne, Germany) (C. Walter, Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, September 7-10 2003, pp. 2–16.
4. M.-L. Akkar and C. Giraud, *An implementation of DES and AES, secure against some attacks*, in Çetin K. Koç et al. [16].
5. R. Anderson and M. Kuhn, *Tamper resistance – a cautionary note*, Proc. of the second USENIX workshop on electronic commerce (Oakland, California), Nov. 18-21 1996, pp. 1–11.
6. ______, *Low cost attacks on tamper resistant devices*, Proc. of 1997 Security Protocols Workshop, Lectures Notes in Computer Science (LNCS), vol. 1361, Springer, 1997, pp. 125–136.
7. R.J. Anderson, *Security engineering*, Wiley & Sons, New York, 2001.
8. Daniel J. Bernstein, *Cache-timing attacks on AES*, Available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, November 2004.
9. E. Biham and A. Shamir, *Differential fault analysis of secret key cryptosystems*, Proc. of Advances in Cryptology – Crypto '97 (Berlin) (Burt Kaliski, ed.), vol. 1294, Springer-Verlag, 1997, Lecture Notes in Computer Science Volume 1294, pp. 513–525.
10. J. Blömer and J.P. Seifert, *Fault based cryptanalysis of the advanced encryption standard (AES)*, Cryptology ePrint Archive: Report 2002/075. Available at `http://eprint.iacr.org`.

11. D. Boneh (ed.), *Advances in cryptology - CRYPTO '03*, Lectures Notes in Computer Science (LNCS), vol. 2729, Springer-Verlag, 2003.
12. D. Boneh, R.A. DeMillo, and R.J. Lipton, *On the importance of checking cryptographic protocols for faults*, Advances in Cryptology - EUROCRYPT '97, Konstanz, Germany (W. Fumy, ed.), LNCS, vol. 1233, Springer, 1997, pp. 37–51.
13. E. Brier, C. Clavier, and F. Olivier, *Correlation power analysis with a leakage model*, proceedings of CHES, Lectures Notes in Computer Science (LNCS), vol. 3156, Springer, 2004, pp. 16–29.
14. E. Brier, H. Handschuh, and C. Tymen, *Fast primitives for internal data scrambling in tamper resistant hardware*, in Çetin K. Koç et al. [16], pp. 16–27.
15. B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, *Password interception in a SSL/TLS channel*, in Boneh [11].
16. Çetin K. Koç, David Naccache, and Christof Paar (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2001*, Lectures Notes in Computer Science (LNCS), vol. 2162, Springer-Verlag, August 2001.
17. Çetin K. Koç and Christof Paar (eds.), *Cryptographic Hardware and Embedded Systems - CHES '99*, Lectures Notes in Computer Science (LNCS), vol. 1717, Springer-Verlag, August 1999.
18. Çetin K. Koç and Christof Paar (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2000*, Lectures Notes in Computer Science (LNCS), vol. 1965, Springer-Verlag, August 2000.
19. S. Chari, C. Jutla, J. Rao, and P. Rohatgi, *Towards sound approaches to counteract power-analysis attacks*, Advances in Cryptology - CRYPTO '99 (M. Wiener, ed.), Lectures Notes in Computer Science (LNCS), vol. 1666, Springer-Verlag, 1999.
20. S. Chari, J.R. Rao, and P. Rohatgi, *Template attacks*, in Kaliski et al. [38].
21. Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye, *Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity*, IEEE Transactions on Computers **53** (2004), no. 6.
22. C. Clavier, J.S. Coron, and N. Dabbous, *Differential power analysis in the presence of hardware countermeasures*, in Kaliski et al. [39].
23. K. Compton and S. Hauck, *Reconfigurable computing: A survey of systems and software*, ACM Computing Surveys **34** (2002), no. 2.
24. J.-S. Coron, *Resistance against differential power analysis for elliptic curves cryptosystems*, in Çetin K. Koç and Paar [17].
25. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, *A practical implementation of the timing attack*, Proc. CARDIS 1998, Smart Card Research and Advanced Applications (J.-J. Quisquater and B. Schneier, eds.), LNCS, Springer, 1998.
26. K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic analysis: Concrete results*, Proc. of Cryptographic Hardware and Embedded Systems (CHES 2001) (Çetin Kaya Koç, David Naccache, and Christof Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer, 2001, pp. 251–261.
27. Jovan D. Golic, *DeKaRT: A new paradigm for key-dependent reversible circuits*, in Kaliski et al. [39], pp. 98–112.
28. L. Goubin and J. Patarin, *DES and differential power analysis: the duplication method*, in Çetin K. Koç and Paar [17].
29. P. Gutmann, *Secure deletion of data from magnetic and solid-state memory*, Proc. of 6th USENIX Security Symposium, 1997, pp. 77–89.
30. ———, *Data remanence in semiconductor devices*, Proc. of 7th USENIX Security Symposium, 1998.

31. Helena Handschuh, Pascal Paillier, and Jacques Stern, *Probing attacks on tamper-resistant devices*, in Çetin K. Koç and Paar [17].
32. Yuval Ishai, Amit Sahai, and David Wagner, *Private circuits: Securing hardware against probing attacks*, in Boneh [11].
33. K. Itoh, J. Yajima, M. Takenaka, and N. Torii, *DPA countermeasures by improving the window method*, in Kaliski et al. [38].
34. T. Izu and T. Takagi, *Fast parallel elliptic curve multiplications resistant to side channel attacks*, Proc. of PKC '2002 (David Naccache and Pascal Paillier, eds.), Lecture Notes in Computer Science, vol. 2274, Springer, 2002, pp. 335–345.
35. N. Joshi, K. Wu, and R. Karry, *Concurrent error detection schemes for involution ciphers*, in Çetin K. Koç and Paar [18], pp. 400–412.
36. M. Joye and J.-J. Quisquater, *Hessian elliptic curves and side-channel attacks*, in Çetin K. Koç et al. [16].
37. Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater, *Chinese remaindering based cryptosystems in the presence of faults*, Journal of cryptology **12** (1999), no. 4, 241–245.
38. Burton S. Kaliski, Çetin K. Koç, and Christof Paar (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2002*, Lectures Notes in Computer Science (LNCS), vol. 2523, Springer-Verlag, August 2002.
39. Burton S. Kaliski, Çetin K. Koç, and Christof Paar (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2003*, Lectures Notes in Computer Science (LNCS), vol. 2779, Springer-Verlag, September 2003.
40. M. Karpovsky, K.J. Kulikowski, and A. Taubin, *Differential fault analysis attack resistant architectures for the advanced encryption standard*, proceedings of CARDIS 2004.
41. R. Karri, G. Kuznetsov, and M. Gössel, *Parity-based concurrent error detection of substitution-permutation network block ciphers*, in Kaliski et al. [39].
42. P. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology - CRYPTO '96, Santa Barbara, California (N. Koblitz, ed.), LNCS, vol. 1109, Springer, 1996, pp. 104–113.
43. P. Kocher, Jaffe J., and B. Jub, *Differential power analysis*, Proc. of Advances in Cryptology – CRYPTO '99 (M. Wiener, ed.), LNCS, vol. 1666, Springer-Verlag, 1999, pp. 388–397.
44. Olivier Kömmerling and Markus G. Kuhn, *Design principles for tamper-resistant smartcard processors*, Proc. of USENIX Workshop on Smartcard Technology (Smartcard '99), 1999.
45. F. Mace, F.-X. Standaert, I. Hassoune, J.-D. Legat, and J.-J. Quisquater, *A dynamic current mode logic to counteract power analysis attacks*, proceedings of DCIS, 2004.
46. D.P. Maher, *Fault induction attacks, tamper resistance, and hostile reverse engineering in perspective*, Financial Cryptography: First International Conference (FC '97) (R. Hirschfeld, ed.), Lectures Notes in Computer Science (LNCS), vol. 1318, Springer-Verlag, 1997.
47. S. Mangard, *Hardware countermeasures against DPA - a statistical analysis of their effectiveness*, proceedings of CT-RSA, Lecture Notes in Computer Science, vol. 2964, Springer-Verlag, 2004, pp. 222–235.
48. D. May, H. Muller, and N. Smart, *Randomized register renaming to foil DPA*, in Çetin K. Koç et al. [16].
49. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.

50. T. S. Messerges, E. A. Dabbish, and R. H. Sloan, *Investigations of power analysis attacks on smartcards*, Proc. USENIX Workshop on Smartcard Technology, 1999.
51. Th.S. Messerges, *Using second-order power analysis to attack DPA resistant software*, in Çetin K. Koç and Paar [18].
52. P.L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), no. 170, 519–521.
53. National Bureau of Standards, *FIPS 197, Advanced Encryption Standard*, Federal Information Processing Standard, NIST, U.S. Dept. of Commerce,November 2001.
54. ———, *FIPS PUB 46, The Data Encryption Standard*, Federal Information Processing Standard, NIST, U.S. Dept. of Commerce, Jan 1977.
55. S.B. Ors, F. Gurkaynak, E. Oswald, and B. Preneel, *Power-analysis attack on an asic aes implementation*, proceedings of ITCC, 2004.
56. J.-J. Quisquater and D. Samyde, *Eddy current for magnetic analysis with active sensor*, Proc. of Esmart 2002, 2002.
57. Jean-Jacques Quisquater and François Koeune, *Side-channel attacks: state-of-the-art*, CRYPTREC project deliverable, available at `http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1047_Side_Channel_report.pdf`, October 2002.
58. Jean-Jacques Quisquater and David Samyde, *A new tool for non-intrusive analysis of smart cards based on electro-magnetic emissions: the SEMA and DEMA methods*, Eurocrypt rump session, 2000.
59. ———, *Electromagnetic analysis (EMA): measures and countermeasures for smart cards*, Smart cards programming and security (e-Smart 2001), Lectures Notes in Computer Science (LNCS), vol. 2140, Springer, 2001, pp. 200–210.
60. W. Rankl and W. Effing, *Smart card handbook*, John Wiley & Sons, 1997.
61. R. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, **21** (1978).
62. W. Schindler, *Optimized timing attacks against public key cryptosystems*, Statistics & Decisions (2000), to appear.
63. W. Schindler, J.-J. Quisquater, and F. Koeune, *Improving divide and conquer attacks against cryptosystems by better error detection correction strategies*, Proc. of 8th IMA International Conference on Cryptography and Coding (Berlin) (B. Honary, ed.), Springer, December 2001, Lecture Notes in Computer Science Volume 2260, pp. 245–267.
64. A. Shamir, *How to check modular exponentiation*, Presented at the rump session of EUROCRYPT '97, Konstanz, Germany.
65. A. Shamir, *Protecting smart cards from passive power analysis with detached power supplies*, in Çetin K. Koç and Paar [18].
66. S. Skorobogatov and R. Anderson, *Optical fault induction attacks*, in Kaliski et al. [38].
67. F.-X. Standaert, S.B. Ors, and B. Preneel, *Power analysis of an fpga implementation of rijndael: is pipelining a dpa countermeasure?*, proceedings of CHES, Lectures Notes in Computer Science (LNCS), vol. 3156, Springer, 2004, pp. 30–44.
68. F.-X. Standaert, S.B. Ors, J.-J. Quisquater, and B. Preneel, *Power analysis attacks against FPGA implementations of the DES*, proceedings of FPL, Lecture Notes in Computer Science, vol. 3203, Springer-Verlag, 2004, pp. 84–94.
69. H. Bar-El *et al.*, *The sorcerer's apprentice guide to fault attacks*, Tech. Report 2004/100, IACR eprint archive, 2004, Available at http://eprint.iacr.org.
70. K. Tiri, M. Akmal, and I. Verbauwhede, *A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards*, proceedings of ESSCIRC, 2003.

71. J. Waddle and D. Wagner, *Towards efficient second-order power analysis*, proceedings of CHES, Lecture Notes in Computer Science, vol. 3156, Springer-Verlag, 2004, pp. 1–15.
72. J.D. Waddle and D.A. Wagner, *Fault attacks on dual-rail encoded systems*, Tech report UCB//CSD-04-1347, UC Berkeley, August 23, 2004.

A RSA

RSA, named after the initials of its authors, Rivest, Shamir and Adleman [61] is probably the most famous asymmetric encryption (and signature) primitive. It basically goes as follows:

1. Alice chooses two large prime numbers p and q and computes their product $n = pq$ and $\phi(n) = (p-1)(q-1)$.
2. She also chooses a value e that has no common factor with $\phi(n)$ and computes $d = e^{-1} \bmod \phi(n)$.
3. Alice publishes (n, e) as her public key, and keeps d as her private key.
4. To send her a message m (with $0 \leq m < n$), Bob computes $c = m^e \bmod n$.
5. Alice decrypts c by computing $c^d \bmod n$. By Euler's theorem, it can easily be shown that the result is equal to m.

We refer the reader to [49] for more information on RSA.

B The Data Encryption Standard : A Case Study

In 1977, the DES algorithm [54] was adopted as a Federal Information Processing Standard (FIPS) for unclassified government communication. Although a new Advanced Encryption Standard was selected in October 2000 [53], DES is still widely used, particularly in the financial sector. DES encrypts 64-bit blocks with a 56-bit key and processes data with permutations, substitutions and XOR operations. It is a good example of Feistel cipher and its structure allows very efficient hardware implementations.

Basically, the plaintext is first permuted by a fixed permutation *IP*. Next the result is split into two 32-bit halves, denoted with L (left) and R (right) to which a round function is applied 16 times. The ciphertext is calculated by applying the inverse of the initial permutation *IP* to the result of the 16th round.

The secret key is expanded by the key schedule algorithm to sixteen 48-bit round keys K_i and in each round, a 48-bit round key is XORed to the text. The key schedule consists of known bit permutations and shift operations. As a consequence, finding any round key bit directly involves that the secret key is corrupted.

The round function is represented in Figure 8 (a) and is easily described by:

$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus f(R_i, K_i)$$

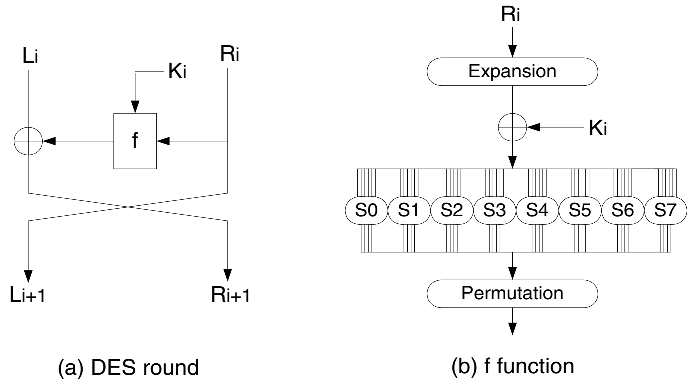


Fig. 8. Data Encryption Standard

where f is a nonlinear function detailed in Figure 8 (b): the R_i part is first expanded to 48 bits with the E box, by doubling some R_i bits. Then, it performs a bitwise modulo 2 sum of the expanded R_i part and the 48-bit round key K_i. The output of the XOR function is sent to eight non-linear S-boxes. Each of them has six input bits and four output bits. The resulting 32 bits are permuted by the bit permutation P.

Finally, DES decryption consists of the encryption algorithm with the same round keys but in reversed order.

Static Analysis of Authentication[*]

Riccardo Focardi

Dipartimento di Informatica,
Università Ca'Foscari di Venezia
focardi@dsi.unive.it

Abstract. Authentication protocols are very simple distributed algorithms whose purpose is to enable two entities to achieve mutual and reliable agreement on some piece of information, typically the identity of the other party, its presence, the origin of a message, its intended destination. Achieving the intended agreement guarantees is subtle because they typically are the result of the encryption/decryption of messages composed of different parts, with each part providing a "piece" of the authentication guarantee. This tutorial paper presents the basics of authentication protocols and illustrates a specific technique for statically analysing protocol specifications. The technique allows us to validate protocols in the presence of both malicious outsiders and compromised insiders, with no limitation on the number of parallel sessions.

This paper covers the course "Static Analysis of Authentication" given by the author at the FOSAD'04 school. The static analysis technique described here is a joint work with Michele Bugliesi and Matteo Maffei (Università di Venezia) [8,12].

1 Introduction

Security protocols are designed to provide diverse security guarantees in possibly hostile environments: typical guarantees include the secrecy of a message exchange between two trusted entities, the freshness and authenticity of a message, the authenticity of a claimed identity, ... and more. The presence of hostile entities makes protocol design complex and often error prone, as shown by many attacks to long standing protocols reported in the literature (see, e.g., [13,14,22,27,28]). In most cases, such attacks dwell on flaws in the protocols' logic, rather than on breaches in the underlying cryptosystem. Indeed, even when cryptography is assumed as a fully reliable building-block, an intruder can engage a number of potentially dangerous actions, notably, intercepting/replaying/forging messages, to break the intended protocol invariants. Formal methods have proved very successful as tools for protocol design and validations. On the one hand, failures to model-check protocols against formal specifications have lead to the discovery of several attacks (see, e.g.,

[*] Work partially supported by EU Contract IST-2001-32617 'MyThS' and by MIUR Project 'Abstract Interpretation: Design and Applications' (AIDA).

[26,28]). On the other hand, static techniques, based on type systems and control-flow analyses have proved effective in providing static guarantees of correctness [1,2,5,6,10,16,17].

Informally, *authentication* protocols enable two entities to achieve mutual and reliable agreement on some piece of information, typically the identity of the other party, its presence, the origin of a message, its intended destination. Even if there are many formalizations of authentication and many formal tools to verify it, not much has been done on the side of static analysis. Static analysis techniques like, e.g., type systems and control flow, aim at proving a property of a program, or a protocol, by only inspecting the source code. They are appealing because (*i*) only the code is inspected and there is no need of generating and exploring all the possible execution sequences; (*ii*) by exploiting compositionality, it becomes feasible to (even automatically) prove correctness of protocols with unbounded number of sessions and participants. These advantages are payed by approximating the property of interest thus loosing precision: in fact, static analyses are typically sound, i.e., an incorrect protocol is never validated, but not complete. As a consequence, there might be false-negatives, i.e., correct protocols that cannot be validated.

This tutorial paper introduces the basics of authentication protocols in a smooth and incremental way (Section 2), discusses how authentication is formalized in terms of correspondence assertions (Section 3), illustrates a type-based technique for statically analysing protocol specifications (Section 4) and briefly discusses related work (Section 5). This paper covers the course "Static Analysis of Authentication" given by the author at the FOSAD'04 school. The static analysis technique described in Section 4 is a joint work with Michele Bugliesi and Matteo Maffei (Università di Venezia). Here, we only focus on the basic ideas and concepts of such analysis and more detail can be found in [8,12].

2 Authentication Protocols

As mentioned above, *authentication* protocols enable two entities to achieve mutual and reliable agreement on some piece of information, typically the identity of the other party, its presence, the origin of a message, its intended destination. Authentication is thus related to the crucial problem of reliably agreeing on information that is spread on a (possibly untrusted) network like, e.g., the Internet. In this section, we present the basic techniques to achieve this property, mainly focussing on its simplest form called *entity authentication* or *identification*, which can be informally stated as follows:

> **Entity authentication** enable one entity, the *claimant*, to prove its claimed identity to another entity, the *verifier*.

Entity authentication basically amounts to reliably agreeing on the claimant identity. The application fields are extremely numerous, since identification is a basic requirement for many applications. For example, we need entity authentication for personal accounting on a system, physical access to restricted areas, payment for the use of resources or services, e-commerce, and so on.

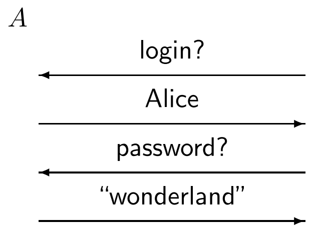
A first important classification of authentication techniques can be done depending on *what* they are based on [25]:

1. *Something known*: the claimant proves its identity by demonstrating the knowledge of a secret like, e.g., a password, a Personal Identification Number (PIN), a cryptographic key, ...
2. *Something possessed*: the claimant proves its identity by demonstrating the possession of a physical object like, e.g., a magnetic card, a smart card, a palmtop or Personal Digital Assistant (PDA), ...
3. *Something inherent*: the claimant proves its identity by showing a physical characteristic or involuntary action like, e.g., a signature, a fingerprint, e retinal pattern, the writing rate on a keyboard, ...

It is important to notice that these techniques are often combined together in order to simplify their use and complement their respective benefits. A very common example is the cell-phone smart card (i.e., *something possessed*) which contains a cryptographic key (i.e., *something known*) used to authenticate with the service provider. The user does not need to remember the fairly long cryptographic key contained in the smart card: she is just required to enter a short PIN (i.e., *something known*) which enables the smart card to authenticate with the provider. This "two-phase" authentication has the aim of simplifying the user task without decreasing the level of security. The use of cryptographic keys to authenticate will be treated in Section 2.2.

2.1 Password-Based Authentication

The simpler, and undoubtedly most used, authentication scheme is the *password-based* one. It is a *something known* scheme in which the claimant and the verifier share the knowledge of a secret password. The password is shown by the claimant in order to authenticate herself to the verifier, as illustrated below:



Notice that this scheme is perfectly safe if Alice and Bob communication is protected, but it becomes completely vulnerable whenever the password can be intercepted by a malicious third party. As a matter of fact, once password secrecy is lost, anyone knowing it can easily impersonate Alice by just re-running the protocol above with Bob. When this happens, it is violated the basic property every authentication protocol should provide:

Non-impersonation S should never be able to impersonate A with B, even after observing previous protocol runs between A and B.

This ideal property requiring the impossibility of impersonating the claimant is typically achieved in a computational form:

> **Non-impersonation (computational)** The probability that S impersonates A with B is negligible, even after observing previous protocol runs between A and B.

where negligible means “is so small that it is not of practical significance”. For example, in this case, it should be small enough to guarantee that once S has broken the protocol, A has already changed her password. This computational version of non-impersonation is of much more practical interest, since in cryptography every key can be broken given enough time either by “brute force”, i.e., by trying all the possible keys, or using some more sophisticated cryptanalysis technique. Thus, it is important to guarantee that this happens with a probability that is small enough to make the attack useless.

Even if the password-based scheme is very vulnerable on an untrusted network, like the Internet, it has been used quite widely in the past and is still used by many web-sites. The telnet protocol is a striking example: in order to connect to a remote host, authentication was implemented by following the above scheme, thus allowing an intruder to easily intercept the password sent in clear on the network. Many web-sites are still not protected by cryptography (through, e.g., the Secure Sockets Layer, SSL, protocol), and, similarly to what happens with telnet, authentication is achieved by a password or a PIN which is just sent in clear to the web server.

To develop stronger authentication protocols it is first necessary to ask ourselves where the weakness of the password-based scheme comes from. The answer is quite easy to give: proving the knowledge of a secret by just revealing it is very unsafe, especially if we want to reuse such secret for further authentications! In the next section we give the basics of *challenge-response* protocols, whose aim is exactly to show the knowledge of a secret without revealing it, thus circumventing the weakness discussed above. We will see that, even if those protocols are more solid that password-based ones, some subtle attacks can be yet easily mounted if the protocols are not carefully developed.

2.2 Challenge-Response Protocols

The idea of challenge-response protocols is to prove the knowledge of a secret without revealing it: this is done by replying to a claimant’s challenge with a response that depends on the secret without revealing much of it. More precisely, knowing the secret it must be easy to check if the response depends on it, but it must be (computationally) unfeasible to derive the secret from the response.

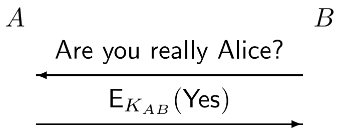
The generation of these particular responses is based on cryptographic techniques. We thus briefly review the basic notions of cryptography [29]. A cryptosystem $\mathcal{S}$ is a quintuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, E, D)$ where $\mathcal{P}$ is the set of *plaintext*, $\mathcal{C}$ is the set of *ciphertexts*, $\mathcal{K}$ is the set of *keys*, and $E : \mathcal{P} \times \mathcal{K} \rightarrow \mathcal{C}$ and $D : \mathcal{C} \times \mathcal{K} \rightarrow \mathcal{P}$ are the *encryption* and *decryption* functions, respectively. Function E, given a plaintext and an encryption key k, returns the corresponding ciphertext, while function D, given a ciphertext and the corresponding decryption key k^{-1}, returns the corresponding plaintext. We write $E_k(x)$ and $D_{k^{-1}}(y)$ in place of $E(x,k)$ and $D(y,k^{-1})$, respectively. Formally, we have that $D_{k^{-1}}(E_k(x)) = x$, meaning that decrypting an encrypted message, using the appropriate key, gives back the original message. When $k = k^{-1}$ the cryptosystem is symmetric, otherwise it is asymmetric. Every "good" cryptosystem should fulfill at least the following properties:

Property 1. the probability of encrypting and decrypting without knowing the encryption and decryption key is negligible.

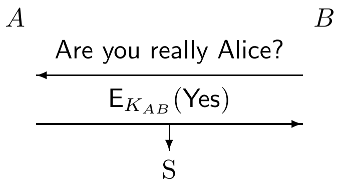
Property 2. the probability of finding the encryption and decryption keys by only observing a ciphertext is negligible.

In practice, there are no cryptosystems with an absolute proof of security in terms of the properties above. What can be proven are statements like "a given cryptosystem is secure if it is unfeasible to factor a large number", i.e., the security of cryptosystems is given in term of some well-known problem which is difficult to solve. We are not interested in giving more detail about cryptography. From now on we will assume the two properties above hold, and we invite the interested reader to refer to, e.g., [29,25] for more detail about cryptography.

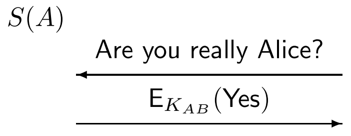
Replay Attacks. Let us now do a first attempt to implement challenge-response in a very simple, unfortunately flawed, way. Let K_{AB} be a key shared between Alice and Bob and let E be a symmetric encryption function:



When Bob receives the response, he decrypts it through K_{AB} and checks the reply. Since only Alice and Bob know the secret key, Bob might deduce that this reply really comes from Alice. As a matter of fact, by property 1, the probability for an intruder to forge the right response is negligible. Moreover, property 2 guarantees that the secret key is not compromised by divulging encrypted messages. There is however a tremendous flaw in the protocol due to the fact that the challenge is always the same and so is the expected response: the intruder S, as done in the password scheme, just needs to intercept one Alice's reply and use it for later authentication.

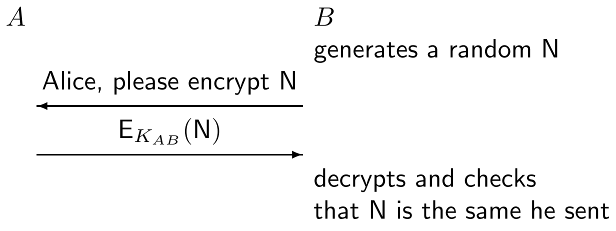


With $S(A)$ we denote S impersonating A:

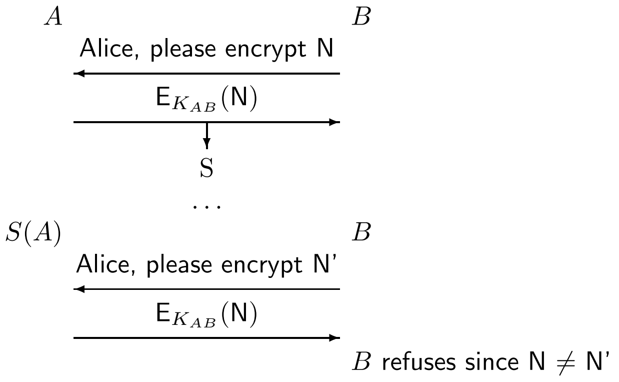


Notice that S is not breaking cryptography, but is just *replaying* a message previously sent by one of the honest parties. For this reason this kind of attacks are typically called *replay-attacks*.

So far, we have seen that having a response just depending on the secret is not of big help if it is always the same: intercepting it is equivalent to intercepting the secret and naturally leads to replay attacks. In order to make the response non-reusable we force it to be different at every protocol run. This is achieved by adding a so called *time-variant* parameter inside the response. Typical time-variant parameters are nonces, sequence numbers and time-stamps [25]. Here, for the sake of simplicity, we only consider nonces, i.e., numbers used only once that the verifier checks for avoiding replays of old messages. A simple way for implementing nonces is to use large enough random numbers that are used for just one session. This idea is better illustrated through an example:



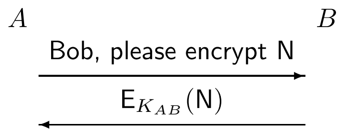
B decrypts the received message using K_{AB} and accepts only if the result is equal to nonce N. Let us see how this prevents replay attacks:


A
B
Alice, please encrypt N
$\mathsf{E}_{K_{AB}}(\mathsf{N})$
S
...
$S(A)$
B
Alice, please encrypt N'
$\mathsf{E}_{K_{AB}}(\mathsf{N})$
B refuses since N $\neq$ N'

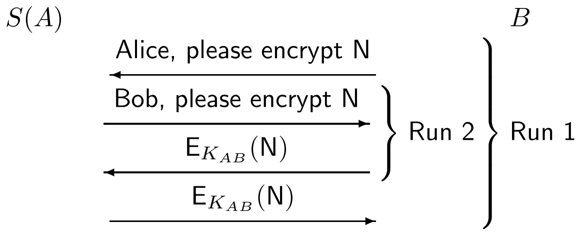
Since the response must contain the randomly generated nonce, S cannot reuse previously intercepted responses. To summarize, we always need that

– Challenge is time-variant;
– Response depends on both the challenge and the secret.

Reflection Attacks. Even with time-variant challenges and responses, there is still a source of possible flaws due to the ambiguity of encrypted messages. Let us assume that the protocol above can indifferently be run by Alice and Bob to authenticate with Bob and Alice, respectively. In other words, we assume that the protocol can also be run with Alice as verifier and Bob as claimant, as illustrated below:



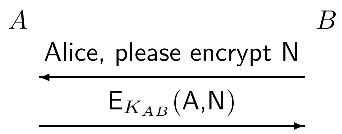
In this (quite realistic) setting, it might happen that the same message is interpreted in two different ways, leading to a security breach. Let us see a concrete example of the so called *reflection-attack*:


$S(A)$
B
Alice, please encrypt N
Bob, please encrypt N
$\mathsf{E}_{K_{AB}}(\mathsf{N})$
$\mathsf{E}_{K_{AB}}(\mathsf{N})$
Run 2
Run 1

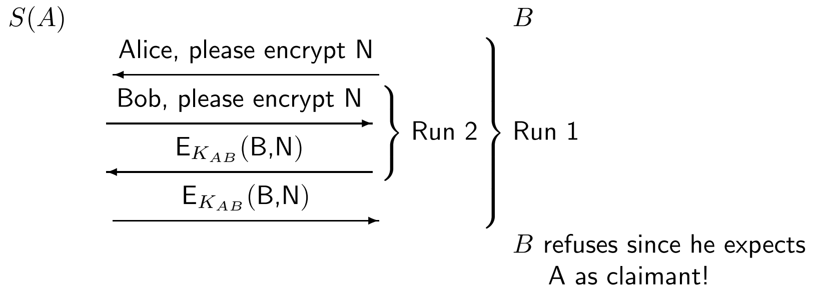
The first and the last message are part of Run 1 where A should authenticate with B. S succeeds impersonating A by starting a second run where he asks B to authenticate himself apparently with A. To do so S "reflects" in the second message the nonce N to B and obtains as a response the encryption $\mathsf{E}_{K_{AB}}(\mathsf{N})$. This can be reflected to B to conclude the first run where B asked A to encrypt N. There are thus two parallel sessions of the same protocol with exchanged roles. Notice that A might be even off-line, since all the encryptions are performed by B himself.

The flaw is based on the ambiguity of message $\mathsf{E}_{K_{AB}}(\mathsf{N})$ that does not clarify whether it is a response from Alice to Bob or from Bob to Alice. The use of symmetric key encryption contributes to this ambiguity since both Alice and Bob could have encrypted such a message.

The easiest way to make explicit an information is just to include it inside the encryption, e.g., indicating the claimant of the session:



Bob now checks both the nonce and the identifier of Alice. If we try to mount the reflection attack we obtain the following execution:


$S(A)$
B
Alice, please encrypt N
Bob, please encrypt N
$\mathsf{E}_{K_{AB}}(\mathsf{B,N})$
$\mathsf{E}_{K_{AB}}(\mathsf{B,N})$
Run 2
Run 1
B refuses since he expects A as claimant!

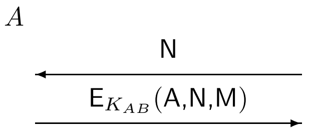
Thus, together with a time-variant challenge and a response that depends both on the challenge and on the secret, we also need that ciphertexts are explicit enough about who is the claimant and who is the verifier. In many cases, it is sufficient to specify one of the two, since the other one is deduced by the cryptographic key. For example, in the protocol above we have only specified the claimant since the verifier is implicitly the other entity sharing the key with the claimant (we are assuming that the key is shared between only two participants). Based on this notions it is possible to develop many protocol variants depending on which entity is specified (A or B) and, more interestingly, depending on which messages are encrypted, as discussed below.

2.3 Three Classes of Challenge-Response Protocols

Here, we extend the discussion above to asymmetric encryption and we write SK_A and PK_A to denote the private and the public key of entity A. We assume that only A knows her private key while every entity knows everyone else's public key; moreover all messages encrypted with SK_A and PK_A can be decrypted using PK_A and SK_A, respectively. As a consequence, everyone can generate ciphertext $\mathsf{E}_{\mathsf{PK}_A}(M)$ but only Alice can decrypt it; on the other hand, only Alice can generate ciphertext $\mathsf{E}_{\mathsf{SK}_A}(M)$, representing a digital signature, but every other entity can decrypt it.

We classify nonce-based protocols into three categories depending on what is encrypted and what is sent in clear [11,8,12].

Plain-Cipher (PC). B sends out the nonce in clear and receives it back encrypted together with a message which is authenticated. A proves her identity to B by showing the knowledge of the encryption key. An example is the protocol discussed so far, that we extend by adding a message M that Alice wants to authenticate with Bob:



This protocol authenticates A sending message M to B, since only A may have generated the ciphertext. The same effect can be achieved by using B in place of A in the ciphertext, or, in an asymmetric cryptosystem, using private Alice's key SK_A in place of K_{AB}. In this latter case it is necessary to specify B in place of A in the ciphertext, since the private key only indicates the claimant A.

Cipher-Plain (CP). B sends out the nonce encrypted and receives it back in clear. A proves her identity to B by showing the knowledge of the decryption key. For example:

$$\begin{tikzcd} A & B \\ {} & {} \arrow[l, "\mathsf{E}_{K_{AB}}(\mathsf{B,N,M})"'] \\ {} \arrow[r, "\mathsf{N}"] & {} \end{tikzcd}$$

This protocol authenticates A receiving message M from B, since only A may have decrypted the ciphertext. A similar effect is achieved by using A in place of B in the ciphertext, or, with asymmetric keys, using PK_A in place of the symmetric key K_{AB}.

Cipher-Cipher (CC). The nonce is sent out and received back encrypted. This is useful if both entities want to exchange messages. A proves her identity to B by showing the knowledge of either the encryption key (as in PC) or the decryption key (as in CP). For example:

$$\begin{tikzcd} A & B \\ {} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{B},\mathsf{M}_B,\mathsf{N})"'] \\ {} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{A},\mathsf{M}_A,\mathsf{N})"] & {} \end{tikzcd}$$

This protocol authenticates A sending message M_A to B and receiving message M_B from B, as only A may have decrypted the first ciphertext. Again, the same effect may be achieved using either the private keys SK_B and SK_A, or a symmetric key K_{AB} in place of the two public keys.

Notice that the above mentioned categories are a generalization of POSH (Public Out Secret Home), SOPH (Secret Out Public Home) and SOSH (Secret Out Secret Home), introduced in [18]. We actually relax "Secret" into "Cipher" as handshakes may be composed of signed messages, guaranteeing integrity rather than secrecy. For example, as already noticed, PC includes protocols with a cleartext challenge and a signed response, which would not "fit well" into the POSH category since the nonce is not sent back as a secret. Notice also that PC and CP are called incoming and outgoing tests in [19].

2.4 A Well Known Example: The Needham-Schroeder Protocol

We illustrate a very well known case study that was discovered to be flawed by Gavin Lowe in [23]. Using the classification above it is rather easy to see how the protocol works and where is actually flawed. The original protocol is as follows:

$$\begin{tikzcd}
A & B \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{A},\mathsf{N}_A)"] & {} \\
{} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)"'] \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{N}_B)"] & {}
\end{tikzcd}$$

The aim is to achieve mutual authentication between A and B. To understand the protocol logic it is useful to decompose it into two (chained) challenge-response subprotocols, each achieving unilateral authentication. The first two messages are indeed a CC challenge-response in which Bob is authenticated to Alice:

$$\begin{tikzcd}
A & B \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{A},\mathsf{N}_A)"] & {} \\
{} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)"']
\end{tikzcd}$$

Similarly, the second and the third message should authenticate Alice to Bob:

$$\begin{tikzcd}
A & B \\
{} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)"'] \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{N}_B)"] & {}
\end{tikzcd}$$

However, it is quite evident that this subprotocol does not specify B as verifier as done for A in the first subprotocol. This lack is indeed the source of the man-in-the-middle attack discovered in [23] and illustrated below:

$$\begin{tikzcd}
A & S(A) & & B \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_E}(\mathsf{A},\mathsf{N}_A)"] & {} & & \\
& & {} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{A},\mathsf{N}_A)"] & {} \\
& & {} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)"'] \\
{} & {} \arrow[l, "\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)"'] & & \\
{} \arrow[r, "\mathsf{E}_{\mathsf{PK}_E}(\mathsf{N}_B)"] & {} & & \\
& & {} \arrow[r, "\mathsf{E}_{\mathsf{PK}_B}(\mathsf{N}_B)"] & {}
\end{tikzcd}$$

We have a correct session between A and S which is exploited by S to impersonate A with B, on the right-hand side. The trick is that the response sent by B does not contain his name and can be thus forwarded by S to A in order to be decrypted. In the end, B is convinced that A has decrypted N_B for him but instead Alice has decrypted such a nonce for S. So, Bob authenticates Alice but Alice has started the protocol with the enemy S. How to patch the protocol should be clear from the discussion above, and we leave it as an exercise.

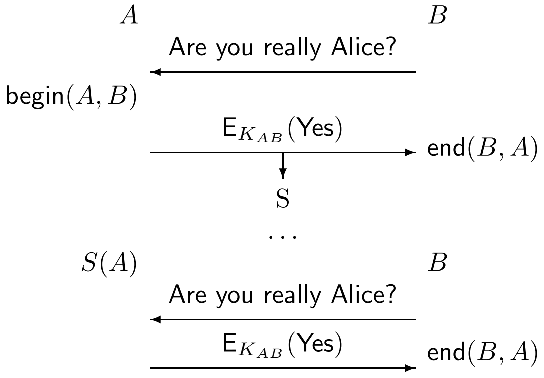
3 Authentication as Correspondence

In this section, we discuss how to formalize authentication as a property of protocol behaviour. Differently from, e.g., secrecy, authentication is not easily formulated as a property of the execution state. For example, for secrecy we can require that we never reach a state where secret data have been learnt by the intruder S. What about authentication? How can we formalize that S has impersonated Alice? A very powerful idea was proposed by Woo and Lam in [31]. Let us assume that the start and the termination of a protocol session generate two observable events, say $\mathsf{begin}(A, B)$ and $\mathsf{end}(B, A)$.

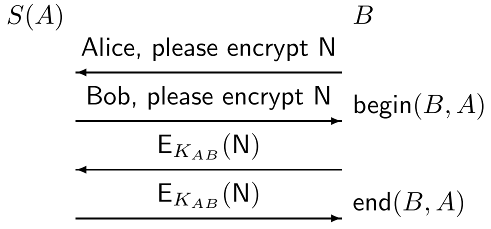
Definition 1 (Correspondence). *A protocol guarantees authentication of A with B iff every* $\mathsf{end}(B, A)$ *is matched by a previous distinguished* $\mathsf{begin}(A, B)$.

Intuitively, every time Bob authenticates Alice, then Alice should have started a session with Bob. We see how this formalization works by reconsidering the previously illustrated flaws.

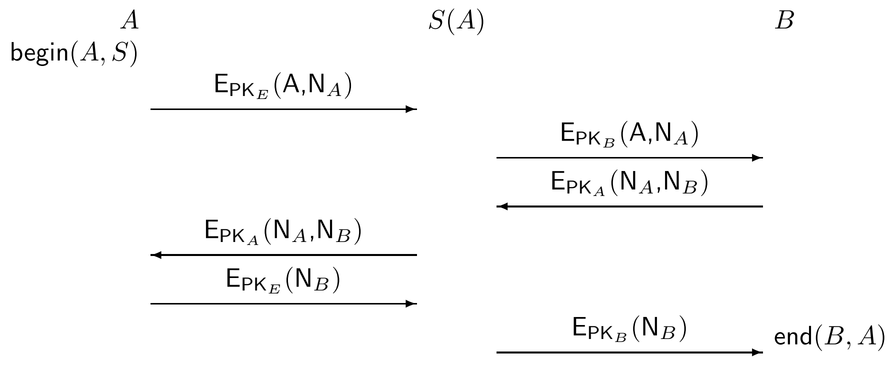
A reply attack is characterized by many end's with just one begin. In fact, messages from one session are replayed in order to impersonate Alice:



A reflection attack is instead captured since we have a $\mathsf{end}(B, A)$ with no matching $\mathsf{begin}(A, B)$, but with $\mathsf{begin}(B, A)$ instead. As a matter of fact, A might be even off-line during the attack, and a parallel run of B is exploited to mount the attack:



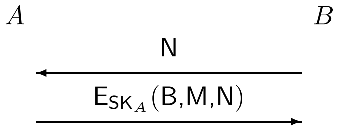
The man-in-the-middle attack is captured since $\mathsf{end}(B, A)$ is not matched by the required $\mathsf{begin}(A, B)$, but by $\mathsf{begin}(A, S)$ instead. Alice is running with S and this session is used to impersonate her with Bob:


A
$S(A)$
B
$\mathsf{begin}(A,S)$
$\mathsf{E}_{\mathsf{PK}_E}(\mathsf{A},\mathsf{N}_A)$
$\mathsf{E}_{\mathsf{PK}_B}(\mathsf{A},\mathsf{N}_A)$
$\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)$
$\mathsf{E}_{\mathsf{PK}_A}(\mathsf{N}_A,\mathsf{N}_B)$
$\mathsf{E}_{\mathsf{PK}_E}(\mathsf{N}_B)$
$\mathsf{E}_{\mathsf{PK}_B}(\mathsf{N}_B)$
$\mathsf{end}(B,A)$

In the next section we show how this dynamic correspondence property may be proved statically, i.e., by only inspecting the protocol code.

4 Static Analysis

Developing static analyses for authentication protocols is particularly difficult since authentication is typically provided by encrypting/decrypting messages which are composed of different parts, each of them providing a "piece" of the authentication guarantee. As an example, consider the following PC protocol:


A
B
N
$\mathsf{E}_{\mathsf{SK}_A}(\mathsf{B},\mathsf{M},\mathsf{N})$

in which Bob is sending to Alice a random nonce N which is signed by Alice together with a message M and Bob identifier B. Let us recall how each message component of $\mathsf{E}_{\mathsf{SK}_A}(\mathsf{B},\mathsf{M},\mathsf{N})$ contributes in achieving authentication: N is needed to guarantee *freshness*, i.e., $\mathsf{E}_{\mathsf{SK}_A}(\mathsf{B},\mathsf{M},\mathsf{N})$ cannot be a replay of an old protocol session since N is used only once; B specifies the intended receiver of M and the signature guarantees that A is the sender.

Our approach, first introduced in [10], is based on considering a "minimal" set of authentication *patterns* that allow parties to authenticate. Through suitable tags, we make explicit in the encrypted message which pattern it correspond to. For example message $\mathsf{E}_{\mathsf{SK}_A}(\mathsf{B},\mathsf{M},\mathsf{N})$ is tagged as $\mathsf{E}_{\mathsf{SK}_A}(\mathsf{Id}(\mathsf{B}),\mathsf{Auth}(\mathsf{M}),\mathsf{Verif}_{\mathsf{PC}}(\mathsf{N}))$ to denote the fact that the nonce N is authenticating M to the verifier B in a PC nonce handshake. When a message is decrypted, tags are recognized by the party that may exploit them to achieve/provide authentication guarantees.

The advantages of this tag-based approach may be summarized as follows: (*i*) it is extremely compositional: tags constitute a common trusted "interface" that distributed parties may use to authenticate each other; the correct use of tags is enforced locally (through suitable typing rules) and correctness is preserved when communicating parties are concurrently executed; (*ii*) this strong form of compositionality allows to safely mix different protocols once their sequential components are type-checked; thus, our tagging discipline naturally scales to multi-protocol settings; (*iii*) the fact that tags correspond to a small set of a-priori selected patterns, makes the type system quite simple and easy to use; in [15] we have implemented a tool that is able to automatically infer tags and types needed to validate protocols, thus obtaining a completely automated validation technique for protocol with unbounded number of sessions and participants; (*iv*) even if the set of authentication patterns we have selected is small, it is expressive enough to capture many of the protocols in literature; this gives also new insights on which are the basic mechanisms for guaranteeing authentication.

In this section we give an overview of the technique. In particular, Section 4.1 presents the ρ-spi calculus, a simple language for specifying cryptographic protocols, Section 4.2 illustrates the types-and-effect system and Section 4.3 discusses the main safety results. More detail can be found in [8,12].

4.1 A Simple Calculus for Protocols

The ρ-spi calculus derives from the spi calculus [2], and inherits many of the features of *Lysa* [5], a version of the spi calculus proposed for the analysis of authentication protocols. ρ-spi differs from both calculi in several respects: it incorporates the notion of tagged message exchange from [9], it provides new authentication-specific constructs, and offers primitives for declaring process identities and keys.

Table 1. The syntax of ρ-spi calculus

Notation: $TAG \in \{\mathsf{Id}, \mathsf{Auth}, \mathsf{Verif}_H, \mathsf{Claim}_H | H = \mathsf{PC}, \mathsf{CP}, \mathsf{CC?}, \mathsf{CC!}\}$
$m \in \mathcal{N} \cup \mathcal{V}$

$\mathcal{M}, \mathcal{K}$ ::=	*Patterns*	P, Q ::=	*Processes*
a, b, k, n	names	$I \rhd S$	(principal)
x, y, z	variables	$I \rhd !S$	(replication)
$\mathsf{Pub}(m)$	public key	$P \vert Q$	(composition)
$\mathsf{Priv}(m)$	private key	$\mathsf{let}\ k = \mathsf{sym\text{-}key}(I_1, I_2).P$	(symmetric-key)
$TAG(\mathcal{M})$	tagged pattern	$\mathsf{let}\ k = \mathsf{asym\text{-}key}(I).P$	(asymmetric-key)
$(\mathcal{M}_1, \mathcal{M}_2)$	pair		

S ::= *Sequential Processes*	
$\mathbf{0}$	(nil)
$\mathsf{new}(n).S$	(restriction)
$\mathsf{in}(\mathcal{M}).S$	(input)
$\mathsf{out}(\mathcal{M}).S$	(output)
$\mathsf{encrypt}\{\mathcal{M}\}_{\mathcal{K}}\ \mathsf{as}\ x.S$	(symmetric encryption)
$\mathsf{encrypt}\{\vert\mathcal{M}\vert\}_{\mathcal{K}}\ \mathsf{as}\ x.S$	(asymmetric encryption)
$\mathsf{decrypt}\ x\ \mathsf{as}\ \{\mathcal{M}\}_{\mathcal{K}}.S$	(symmetric decryption)
$\mathsf{decrypt}\ x\ \mathsf{as}\ \{\vert\mathcal{M}\vert\}_{\mathcal{K}}.S$	(asymmetric decryption)
$\mathsf{begin}(I_1, I_2, \mathcal{M}_1; \mathcal{M}_2).S$	(begin)
$\mathsf{end}(I_1, I_2, \mathcal{M}_1; \mathcal{M}_2).S$	(end)

The syntax is reported in Table 1 and described below. Patterns, denoted by $\mathcal{M}, \mathcal{K}$, are recursively defined over names, variables, public and private keys, tagged patterns and pairs We presuppose two countable sets: $\mathcal{N}$ of names and $\mathcal{V}$ of variables. We reserve a, b, k, n for names and x, y, z for variables, with m ranging over both names and variables. Identities $\mathcal{ID}$ are a subset of names and are ranged over by I and J. Identities are further partitioned into *trusted principals* $\mathcal{ID}_{\mathcal{P}}$, ranged over A and B, and *enemies* $\mathcal{ID}_{\mathcal{E}}$, ranged over by E. The pair composed by a public key and the corresponding private one is noted by $\mathsf{Pub}(m)$, $\mathsf{Priv}(m)$, similarly to [2]. In the rest of the paper, we will use the following notation convention: $\overline{\mathsf{Pub}} = \mathsf{Priv}$ and vice-versa. Tags, denoted by TAG, are a special category of names. They specify the role of each message component. Specifically: all identifiers relevant to authentication are tagged by Id; messages that should be authenticated are tagged by Auth; finally, nonces are tagged by Verif_H or Claim_H, with $H \in \{\mathsf{PC}, \mathsf{CP}, \mathsf{CC!}, \mathsf{CC?}\}$. Such tags specify the role played by the entity tagged by Id (verifier or claimant) and the kind of nonce hand-shake. Notice that in CC nonce handshakes we distinguish challenge from response ciphertexts, denoted by $\mathsf{CC?}$ and $\mathsf{CC!}$, respectively.

Processes (or *protocols*), ranged over by P, Q, are the parallel composition of principals. Each principal is a sequential process associated with an identity I, noted $I \rhd S$. The replicated form $I \rhd !S$ indicates an arbitrary number of copies of $I \rhd S$. In order to allow the sharing of keys among principals, we provide ρ-spi with let-bindings: $\mathsf{let}\ k = \mathsf{sym\text{-}key}(I_1, I_2).P$ declares (and binds) the long-term key k shared between I_1 and I_2 in the scope P. Similarly, $\mathsf{let}\ k = \mathsf{asym\text{-}key}(I).P$ declares, and binds in the scope P, the key pair $\mathsf{Pub}(k)$, $\mathsf{Priv}(k)$ associated to I.

Sequential processes may never fork into parallel components: this assumption helps assign unique identities to (sequential) processes, and involves no significant loss of expressive power as protocol principals are typically specified as sequential processes, possibly sharing some long-term keys. The sequential process $\mathbf{0}$ is the null process that does nothing, as usual. Process $\mathsf{new}(n).S$ generates a fresh name n local to S. We presuppose a unique (anonymous) public channel, the network, from/to which all principals, including intruders, read/send messages. Similarly to *Lysa*, our input primitive may (atomically) test part of the message read, by pattern-matching. If the input message matches the pattern, then the variables occurring in the pattern are bound to the remaining subpart of the message; otherwise the message is not read at all. For example, process '$\mathsf{in}(\mathsf{Claim}(x)).P$' may only read messages of the form $\mathsf{Claim}(M)$, binding x to M in P. Encryption just binds x to the encrypted message, while decryption checks if the message contained in x matches the form $\{\mathcal{M}\}_{\mathcal{K}}$ (or $\{|\mathcal{M}|\}_{\mathcal{K}}$), i.e., the payload matches $\mathcal{M}$ and is encrypted with the appropriate key. Only in this case x is decrypted and the variables in the pattern $\mathcal{M}$ get bound to the decrypted messages. Similarly to the input primitive, decryption may also test part of the decrypted messages by pattern-matching mechanism. Finally, the $\mathsf{begin}(I_1, I_2, \mathcal{M}_1; \mathcal{M}_2).S$ and $\mathsf{end}(I_2, I_1, \mathcal{M}_1; \mathcal{M}_2).S$ primitives are used to check the *correspondence assertions* [31]. The former primitive declares that I_1 is beginning a protocol session with I_2 for confirming the reception of message

$\mathcal{M}_1$ and authenticating message $\mathcal{M}_2$; the latter one indicates that I_2 is ending a protocol session with I_1 getting confirmation from I_1 of the reception of message $\mathcal{M}_1$ and authenticating message $\mathcal{M}_2$.

Example 1. To illustrate the use of tags and correspondence assertions, let us consider the protocols presented so far. The first protocol of Section 2.3 can be decorated with tags and correspondence assertions as follows:

$$\begin{tikzcd}[column sep=huge]
& A & & B & \\
& {} & & {} \arrow[ll, "N"'] & \\
\mathsf{begin}(A,B;M) & {} \arrow[rr, "{\mathsf{E}_{K_{AB}}(\mathsf{Id}(A),\mathsf{Claim}_{\mathsf{PC}}(N),\mathsf{Auth}(M))}"] & & {} & \\
& & & & \mathsf{end}(B,A;M)
\end{tikzcd}$$

The tagged structure specifies that that the nonce N is used by the verifier B for authenticating M with A in a PC nonce hand-shake. According to the correspondence assertions, at the end of the protocol B authenticates A *sending* message M.

The second protocol of Section 2.3 can be decorated as follows:

$$\begin{tikzcd}[column sep=huge]
& A & & B & \\
& {} & & {} \arrow[ll, "{\mathsf{E}_{K_{AB}}(\mathsf{Id}(B),\mathsf{Verif}_{\mathsf{CP}}(N),\mathsf{Auth}(M))}"'] & \\
\mathsf{begin}(A,B,M;) & & & & \\
& {} \arrow[rr, "N"] & & {} & \\
& & & & \mathsf{end}(B,A,M;)
\end{tikzcd}$$

Indeed, the nonce N is used by the verifier B for authenticating M with A in a CP nonce handshake. According to the correspondence assertions, at the end of the protocol B authenticates A *receiving* message M.

Finally, the third protocol of Section 2.3 can be decorated as follows:

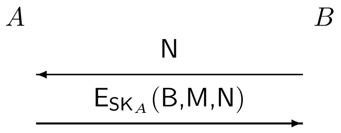
$$\begin{tikzcd}[column sep=huge]
& A & & B & \\
& {} & & {} \arrow[ll, "{\mathsf{E}_{\mathsf{PK}(A)}(\mathsf{Id}(B),\mathsf{Auth}(M_B),\mathsf{Verif}_{\mathsf{CC?}}(N))}"'] & \\
\mathsf{begin}(A,B,M_B;M_A) & {} \arrow[rr, "{\mathsf{E}_{\mathsf{PK}(B)}(\mathsf{Id}(A),\mathsf{Auth}(M_A),\mathsf{Claim}_{\mathsf{CC!}}(N))}"] & & {} & \\
& & & & \mathsf{end}(B,A,M_B;M_A)
\end{tikzcd}$$

Notice that the use of ? and ! in the nonce tag disambiguates whether the ciphertext is used in a CC handshake as challenge or response. By an inspection of the correspondence assertions, at the end of the protocol B authenticates A receiving message M_B and sending message M_A. □

Table 2. PC Protocol in ρ-spi calculus

$$\begin{aligned}
\mathit{Protocol} &\triangleq \mathsf{let}\ k_A = \mathsf{asym\text{-}key}(A)\ .\ (B \rhd !\mathit{Initiator} \mid A \rhd !\mathit{Responder}) \\
\mathit{Initiator} &\triangleq \mathsf{new}(n).\mathsf{out}(n).\mathsf{in}(z). \\
&\qquad \mathsf{decrypt}\ z\ \mathsf{as}\ \{\!|\mathsf{Id}(B), \mathsf{Auth}(x), \mathsf{Verif}_{\mathsf{PC}}(n)|\!\}_{\mathsf{Pub}(k_A)}.\mathsf{end}(B, A; x).\mathbf{0} \\
\mathit{Responder} &\triangleq \mathsf{in}(x).\mathsf{new}(m).\mathsf{begin}(A, B; m). \\
&\qquad \mathsf{encrypt}\ \{\!|\mathsf{Id}(B), \mathsf{Auth}(m), \mathsf{Verif}_{\mathsf{PC}}(x)|\!\}_{\mathsf{Priv}(k_A)}\ \mathsf{as}\ z.\mathsf{out}(z).\mathbf{0}
\end{aligned}$$

Example 2. To illustrate the ρ-spi calculus syntax, let us consider the following public key variant of the first protocol in Example 1.



The difference is that A signs the response instead of using symmetric cryptography. The ρ-spi calculus specification is in Table 2. After declaring the key pair for A, an unbounded number of instances of B as initiator and an unbounded number of instances of A as responder are run in parallel. The *Initiator* B generates a fresh nonce and sends it in clear on the network. Then it reads a message from the network and tries to decrypt it with the public key of the responder A and checks that its own identifier tagged by Id is the first component of the message payload and the nonce tagged by $\mathsf{Verif}_{\mathsf{PC}}$ is fresh, i.e., it is the one just generated. If this is the case, B authenticates A sending message x, namely the message tagged by Auth. The *Responder* A receives a nonce from the network, generates a new message m, declares the start of the session with the initiator B for authenticating message m, signs m together with the responder identifier and the nonce (all message components are tagged) and sends the obtained ciphertext on the network. $\square$

Operational Semantics. We define the operational semantics of ρ-spi in terms of *traces*, after [7]. A trace is a possible sequence of *actions* performed by a process. Each process primitive has an associated action and we denote with *Act* the set of all possible actions. The dynamics of the calculus is formalized by means of a transition relation between *configurations*, i.e., pairs $\langle s, P\rangle$, where $s \in \mathit{Act}^*$ is a trace, P is a (closed) process. Each transition $\langle s, P\rangle \rightarrow \langle s :: \alpha, P'\rangle$ simulates one computation step in P and records the corresponding action in the trace.

Principals do not directly synchronize with each other. Instead, they may receive from the unique channel an arbitrary message M known by the environment, which models the Dolev-Yao intruder: the environment knows all the identity labels, the messages sent on the network, the content of ciphertexts whose decryption key is known, ciphertexts created by its knowledge and all the keys declared as owned by E together with all the public keys. Finally, it may create fresh names not appearing in the trace. The transition relation is given in detail in [8,12].

Definition 2 (Traces). *The set $T(P)$ of traces of process P is the set of all the traces generated by a finite sequence of transitions from the configuration $\langle \epsilon, P \rangle$: $T(P) = \{s \mid \exists P' \text{ s.t. } \langle \epsilon, P \rangle \rightarrow^* \langle s, P' \rangle\}$*

The notion of safety extends the standard *correspondence* property of [24,31] by distinguishing between received and sent messages.

Definition 3 (Safety). A trace s is safe if and only if whenever $s = s_1 :: end(B, A, M_1; M_2) :: s_2$, then $s_1 = s_1' :: begin(A, B, M_1; M_2) :: s_1''$, and $s_1' :: s_1'' :: s_2$ is safe. A process P is safe if, $\forall s \in T(P), s$ is safe.

A trace is *safe* if every $end(B, A, M_1; M_2)$ action is preceded by a distinct $begin(A, B, M_1; M_2)$. Intuitively, this guarantees that whenever B authenticates A receiving M_1 and sending M_2, then A has received M_1 and has sent M_2 in a protocol session with B.

Example 3. To illustrate the semantics of the calculus and the notion of safety, let us consider a (flawed) simplification of the protocol of Example 2, obtained by eliminating the nonce.

$$\begin{tikzcd} A \arrow[r, "\mathsf{E}_{\mathsf{SK}_A}(\mathsf{B,M})"] & B \end{tikzcd}$$

The ρ-spi calculus specification may be easily obtained from the one of the original protocol in Table 2:

$$\begin{aligned} \mathit{Initiator}_f &\triangleq \mathsf{in}(z).\mathsf{decrypt}\ z\ \mathsf{as}\ \{|B, x|\}_{\mathsf{Pub}(k_A)}.\mathsf{end}(B, A; x).\mathbf{0} \\ \mathit{Responder}_f &\triangleq \mathsf{new}(m).\mathsf{begin}(A, B; m).\mathsf{encrypt}\ \{|B, m|\}_{\mathsf{Priv}(k_A)}\ \mathsf{as}\ z.\mathsf{out}(z).\mathbf{0} \end{aligned}$$

This protocol suffers of the standard replay attack where S impersonates A by just replaying a previously intercepted message, which can be observed through the following execution trace:

$$\begin{array}{l} asym - key(k_A, A) :: new(m) :: \mathbf{begin}(A, B; m) :: \\ encrypt\{|B, m|\}_{\mathsf{Priv}(k_A)} :: out(\{|B, m|\}_{\mathsf{Priv}(k_A)}) :: \\ in(\{|B, m|\}_{\mathsf{Priv}(k_A)}) :: decrypt\{|B, m|\}_{\mathsf{Priv}(k_A)} :: \mathbf{end}(B, A; m) :: \\ in(\{|B, m|\}_{\mathsf{Priv}(k_A)}) :: decrypt\{|B, m|\}_{\mathsf{Priv}(k_A)} :: \mathbf{end}(B, A; m) \end{array}$$

Notice that the same message $\{B, m\}_{\mathsf{Priv}(k_A)}$ is read twice by two different instances of the Responder. This causes two ends with just one begin, thus making this trace unsafe. The presence of the nonce repairs the protocol avoiding this replay attack. □

4.2 ρ-Spi Calculus Types and Effects: An Overview

Types in ρ-spi calculus regulate the use of terms in the authentication task and are reported below.

$T ::=$	$\mathsf{SharedKey}(I, J)$	symmetric key
	$\mathsf{Key}(I)$	asymmetric seed
	$\mathsf{PublicKey}(I)$	public key
	$\mathsf{PrivateKey}(I)$	private key
	Un	untrusted
	$\mathsf{Nonce}^{\ell}(I, J)$	nonce
	$\mathsf{Enc}(e; f)$	ciphertext

Key types aim at preserving key secrecy and regulating the correct use of keys: a long-term key shared between I and J has type $\mathsf{SharedKey}(I, J)$ and it can only be used by I and J; a name used for creating a key pair owned by I has type $\mathsf{Key}(I)$; the public and private keys generated by that name have type $\mathsf{PublicKey}(I)$ and $\mathsf{PrivateKey}(I)$, respectively. While private keys can only be used by their owners, public keys are available to every principal. Every untagged term potentially known by the enemy has type Un.

Nonce and ciphertext types are more specifically related to authentication: A nonce used by I and J has type $\mathsf{Nonce}^{\ell}(I, J)$, where $\ell \in \{\mathit{Un}, \mathit{Ciph}, \mathit{Priv}, \mathit{Int}\}$ specifies secrecy and integrity properties of the nonce: it is *Un* when the nonce is sent in clear on the network (in PC challenges and CP responses), *Ciph* when the nonce is sent encrypted but it is supposed to be sent back in clear (CP challenges), *Priv* when the nonce secrecy is preserved by the hand-shake (CC hand-shakes using public or symmetric keys) and *Int* when the integrity of the nonce is guaranteed by signatures (CC hand-shakes using private keys). Ciphertexts encrypted by trusted principals have type $\mathsf{Enc}(e; f)$, where e and f are effects conveyed or required but such a ciphertext when it is decrypted and encrypted. Effects have the following syntax:

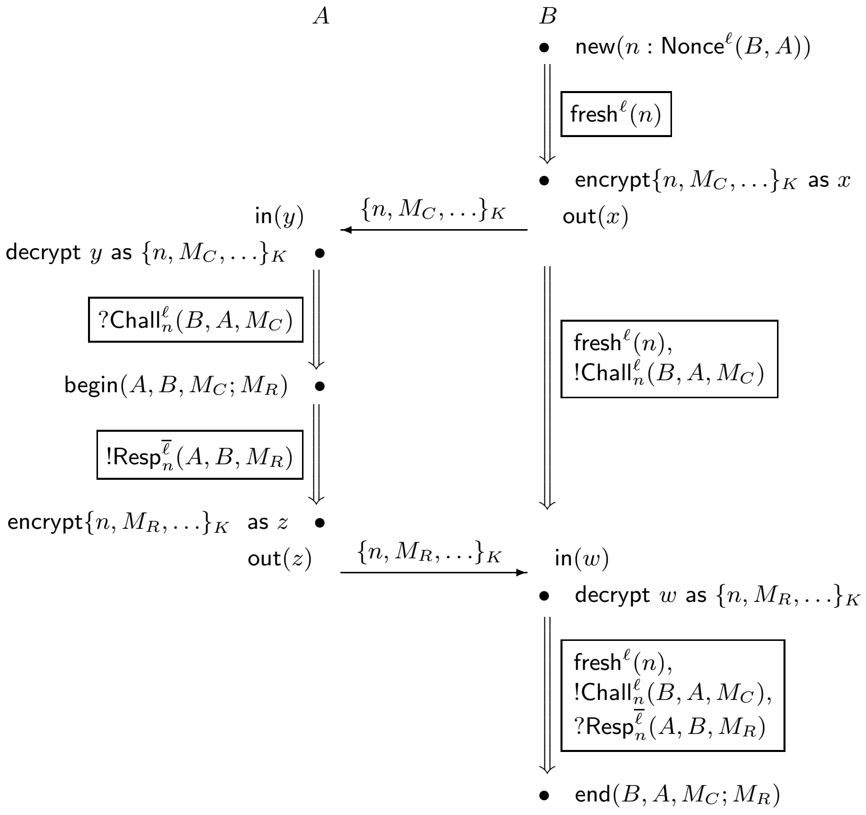
$t ::=$	$\mathsf{fresh}^{\ell}(n)$	nonce freshness
	$[?\vert!]\mathsf{Chall}^{\ell}_{N}(I, J, M)$	challenge
	$[?\vert!]\mathsf{Resp}^{\ell}_{N}(I, J, M)$	response

If the ciphertext is a challenge sent by I to J containing the nonce N and the message M, then it will have type $\mathsf{Enc}(\mathsf{Chall}^{\ell}_{N}(I, J, M), \_)$, Similarly, if the ciphertext is a response sent by I to J containing the nonce N and the message M, then the type will be $\mathsf{Enc}(\_, \mathsf{Resp}^{\ell}_{N}(I, J, M))$. So, in $\mathsf{Enc}(e, f)$, e is used for challenge effects and f for response effects. The crucial point is that these ciphertext types can be reconstructed by the tags, i.e., tagged messages convey enough information to decide whether they are challenges or responses, who are the parties involved, which is the nonce, which are the authenticated messages and which kind of challenge-response mechanism is used (label $\ell \in \{\mathit{Un}, \mathit{Ciph}, \mathit{Priv}, \mathit{Int}\}$).

The intuitive meaning of the remaining effects is given below:

– The atomic effect $\mathsf{fresh}^{\ell}(n)$ tracks the freshness of the nonce n. A nonce is fresh if it is new and it has not yet been used for justifying an end event.

Table 3. The use of types and effects


A
B
$\mathsf{new}(n : \mathsf{Nonce}^{\ell}(B, A))$
$\boxed{\mathsf{fresh}^{\ell}(n)}$
$\mathsf{encrypt}\{n, M_C, \ldots\}_K$ as x
$\mathsf{in}(y)$
$\{n, M_C, \ldots\}_K$
$\mathsf{out}(x)$
$\mathsf{decrypt}\ y$ as $\{n, M_C, \ldots\}_K$
$\boxed{?\mathsf{Chall}^{\ell}_{n}(B, A, M_C)}$
$\mathsf{fresh}^{\ell}(n)$, $!\mathsf{Chall}^{\ell}_{n}(B, A, M_C)$
$\mathsf{begin}(A, B, M_C; M_R)$
$\boxed{!\mathsf{Resp}^{\overline{\ell}}_{n}(A, B, M_R)}$
$\mathsf{encrypt}\{n, M_R, \ldots\}_K$ as z
$\mathsf{out}(z)$
$\{n, M_R, \ldots\}_K$
$\mathsf{in}(w)$
$\mathsf{decrypt}\ w$ as $\{n, M_R, \ldots\}_K$
$\mathsf{fresh}^{\ell}(n)$, $!\mathsf{Chall}^{\ell}_{n}(B, A, M_C)$, $?\mathsf{Resp}^{\overline{\ell}}_{n}(A, B, M_R)$
$\mathsf{end}(B, A, M_C; M_R)$

- The atomic effect $?\mathsf{Chall}^{\ell}_{N}(I, J, M)$ tracks the decryption of a ciphertext representing a challenge, containing the nonce N and the message M, sent by I to J. Similarly for the atomic effect $?\mathsf{Resp}^{\ell}_{N}(I, J, M)$.
- The atomic effect $!\mathsf{Chall}^{\ell}_{N}(I, J, M)$ tracks the generation of a challenge ciphertext, containing the nonce N and the message M, sent by I to J.
- The atomic effect $!\mathsf{Resp}^{\ell}_{N}(I, J, M)$, instead, *enables* the generation of a ciphertext representing a response, containing the nonce N and the message M, sent by I to J.

We illustrate, through a simple example, the use of types and effects. Let us consider a CC handshake based on symmetric cryptography, allowing the initiator B to authenticate the responder A receiving message M_C and sending message M_R. The narration of the protocol, decorated with correspondence assertions, types and effects, is depicted in Table 3. Protocol code is written close to •'s and message exchanges (left/right arrows), the nonce type is declared inside the new command, and the effects are enclosed in frame boxes placed aside down-arrows. The exchanged messages are not completely specified, since we want to give an idea of the general scheme adopted by the static analysis.

The initiator B generates a fresh nonce n of type $\mathsf{Nonce}^{\ell}(B,A)$: the atomic effect $\mathsf{fresh}^{\ell}(n)$ tracks the freshness of the nonce. The corresponding rule is

$$\frac{B; \Gamma, n : \mathsf{Nonce}^{\ell}(B,A) \vdash S : e}{B; \Gamma \vdash \mathsf{new}(n : \mathsf{Nonce}^{\ell}(B,A)).S : e - [\mathsf{fresh}^{\ell}(n)]}$$

Notice that judgements have the form $I; \Gamma \vdash S : e$ meaning that "the sequential process S can be typed according to the typing environment Γ and the effect e when executed by I". So in the example above, $\mathsf{new}(n : \mathsf{Nonce}^{\ell}(B,A)).S$ is typed under $B;\Gamma$ if the continuation process S is typed under the same environment enriched with nonce type $n : \mathsf{Nonce}^{\ell}(B,A)$ and with the effect e possibly containing $\mathsf{fresh}^{\ell}(n)$. (Notice that writing $e - [\mathsf{fresh}^{\ell}(n)]$ in the thesis of the rule allows effect $\mathsf{fresh}^{\ell}(n)$ to be in e without forcing its presence, thus allowing to type-check protocols that declare nonces without checking them.)

The nonce is encrypted together with message M_C in a challenge sent by B to A (effect $!\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$). This atomic effect is derived according to the tagged structure attached to the ciphertext, and omitted here for simplicity. We just assume that tags are such that the message can be recognized to be a challenge from B to A for authenticating message M_C, i.e., the type of the message is $\mathsf{Enc}(\mathsf{Chall}^{\ell}_{n}(B,A,M_C), \_)$. The effect $\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$ conveyed by this type is "transferred" to the continuation process in the form $!\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$, via the following rule:

$$\frac{B; \Gamma \vdash M : \mathsf{Enc}(e_C; e_R) B; \Gamma, z : \mathsf{Un} \vdash S : e + !e_C}{B; \Gamma \vdash \mathsf{encrypt}\ M\ \mathsf{as}\ z.S : e + !e_R}$$

The effect $!\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$ records the fact the challenge has been generated. Similarly, the ciphertext is received and decrypted by the responder A producing an effect $?\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$.

The following $\mathsf{begin}(A,B,M_C;M_R)$ assertion requires a challenge containing the message M_C (effect $?\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$) and justifies a response for authenticating M_R (effect $!\mathsf{Resp}^{\overline{\ell}}_{n}(A,B,M_R)$), as formalized by the corresponding typing rule:

$$\frac{A; \Gamma \vdash S : e + [!\mathsf{Resp}^{\overline{\ell}}_{N}(A,I,M_2)] \Gamma \vdash N : \mathsf{Nonce}^{\ell}(I,A) \Gamma \vdash M_2 : T}{A; \Gamma \vdash \mathsf{begin}(A,I,M_1;M_2).S : e + [?\mathsf{Chall}^{\ell}_{N}(I,A,M_1)]}$$

The relation between ℓ and $\overline{\ell}$ determines how the nonce may be sent out and received back by the originator: the nonce may be sent out in clear and received back encrypted in PC handshakes, thus $\overline{Un} = Ciph$, or vice-versa in CP handshakes, formalized as $\overline{Ciph} = Un$. Finally, in CC handshakes the nonce may be either sent out and received back encrypted ($\overline{Priv} = Priv$) or signed ($\overline{Int} = Int$).

The atomic effect $!\mathsf{Resp}^{\overline{\ell}}_{n}(A,B,M_R)$ enables A to generate a ciphertext with type $\mathsf{Enc}(\_; [\mathsf{Resp}^{\overline{\ell}}_{n}(A,B,M_R)])$. Hence the ciphertext is received and decrypted by B: such a decryption is tracked by $?\mathsf{Resp}^{\overline{\ell}}_{n}(A,B,M_R)$, as done before.

Notice that the effects on B's side describe the generation of a fresh challenge ($!\mathsf{Chall}^{\ell}_{n}(B,A,M_C)$ and $\mathsf{fresh}^{\ell}(n)$) and the reception of a corresponding response

Table 4. Typed PC Protocol in ρ-spi calculus

$$\begin{aligned}
\mathit{Protocol} &\triangleq \mathsf{let}\ k_A = \mathsf{asym\text{-}key}(A)\ .\ (B \triangleright !\mathit{Initiator} \mid A \triangleright !\mathit{Responder})\\
\mathit{Initiator} &\triangleq \mathsf{new}(n : \mathsf{Un}).\mathsf{out}(n).\mathsf{in}(z).\\
&\qquad \mathsf{decrypt}\ z\ \mathsf{as}\ \{|\mathsf{Id}(B), \mathsf{Auth}(x), \mathsf{Verif}_{\mathsf{PC}}(n)|\}_{\mathsf{Pub}(k_A)}.\mathsf{end}(B, A; x).\mathbf{0}\\
\mathit{Responder} &\triangleq \mathsf{in}(x).\mathsf{new}(m : \mathsf{Un}).\mathsf{begin}(A, B; m).\\
&\qquad \mathsf{encrypt}\ \{|\mathsf{Id}(B), \mathsf{Auth}(m), \mathsf{Verif}_{\mathsf{PC}}(x)|\}_{\mathsf{Priv}(k_A)}\ \mathsf{as}\ z.\mathsf{out}(z).\mathbf{0}
\end{aligned}$$

based on the same nonce ($?\mathsf{Resp}_n^{\overline{\ell}}(A, B, M_R)$). This is all we need to correctly conclude the protocol through $\mathsf{end}(B, A, M_C; M_R)$. This is formalized by the typing rule for end:

$$\frac{B; \Gamma \vdash S : e \qquad\qquad \Gamma \vdash n : \mathsf{Nonce}^{\ell}(B, A)}{B; \Gamma \vdash \mathsf{end}(B, A, M_1; M_2).S : e + [!\mathsf{Chall}_n^{\ell}(B, A, M_1), ?\mathsf{Resp}_n^{\overline{\ell}}(A, B, M_2), \mathsf{fresh}^{\ell}(n)]}$$

This rule "consumes" the relative effects thus guaranteeing the injectivity of authentication: every end is matched by a distinct begin.

Notice that the only human effort when typing a protocol is to provide the correct tags and the nonce types. For example, the simple protocol of Table 2 can be type-checked by just adding types declaration inside the primitive new, as shown in Table 4. The nonce type for this example is simply Un, representing the fact that the challenge is sent as cleartext. This fits the general scheme explained above because of the type equality $\mathsf{Nonce}^{\mathsf{Un}}(I, J) = \mathsf{Un}$, i.e., an untrusted value is also an untrusted nonce between every possible parties I and J.

4.3 Safety and Compositionality

Our main result states that if a process can be typed with empty effect and by only assigning type Un to all the identities, then every trace generated by that process is safe.

Theorem 1 (Safety [8,12]). *Let P be a process. If $\overline{I} : \mathsf{Un} \vdash P : []$, where $\overline{I}$ are the identities in P, then P is safe.*

Interestingly, our analysis is strongly compositional, as stated by the following theorem. Let $\mathbf{keys}(k_1, \dots, k_n)$ denote a sequence of key declarations.

Theorem 2 (Strong Compositionality [8,12]). *Let P be the protocol of the form $\mathbf{keys}(k_1, \dots, k_n).(I_1 \triangleright !S_1 | \dots | I_m \triangleright !S_m)$ and $I_1, \dots, I_m$ be the identities in P. Then*

$$I_1, \dots, I_m : \mathsf{Un} \vdash P : []$$

$$\textit{iff}$$

$$I_1, \dots, I_m : \mathsf{Un} \vdash \mathbf{keys}(k_1, \dots, k_n).I_i \triangleright !S_i : [] \qquad \forall i \in [1, m]$$

Intuitively, a protocol is safe if so are all the protocol participants. In addition, judging a participant safe only requires knowledge of the long-term keys it shares with other participants. This is a fairly mild assumption as the information conveyed by the keys is relative to identities of the parties sharing them, not to the protocol they are running. This flexibility has a price, however, in that our result relies critically on the run-time checks on the message tags provided by pattern-matching.

5 Related Work

Tagging is not a new idea and it is proposed and used for verification purposes in [3,4,16,17,22]. Typically, tagging amounts to add a different label to each encrypted protocol message, so that ciphertexts cannot be confused. Our tagging is less demanding, as we do not require that every message is unambiguously tagged since we tag only certain components. In particular, for protocols implemented with stronger tagging techniques, our tags can be safely removed without compromising the protocols' safety.

The Strand Spaces formalism [19,20,21,30] is an interesting framework for studying authentication. There are interesting similarities between our analysis and the way the three kinds of nonce-handshakes are checked in Strand Spaces. It would be interesting to explore how our type system could be applied in such a framework, in order to provide mechanical proofs of safety.

The recent work by Bodei *et al.* on a control-flow analysis for message authentication in *Lysa* [5,6] is also strongly related to our present approach. The motivations and goals, however, are different, since message authentication concerns the origin of a message while agreement provides guarantees about the presence in the current session of the claimant and its willingness to authenticate with the verifier.

Finally, [11] compares our type and effect system with the one by Gordon and Jeffrey, drawing on a translation of tagged protocols, validated by our system, into protocols that type check with Gordon and Jeffrey's system. The paper shows that tags discussed in this paper can be compiled even in the static types of [18]. This allows the tag inference procedure of [15] to be exploited for inferring such types.

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
3. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.
4. B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Proceedings of Foundations of Software Science and Computation Structures*, pages 136–152, 2003.

5. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 126–140. IEEE Computer Society Press, June 2003.
6. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis can find new flaws too. In *Proceedings of the Workshop on Issues on the Theory of Security (WITS'04)*, ENTCS. Elsevier, 2004.
7. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of ICALP 01*, volume 2076, pages 667–681. LNCS 2076, Springer Verlag, 2001.
8. M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication (full version). Submitted for Publication.
9. M. Bugliesi, R. Focardi, and M. Maffei. Principles for entity authentication. In *Proceedings of 5th International Conference Perspectives of System Informatics (PSI 2003)*, volume 2890 of *Lecture Notes in Computer Science*, pages 294–307. Springer-Verlag, July 2003.
10. M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *Proceedings of European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.
11. M. Bugliesi, R. Focardi, and M. Maffei. Analysis of typed-based analyses of authentication protocols. In *Proceedings of 18th IEEE Computer Security Foundations Workshop (CSFW 2005)*. IEEE Press, 2005. To appear.
12. M. Bugliesi, R.Focardi, and M.Maffei. Authenticity by tagging and typing. In *2nd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE 2004), ISBN 1-58113-971-3*, pages 1–12. ACM press, October 2004.
13. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz, November 1997.
14. R. Focardi, R. Gorrieri, and F. Martinelli. Non interference for the analysis of cryptographic protocols. In *Proceedings of ICALP'00*, pages 354–372. Springer LNCS 1853, July 2000.
15. R. Focardi, M. Maffei, and F. Placella. Inferring authentication tags. In *Proceedings of IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues on the Theory of Security (WITS 2005)*. ACM Digital Library, January 2005.
16. A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 145–159. IEEE Computer Society Press, June 2001.
17. A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proceedings of 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 77–91. IEEE Computer Society Press, 24-26 June 2002.
18. A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2004.
19. J. D. Guttman and F. J. Thayer Fàbrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.
20. J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: a rely-guarantee method. In *Proceedings of European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*, pages 325–339. Springer-Verlag, 2004.
21. Joshua D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. In *Proceedings of 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 24–34. IEEE Computer Society Press, July 2000.

22. J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 255–268. IEEE Computer Society Press, July 2000.
23. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.
24. G. Lowe. "A Hierarchy of Authentication Specification". In *Proceedings of the 10th Computer Security Foundation Workshop (CSFW'97)*, pages 31–44. IEEE Computer Society Press, 1997.
25. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.
26. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.
27. R. M. Needham and M. D. Schroeder. Authentication revisited. *ACM SIGOPS Operating Systems Review*, 21(1):7–7, 1987.
28. L. C. Paulson. Relations between secrets: Two formal analyses of the yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.
29. Douglas R. Stinson. *Cryptography, Theory and Practice.* CRC Press, 1995.
30. J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. Journal of Computer Security, 1999. 7(2/3).
31. T.Y.C. Woo and S.S. Lam. "A Semantic Model for Authentication Protocols". In *Proceedings of 1993 IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

Formal Methods for Smartcard Security

Gilles Barthe[1] and Guillaume Dufay[2]

[1] INRIA Sophia-Antipolis, France
Gilles.Barthe@inria.fr
[2] SITE, University of Ottawa, Canada
gdufay@site.uottawa.ca

Abstract. Smartcards are trusted personal devices designed to store and process confidential data, and to act as secure tokens for providing access to applications and services. Smartcards are widely deployed and their usage spans over several application domains including banking, telecommunications, and identity.

Open platform smartcards are new generation trusted personal devices with increased flexibility. Such devices, which benefit of increased connectivity and increased interoperability, can host several applets and allow new applets to be loaded post-issuance. Such an increased flexibility raises concerns about the possibility of logical attacks that could affect a very large number of devices, and requires the development of techniques and tools that can be used to increase the reliability of platforms and applications for trusted personal devices. The objective of this chapter is to describe some applications of formal methods to increase the reliability of smartcards and trusted personal devices.

1 Introduction

Smart cards are trusted personal devices whose characteristics are regulated by the ISO 7816 standard. As other trusted personal devices, smartcards are designed to store and process confidential data, and can act as tokens to provide users with a secure electronic representation in a large network. They are widely deployed and used in application areas such as mobile telecommunications, banking, transportation, electronic identity, and digital rights management (DRM). Further, they hold the promise to play a key role in the e-society, especially as a means to guarantee users a personalized, global, and secure access to applications and services.

The prominent role played by trusted personal devices in security sensitive applications make them an ideal target for attacks. Traditionally, the main concern with smartcards has been with hardware attacks in which the attacker gains access to confidential information or disturbs the functioning of the card through observation (e.g. of power or electro-magnetic radiations) or invasion (e.g. overriding sensors or attaching probes).

With new generation smartcards and trusted personal devices increasingly connected to networks and providing execution support for complex programs, the prospect of logical attacks has urged the trusted personal devices industry to improve the quality of their software, as logical attacks are potentially easier to launch than physical attacks (for example they do not require physical access to the device, and are easier to replicate from one device to the other), and may have a much wider impact. In particular, a malicious attacker spreading over the network and disconnecting or disrupting devices massively could have devastating economic and social consequences and would deeply affect end users confidence in e-society. The Cabir virus which spread through Symbian cell phones during summer 2004, although it did not actually do any damage, sounded a strong warning that cell phone viruses may soon cause havoc if no appropriate security technology is developed. With the increasing use of voice over IP, the nightmare could also extend to all phone infrastructures.

The risk of devastating attacks on trusted personal devices justifies the development of methodologies and tools that increase confidence in the execution platforms that they support and in applications that are executed on-board such devices. The need for methodologies and tools is implicitly recognized by existing standards for evaluating security-sensitive IT products, such as the Common Criteria [31] which require the use of formal methods at its highest and most demanding levels EAL5-EAL7, and has triggered some substantial activity in the community of formal methods. Much activity has centered around establishing the correctness of execution platforms for smartcards, and showing that applications are innocuous.

The purpose of this chapter is to motivate and illustrate applications of formal methods to increase dependability of trusted personal devices, both with respect to platform correctness and applet validation. For concreteness, we focus on devices that embed Java Virtual Machines (JVM) or their variants, in particular Java Card Virtual Machines (JCVM). Java enabled devices are a natural choice for formal methods because: i) they are widely deployed in the field; ii) they feature mechanisms that contribute to the security of the platform and the applications that execute over it; iii) detailed informal specifications of the Java platform are publicly available, and can be scrutinized. However, it should be clear that the methods presented in this paper are relevant to other execution platforms for trusted personal devices.

The remaining of this chapter is organized as follows: Section 2 begins with a brief introduction to smartcards, then continues with an overview of the JavaCard platform and a description of the software security mechanisms that it provides. Section 3 addresses the issue of platform correctness, whereas Section 4 is dedicated to application validation. We conclude in Section 5 with some perspective on emerging trends and directions for further work.

2 A Primer on Smartcards

Smartcards are a prime example of trusted personal devices in the sense that smartcards belong to a single person and are used to enable trusted operations in an information technology and communication infrastructure. Other examples of trusted personal devices include dongles for protected softwares, and under a liberal interpretation of trusted personal devices, cell phones and other smart objects such as PDAs.

The purpose of this section is to provide a brief introduction to smartcards, starting from their characteristics and applications, pursuing with a description of the standard architecture for the current generation of smartcards, and concluding with security issues and mechanisms in smartcards.

2.1 Characteristics and Applications

Smartcards consist of a memory and a microprocessor, with special security functions, usually embedded within a credit-card sized plastic card. Depending of their type (contact or contactless cards), these cards require either a card reader, a.k.a. Card Acceptance Device or CAD for short, or a radio frequency signal for being powered. For interoperability, the standards ISO 7816 define the position of the chip on the card, the physical constraints for the connectors and the communication protocols between the chip and the reader. The CPU of the card is usually an 8 or 32-bits CISC microprocessor running at 5 MHz or more. It relies for execution on different types of memory:

– a ROM, up to 32 kB, that stores the operating system;
– an EEPROM (erasable but slow memory), up to 32 kB, that stores permanent data of the card;
– a RAM that is usually only 256 bytes in size.

The operating system on the card is responsible for communication protocols, internal memory management as well as a filesystem on the EEPROM. This filesystem organizes the memory in files and folders that correspond to the various applications loaded at the same time on the card. Accesses to the filesystem content is read/write controlled and each directory (i.e. application) can follow specific security policies so as to prevent information sharing with other applications on the card. For communication with the terminal, the operating system on the card follows the Application Protocol Data Unit (APDU) protocol. Usually, only the terminal controls the communication. However applications on the card can also use APDUs for communications with the operating system and in some cases, the card itself can initiate the communication with the terminal.

Smartcards are widely used in telecommunications (SIM cards for GSM phones and UICC cards for 3G phones), financial services (banking cards), identification (electronic ID), e-administration (digital signature), multimedia (pay-TV), transportation (contact smartcards for parkings and tolls, and contactless smartcards for public transportation) and health (electronic health records).

2.2 Open Platform Smartcards

Open platform smartcards correspond to new generation smartcards with increased flexibility. Such smartcards:

- integrate with their operating system a virtual machine that abstracts away from any hardware and operating system specifics so that smartcard applications, or *applets*, can be programmed in a high-level language;
- are multi-applications, in that several applets can coexist and communicate on the same card, and support post-issuance, in that new applets may be loaded onto already deployed smartcards.

Java Card. Launched in 1996 by Sun, Java Card [53] is a dialect of Java adapted to the resource constraints of smartcards, and the standard programming language for smartcard applications. Java Card applets are written in a subset of the Java language, and using the Java Card API. They are then compiled down to class files, and then converted to the CAP file format; conversion involves a number of optimizations suggested by smartcard constraints, e.g. names are replaced by tokens and class files are merged together on a package basis. Finally, CAP files are loaded and installed on the card, where they can be executed by the Java Card Runtime Environment JCRE. The JCRE contains the Java Card Virtual Machine JCVM, provides support for the Java Card API and possibly for domain-specific API such as the GSM API for mobile phones, and invokes the services of a native layer to perform low-level tasks such as communication management. This Java Card architecture is summarized in Figure 1.

There are of course many similarities between Java Card and Java: in particular, Java Card programs are Java programs written in a fragment of the language, and the JCVM is a stack-based virtual machine that is closely related to the Java Virtual Machine JVM. There are also a number of differences: Java Card programs are not allowed to use large data types such as floats and strings, arrays of arrays, or finalization. Besides, some features of the JVM like dynamic class loading mechanism and multi-threading are unsupported by the

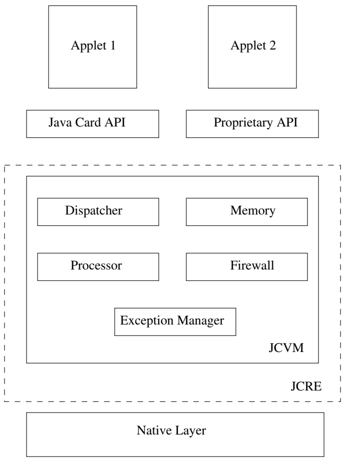


Fig. 1. Java Card architecture

JCVM (but the current version of Java Card features logical channels instead of multi-threading). Furthermore, the Java API and the Java Card API offer different functionalities, or in some cases such as remote method invocation different variants of the same functionality.

Memory management in Java Card also differs from that of Java in several aspects: in particular, garbage collection is optional and can only be performed upon completion of the execution of an applet through an explicit call to the Java Card API. Furthermore, Java Card offers a transaction mechanism for atomicity, since smart cards do not include a power supply, and thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification prescribes the use of a transaction mechanism to control synchronized updates of sensitive data. A statement block surrounded by the methods `beginTransaction` and `commitTransaction` can be considered atomic. If the transaction cannot be completed due to a card tearing or a power loss or a call to `abortTransaction`, the card will roll back its internal state to the state before the transaction was begun. If the card is unpowered, this event will occur as soon as the card is reinserted into a terminal.

Nevertheless, the essential difference from the point of view of this chapter resides in security mechanisms: indeed, Java Card abandons the Java stack inspection mechanism in favor of a simpler firewall mechanism that ensures applet isolation and that is mitigated by provisions to allow controlled communication between applets. The mechanism is discussed in the next paragraph. Further technical information about the current version of Java Card (Java Card 2.2) may be found in specifications and white papers available from Sun web site. It is likely however that Java Card will undergo a major evolution of the language and that future versions of Java Card (or successors to Java Card) will provide support for increased connectivity, and for multi-threading.

We conclude this section by mentioning that while Java Card is a central focus of this chapter, there exist other operating systems for open platform smartcards, in particular Multos, and .Net Card, as well as other dialects of Java for trusted personal devices, in particular MIDP for mobile information devices such as cell phones and PDAs, MHP for home multimedia, STIP for electronic transactions.

Global Platform. Java Card does not provide standardized mechanisms for managing applications on the card. In order to benefit from such mechanisms, most Java Cards also implement Global Platform [80], which provides a card management architecture for multi-application smartcards with post-issuance facilities (and is independent of the runtime environment).

The Global Platform architecture is detailed in two distinct specifications that respectively describe the card functional requirements and its associated security requirements. While the security of the smartcard strongly depends on a correct design and implementation of Global Platform, there has been little use of formal methods to analyze Global Platform; one notable exception is the (as yet unpublished) work of S. Zanella Béguelin, who provides a B model of the specifications, and highlights a number of spots in the specifications where clarifications are required. The model is available from [80].

2.3 Java Card Security

The flexibility of open platform smartcards is at the same time a major asset and a major obstacle to their deployment. On the one hand, writing applets in a high-level language reduces the cost and time to market new applications, and the possibility of running several applets on a single card opens the way for novel applications. On the other hand, such smartcards introduce the possibility to load malicious applets that exploit weaknesses of the platform, make an improper use of the API, or simply launch denial-of-service attacks through immoderate resource consumption. Increased connectivity and post-issuance application loading constitute further complications from the point of view of security and contribute to making large-scale logical attacks a frightening and likely perspective.

Security Mechanisms. Current security architectures for smartcards feature two central mechanisms, a.k.a. security functions, to prevent logical attacks: the firewall, and the bytecode verifier.

Firewall. In the Java Card environment, the security model for partition between applications and between operating system and applications differs from the sandboxing model of Java. In the firewall model of Java Card, an unique context (Applet IDentifier or AID) is associated to each applet on the card. Only one context can be active, the one from the current applet. The JCRE prevents any access from one object to another object with another context, with the exceptions of static variables, arrays declared as global or instances declared as entry points. In the other cases, objects must communicate exchanging objects implementing the `javacard.framework.Shareable` interface. This procedure remains controlled by the JCRE and is the following:

– The server applet must define an interface `SI` extending the `Shareable` interface, a class `C` implementing `SI` and create an object `O` of class `C`;
– To access the object `O` from applet `A`, the client applet `B` must invoke the `JCSystem.getAppletShareableInterface` method;
– The JCRE uses the `getShareableInterfaceObject` method to send a request to Applet `A`. Then Applet `A` determines, given the AID of `B`, if `B` is authorized to access `O` and if so, returns a reference to `O`;
– The applet `B` casts `O` to type `SI` into a class reference `SIO`;
– The firewall prevents the applet `B` from accessing any field or method not defined in `SI`.

An example of such a procedure is given in Figure 2, where the applet `Bob` want to share an object belonging to the applet `Alice`:

```
public interface SI extends Shareable {
  Secret foo(); }

public class Alice extends Applet implements SI {
  private Secret ObjectSecret;

  public Shareable getShareableInterfaceObject(AID Client) {
    if (Client.equals(BobAID))
      return(this);
    return null; }

  public Secret foo() {
    AID Client;
    Secret Response;
    Client = JCSystem.getPreviousContextAID();
    if (Client.equals(BobAID))
      Response = ObjectSecret;
    return Response; } }

public class Bob extends Applet {
  public static SI AliceObj;
  private static Secret AliceSecret;

  public void bar() {
    AliceObj = (SI) JCSystem.getAppletShareableInterface
                       (AliceAID);
    AliceSecret = AliceObj.foo(); } }
```

Fig. 2. Example of the use a `Shareable` object

Bytecode Verification

GOALS. The bytecode verifier is a key security function in the Java Card architecture. Its purpose is to check that applets are correctly formed and correctly typed, and that they do not attempt to perform malicious operations during their execution. It consists on a two steps process. The first one, and the simplest, is a structural analysis of the consistency of the CAP file and its constant pool. The second one requires a static analysis of the program and is meant to ensure that:

– values are used with their correct type (to avoid forged pointers) and method signatures are respected;
– no frame stack or operand stack underflow or overflow will occur;
– visibility of methods (**`private`**, **`public`**, or **`protected`**) is compatible with their use;
– objects and local variables are initialized before being accessed. Together with subroutines, which are discussed below, initialization is one of the main

difficulties from the point of view of bytecode verification, as illustrated e.g. by S. Freund and J. Mitchell [47];
– jumps in the program code remain in legal bounds.

Ensuring such properties is an important step towards guaranteeing security, and the failure to enforce any of these properties may be exploited for launching attacks.

ALGORITHMS. Bytecode verification [63] is a data-flow analysis of a typed virtual machine which operates on the same principles that the standard JVM except for two crucial differences: the typed virtual machine manipulates types instead of values, and executes one method at the time.

The data-flow analysis aims at computing solutions of data-flow equations over a lattice derived from the subtyping relation between JVM types, and uses to this end a generic algorithm due to G. Kildall [57]. In a nutshell, the algorithm manipulates so-called stackmaps that store for each program point an history structure that represents the program states that have been previously reached at this program point. The history structure is initialized to the initial state of the method being verified for the first program point, and to a default state for the other program points. One step of execution proceeds by iterating the execution function of the virtual machine over the states of the history structure. Each non-default state is chosen once and the result of the execution of the typed virtual machine on this state is propagated to its possible successors in the history structure.

Different history structures can be used depending on the accuracy required from the analysis.

– In a monovariant analysis, the history structure stores one program state, which is the least upper bound of the states that have been been previously computed at this program point. In such an analysis, propagating a state in an history structure amounts to taking pointwise the least upper bound (on the type lattice of the virtual machine) of the types appearing in the two states and storing the result back at this location. The termination of the analysis is guaranteed since the set of states does not have an infinite ascending chain, and the state stored in the history structure is increasing.
  As noted by R. Stata and M. Abadi [93], collapsing history structures to a single state as done in the monovariant analysis leads to a bytecode verification algorithm that does not handle subroutines as prescribed by the informal specifications of Sun. To be more precise, monovariant bytecode verification rejects bytecode programs that make a polymorphic use of subroutines. This use of subroutines can lead to two states, for a same program point, that do not have the same number of local variables or the same number of elements in the operand stack and that would then be merged state into an error state, although the execution is valid.
– In a polyvariant analysis, the history structure stores the set of program states that have been previously computed at this program point. In such an analysis, propagating a state in an history structure amounts to adding

the newly computed state to the history structure. The termination of the analysis is guaranteed since the set of states is finite, and the size of the history structure is increasing.

Polyvariant bytecode verification provides an accurate treatment of subroutines, and was introduced independently by P. Brisset (in unpublished work) and by A. Coglio [27]. L. Henrio and B. Serpette [49] propose an improvement of polyvariant bytecode verification in which compatible states in the history structure can be merged so as to keep the size of history structures reasonable. It is interesting that approaching bytecode verification through model-checking [81,10] results in an analysis which capture a similar class of programs as polyvariant bytecode verification.

We refer the reader to [63] for a more detailed account of algorithms for bytecode verification.

On-Device Verification. Currently applets are verified off-card and, in case of a successful verification, signed and loaded on-card. Such a solution is not optimal in the sense that it leaves a crucial component of the security architecture outside of the perimeter of the smartcard. However, there are several proposals for circumscribing the trusted computing base to the smartcard using on-card bytecode verification. One solution adopted in the KVM [29] is to rely on lightweight bytecode verification, initially proposed by E. Rose, in which the program comes equipped with the solution to the dataflow equations, and the role of the verifier is to check that the solutions are correct. Another proposal by X. Leroy [62] is to perform an off-card transformation that allows bytecode verification to be performed in one pass. A later work by D. Deville and G. Grimaud [34] does not require programs to be rewritten or annotated, but exploits instead efficient encodings of the data structures manipulated by the bytecode verifier.

Security Issues. The Java Card security architecture guarantees that downloaded applications are innocuous and comply with some basic policies related to typing, initialization or access control. Such basic policies are the cornerstones upon which the overall security of the smartcard will rely. Therefore it is important to verify that the security architecture does enforce these basic policies as intended. Thus, an important application of formal methods to smartcard security is platform verification, which aims at providing an abstract model of the Java Card platform and security architecture, and at proving that the security functions play their expected role. However, it is not sufficient to show that security functions are correctly designed. In particular, one also has to ensure that other components of the infrastructure are correctly designed: the Java Card API and the Global Platform API constitute two prominent components of the infrastructure whose correct design is central to security. Thus, another important aspect of platform verification is to show the API are correctly designed.

Platform verification is a fundamental step towards guaranteeing the security of smartcards, and a prerequisite for Common Criteria evaluations at the highest levels. Nevertheless, the guarantees offered by the Java Card security architecture are limited, and further verifications must be performed to verify that applications make a legitimate use of the infrastructure, and do not attempt any hostile action. Thus, application validation is another important application of formal methods to smartcard security. There are many facets to applet validation, each with its own objectives and techniques. For example, applet validation may be performed at bytecode level prior to loading an application on card. Another scenario is that applet validation is performed at source level by developers or experts in formal methods working with developers (this is an ideal situation, often formal methods are used at posteriori). While such a scenario is not ideal from the perspective of guaranteeing the security of a smartcard, the smartcard industry has found such a scenario particularly useful in particular for checking that applets respect some given security requirements.

In summary, security is a holistic property of a system, and formal methods must therefore be employed at many different levels to provide strong guarantees about smartcard correctness. Platform verification and application validation are two important aspects of guaranteeing security for smartcards, and the focus of the next sections. Other important aspects of formal methods which are not treated in this chapter include the use of formal methods to establish a relation between the models developed for platform verification and the actual implementations of the platform, see e.g. [23], or the use of formal methods to verify cryptographic algorithms or protocols used by smartcards, see e.g. [67].

3 Platform Certification

Dedicated operating systems for smartcards aims at providing a secure environment for applications execution. For this purpose, special security features are provided, and precise specifications on the platform are given. However, due to the size of these specifications, the use of formal methods is required to ensure to the whole specifications and the corresponding implementation are correct. For instance, once the entire has been formalized, it is possible to give the statement of global properties, such as type correctness or applets isolation, and prove that the platform do not contain design flaws or implementation bugs for these properties. Although the approach was different at the time, a type-system related security hole was indeed found in the bytecode verifier of Java 1.1 by the Kimera project.

Several formalizations of the Java (Card) platform are now available. They differ by the coverage of the runtime environment and virtual machine, the formalism they are built upon, the style of semantics used for the instruction set and the particular aspect of security they aim at verifying. The Isabelle/HOL formalizations of T. Nipkow and co-workers [58] and the executable specifications of the J-book [92] constitute some of the most impressive achievements in this direction to date. The reader may refer to [48] for a not so up-to-date survey of the various formalizations. In the following, we will focus on a another formalization [38] not yet available at the time of the survey. This formalization, written within the Coq proof assistant, covers almost all the aspects of the Java Card platform, has not be written for a specific security property and thus remains general purpose. Besides, it is executable (we consider executability as an essential point to remain as close as possible of a reference virtual machine) and it comes with a tool to resolve constant pool of CAP files and translate them, including the ones with native methods, into the representation of programs given below.

3.1 Formalisms

The following formalization of the Java Card platform has been written in the Coq [30] specification language. Nevertheless, the formalization is easily translatable to other programming language such as CAML, and other proof assistants such as Isabelle and PVS, since it does not use any high level feature of Coq. We will describe in the following the corresponding subset of the specification language.

The keyword **`Inductive`** (resp. **`Mutual Inductive`**) introduces a inductive (resp. mutual inductive) type. Such definitions also declare all constructors for these types. We give below the definitions of the natural numbers `nat` (with the constructors `O` and `succ`, for successor), of parameterized (polymorphic) lists `list`, and of the parameterized `option` type commonly used to formalize non total functions (lift monad):

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.

Inductive list (A:Set) : Set :=
| Nil : list A
| Cons : A → list A → list A.

Inductive option (A:Set) : Set :=
| None : option A
| Some : A → option A.
```

The partial functions `head`, `tail` respectively return for a given list the first element, and the list without its first element. Our formalization uses the type `list` in many places as the type of ordered sets, for instances to represent stacks or arrays. This allows to use all predefined accessors functions and lemmas of Coq and to get a directly executable semantics. However, with the new module system of Coq, it would also be possible to declare abstract modules for basic datatypes and give later an executable implementation of these modules.

Records are introduced by keyword **`Record`**. Accessors of these fields are then expressed as functions named by the corresponding fields of the record. The construction `match...`**`with`**`...`**`end`** introduces pattern matching as in ML languages.

3.2 Virtual Machine

The virtual machine described in the following is a defensive virtual machine, i.e. it follows the specification of the reference virtual machine by explicitly implementing all the pre-conditions (the *must* and *must not* clauses) stated by the official specification. This virtual machine is based on Java Card version 2.1, supports the full instruction set of the specification, includes the firewall, but lacks some other features of the Runtime Environment that rely on native APIs (such as APDUs or transactions), although a methodology to include these APIs is given.

Java Card Programs. In this section, we outline the Coq formalization of Java Card bytecode programs, as executed by the virtual machine. The reader may refer to [38] for a complete description of this formalization.

Type system. The Java Card Virtual Machine distinguishes between primitive types and reference types. Each of these types is subdivided into atomic types. For instance, there are four kinds of reference types that are null types, array types, instance types and interface types. Arrays are described according to the type of their elements which must be primitive or reference. The reference of an instance or an interface corresponds to an index in the set of classes or interfaces of the program. Finally, the type of Java Card types is given by the following mutual inductive definition that captures the considerations below:

```
Inductive vmtype : Set :=
  | Prim : vmtype_prim → vmtype
  | Ref : vmtype_ref → vmtype
with vmtype_ref : Set :=
  | Ref_null : vmtype_ref
  | Ref_array : vmtype → vmtype_ref
  | Ref_instance : class_idx → vmtype_ref
  | Ref_interface : interf_idx → vmtype_ref.
```

where `vmtype_prim` is a simple inductive type that gathers all primitive JCVM types.

We notice that the constructor for arrays, `Ref_array` allows to form types corresponding to arrays of arrays, which is not permitted in Java Card. However our formalization of the operational semantics of the JCVM, and in particular the implementation of the `anewarray` instruction, does not allow to form such a type.

Programs. Our virtual machine does not work directly on the binary representation of the CAP file format but on structured representation of programs, obtained by a dedicated tool. After this transformation, that resolves the constant pool of the program, only the essential components of a typed object-oriented language remain: types, methods, classes and interfaces. Formally, a program is described in Coq by the following record type:

```
Record jcprogram := {
  interfaces : (list Interface);
  classes    : (list Class);
  methods    : (list Method);
  sheap_type : (list type)
}.
```

where the types `Interface`, `Class`, `Method` and `type` are themselves defined as record types. `sheap_type` is used for initialization purposes, to determine types of variables declared as static in the program. For simplicity, we only deal with closed programs hence the packages `java.lang` and `javacard.framework` are an integral part of programs.

Methods. A method is characterized by its status (static or not), its signature (against which one can type-check its arguments upon invocation and returned value upon completion of the method), the number of its local variables (for initializing its execution context), its list of instructions to be executed (the bytecode), its exception handlers (a handler is an object that identifies the code for managing dynamic errors), its maximum operand stack size and finally its owning class and the indexes of the method. Formally, we use the following structure to represent methods:

```
Record Method : Set := {
  is_static          : bool;
  signature          : signature_type;
  nb_local           : nat;
  bytecode           : (list Instruction);
  handlers           : (list handler_type);
  m_max_opstack_size : nat;
  owner              : class_idx;
  method_id          : method_idx
}.
```

where `class_idx` and `method_idx` corresponds respectively to the type of indexes to a class and to a method. Others components of a program, such as interfaces and classes, are represented in a similar fashion.

The type `Instruction` enumerates with an inductive type the different bytecodes of the Java Card language, with theirs operands:

```
Inductive Instruction : Set :=
  | nop           : Instruction
  | push          : vmtype_prim → Z → Instruction
  | ret           : locvars_idx → Instruction
  | invokespecial : nat → method_idx → Instruction
  | invokestatic  : nat → method_idx → Instruction
  | getfield      : vmtype → instance_field_idx → Instruction
  | inc           : vmtype_prim → Z → nat → Instruction
...
```

where `Z` is the type of binary integers. Some bytecodes with similar semantics have been collapsed into a single one. For instance, the bytecode `inc` takes as a supplementary argument a primitive type and thus represents the bytecodes `iinc` et `sinc` of Java Card (incrementation of a register of type `Int` and `Short` respectively). It is possible to represent the 185 Java Card bytecodes by only 44 bytecodes in our formalization. Also, for convenience, slight differences may appear in the operands of instructions, such as for `invokestatic` and `invokespecial` that receive an extra parameter corresponding to the number of arguments of the method.

Each instruction of the virtual machine is given by a small-step semantics; more precisely, each instruction is formalized as a state transformer, i.e. a function that takes as input a state (before the instruction is executed) and returns a new state (after the instruction has been executed).

Memory Model. Java (Card) virtual machine is stack-based. For intermediary computations, values are pushed and popped from an operand stack. Also, execution contexts for methods, called frames, are organized in stack, each new invoked method being pushed in top of the stack, and thus becoming the active frame. In our formalization, the state contains all the dynamic items manipulated by a Java Card program: values, an heap (for created objects) and a stack of frames.

Values. Values constitute the main element manipulated by the virtual machine. In a defensive virtual machine, values are typed and follow a definition similar to the type system, with an inductive type, where we distinguish primitive and reference values. Type information is carried by the constructor and the numerical value by the argument of the constructor. We obtain for reference values:

```
Inductive d_val_prim :=
  | d_ReturnAddress : bytecode_idx → d_val_prim
  | d_Void          : Z → d_val_prim
  | d_Boolean       : Z → d_val_prim
  | d_Byte          : Z → d_val_prim
  | d_Short         : Z → d_val_prim
  | d_Int           : Z → d_val_prim.

Inductive d_val_ref : Set :=
  | d_Ref_null : d_val_ref
  | d_Ref_array : vmtype → heap_idx → d_val_ref
  | d_Ref_instance : class_idx → heap_idx → d_val_ref.
```

where `heap_idx` indicates the location of an object in memory and `class_idx` is an index into the program classes. There is no value corresponding to interfaces, since an interface must be implemented by a class to be used, through an instance.

Finally, the Coq type of Java Card values is defined by:

```
Inductive d_val : Set :=
  | d_Prim : d_val_prim → d_val
  | d_Ref : d_val_ref → d_val
  | d_NonInit : class_idx → bytecode_idx → heap_idx → d_val.
```

where `d_NonInit` is a specific value of non-initialized class instances.

Objects. Objects in memory constitutes the runtime representation of classes and arrays. The `d_Ref_instance` and `d_Ref_array` values defined in the previous section refer to these objects. We collect in a record information needed for each of these types.

We define an inductive type for objects:

```
Inductive obj : Set :=
  | Instance : type_instance → obj
  | Array    : type_array → obj.
```

where `type_instance` and `type_array` are record types containing all dynamic information related to the object, such as values of the fields of instances or values of arrays. Finally, the memory area in which objects are stored during the execution of a program, the heap, is naturally defined as:

```
Definition heap := (list obj).
```

Frames and Stack. Frames contains computational information for methods: an operand stack and a set of local variables (or registers). An index records the referring method, and a program counter points to the next instruction to execute within the method. Finally, the maximum operand stack size is duplicated within the frame to avoid frequent lookup in the method referenced in `method_loc`.

```
Record d_frame : Set := {
  d_opstack          : (list d_val);
  d_locvars          : (list (option d_val));
  d_method_loc       : method_idx;
  d_p_count          : bytecode_idx;
  d_context_ref      : AID;
  d_max_opstack_size : nat
}.
```

where `AID` is the type for an unique identifier of applets (for the firewall mechanism). The type `option` of the field `locvars` will be used to formalize non initialized local variables.

States. The notion of states is defined by the following record:

```
Record d_state : Set := {
  d_sfields_f : d_sfields;
  d_heap_f  : d_heap;
  d_stack_f : (stack d_frame)
}.
```

where `d_sfields` is the type for the static fields (a list of `d_val`, in which will be stored variables declared as static and shared by the whole program). Most of the instructions will act on the top frame, i.e. the head of the stack. Other fields might be added to the notion of states such as input/output buffers for APDUs (see Section 2.1).

Finally, states are used for the construction of return states of instructions. Instruction can progress normally, provoke an error or throw an exception:

```
Inductive d_rstate : Set :=
  | d_Normal         : d_state → d_rstate
  | d_Abnormal       : eLabel → d_state → d_rstate
  | d_ThrowException : xLabel → d_state → d_rstate.
```

where `eLabel` and `xLabel` are inductive types that record the reason of the error or the exception respectively.

Semantics of Instructions. Bytecode instructions update the memory of the JCVM according to operands generated at compile time and to the representation of programs given below. Thus, the main execution function of the virtual machine, that determines which is the next instruction to execute and calls the corresponding function in our formalization, has the following signature:
`d_exec : d_state →d_rstate`
Then, most instructions have a similar execution pattern:

1. the initial state is decomposed;
2. fragments of the state are retrieved;
3. observations are made to determine the new state;
4. the final state is built on the basis of the retrieved fragments and of the observations made.

We illustrate this pattern on the bytecode `ifnull` that compares the first element of the operand stack, that must be a reference value, to null and branches accordingly to the program counter given as a parameter or to the next instruction. The defensive semantics of this bytecode (defensive in the sense it enforces type verification as mentioned in Sun specification and may return a `type_error`) is given by the following `d_ifnull` Coq function:

```
Definition d_ifnull (b : bytecode_idx) (s : d_state) :=
  match d_stack_f s with
  | Nil ⇒ d_Abnormal state_error
  | Cons h lf ⇒
      match head (d_opstack h) with
      | Some v ⇒
          match v with
          | (d_Ref _) ⇒
              match d_res_null v with
              | True ⇒
                 d_update_frame (d_update_pc b
                                  (d_update_opstack
                                    (tail (d_opstack h)) h)) s
              | False ⇒
                 d_update_frame (d_update_pc (succ (d_pc h))
                                  (d_update_opstack
                                    (tail (d_opstack h)) h)) s
              end
          | _ ⇒ d_Abnormal type_error
          end
      | None ⇒ d_Abnormal opstack_error
      end
  end.
```

where the `d_res_null` checks for null pointers and `d_update_frame` function takes as a parameter the current updated frame h, a state s and returns a normal `rstate` which is an update of s where the topmost frame of the execution stack has been replaced by h.

Other security checks are illustrated within the following excerpts of the `invokevirtual` semantics. The function `new_frame_invokevirtual` is used once the initial state has been decomposed. Formally, we set:

```
Definition new_frame_invokevirtual (nargs : nat) (m : Method)
 (nhp : obj) (h : d_frame) (st : d_state) (cap : jcprogram) :=
 (* Extraction and removal of the arguments *)
match l_take nargs (d_opstack h), l_drop nargs (d_opstack h)with
  | Some l, Some l' ⇒
    (* Security check *)
    if test_security_invokevirtual h nhp
    then
      (* Signature check *)
        if (signature_verification l (signature m) cap)
        (* Updates the current frame and pushes the new frame *)
        then d_Normal ...
        else d_AbortCode signature_error st
    else d_ThrowException Security st
  | _, _ ⇒ d_AbortCode opstack_error st
  end.
```

where `h` is the current topmost frame, `obj` is the object on which the method `m` should be invoked and the ellipsis stands for the resulting built state (omitted in the excerpt). The function performs various checks, such as the verification of the arguments of the method w.r.t its signature and the firewall mechanism with `test_security_invokevirtual`, that may throws a `SecurityException`. For example, in case the object `nhp` is an instance, the function will verify whether (1) the active context is the Java Card Runtime Environment context or; (2) the active context is also the context of the instance owner or; (3) the instance is an entry point. If not, the function returns `true` to flag a security violation. Formally:

```
Definition test_security_invokevirtual (h : d_frame)(nhp: obj):=
  (* Tests if the active context is the JCRE *)
  if eqb_AID (context_ref h) jcre_AID
  then true
  else
   match nhp with
     (* The object is an instance *)
     | Instance ti ⇒
       (* Checks for equal contexts or entry point *)
       orb (eqb_AID (context_ref h) (owner_i ti))  (ptE ti)

     (* The object is an array *)
     | Array ta ⇒
       (* Checks for equal contexts or global array *)
```

```
      if eqb_AID (context_ref h) (owner_a ta)
      then true
      else eqb (statusglobal ta) is_global
  end.
```

3.3 Javacard API

In addition to the Java Card Virtual Machine, the Java Card Runtime Environment includes an implementation of the Java Card APIs. This implementation is required to execute those Java Card programs that appeal to the APIs.

In order to obtain a complete Java Card Runtime Environment and to be able to execute or reason about any Java Card program, we must therefore formalize the APIs in Coq. For the APIs that rely on standard Java Card code, the modeling is direct since we can represent them as any Java Card program.

However, special care is required to deal with the APIs that deal with native methods whose code is not written in Java. For such methods, we need to provide an implementation in Coq. In our formalization, we give the implementation of some native methods from the APIs, such as the method `arrayCopy` which appears in the example of Section 4.2 and is used to copy an array. Examples of APIs that are not treated that are not treated by our formalization include for instance the APIs for communication with APDUs, required for instance to select and execute a particular applet. Formalizing these APDUs could be achieved by adding into our definition of a state two fields corresponding to an input and an output buffer [91].

With the complete implementation of the APIs, the starting point of our formalization for executing a program could then be the `main` method from the `Dispatcher` class of the `javacard.framework` package, i.e. the standard starting point of a Java Card smart card after any reset.

3.4 Verified Java(Card) Bytecode Verifiers

In this section, we present the general methodology that we used for verifying bytecode verifiers [7]. In addition to the defensive virtual machine (from Section 3.2), the methodology involves a typed virtual machine, and an offensive virtual machine.

Typed and Offensive Virtual Machine. The bytecode verifier is built on a variant of the defensive virtual machine, called typed (abstract) virtual machine, that uses types as values and thus only performs type verifications. Corresponding datatypes for values are given below (and prefixed with `a_`) and extended to the notion of typed states `a_state` and typed return states `a_rstate`:

```
Inductive a_val_prim :=
| a_ReturnAddress : bytecode_idx → a_val_prim
| a_Void          : a_val_prim
| a_Boolean       : a_val_prim
| a_Byte          : a_val_prim
| a_Short         : a_val_prim
| a_Int           : a_val_prim.
```

```
Mutual Inductive a_val :=
| a_Prim           : a_val_prim → a_val
| a_Ref            : a_val_ref → a_val
with a_val_ref :=
| a_Ref_null       : a_val_ref
| a_Ref_array      : vmtype → a_val_ref
| a_Ref_instance   : class_idx → a_val_ref
| a_Ref_interface  : interf_idx → a_val_ref.
```

Numerical values of the defensive values have been removed, except for the numerical value of return addresses which is a static information and is used to determine the control flow for subroutines instructions. The semantics of bytecode is modified according to these datatypes, however for some bytecodes (branching instructions for instance), this virtual machine is non-deterministic due to the loss of computational information. Possible resulting states are collected together as the result of bytecode execution, and then, the main execution function has the following signature:

`a_exec : a_state →(set a_rstate)`

The correctness of this virtual machine w.r.t to the defensive virtual machine is expressed through cross-validation. Given the obvious abstraction functions `alpha_da` (resp. `alpha_da_rs`) from defensive state to typed state (resp. return states), the diagram from Figure 3 relating defensive execution `d_exec` and typed execution `a_exec` must be commuting. Note that in this Figure, the curved arrow to `set a_rstate` denotes set inclusion, and that this diagram excludes invocation and return instructions that are handled differently (see [5] for more details).

The offensive virtual machine is built on a similar basis, but uses untyped values and do not perform type verification.

```
Definition o_val_prim := Z.
Definition o_val_ref := Z.
Definition o_val := Z.
```

This virtual machine is closer to a real implementation (faster than the defensive virtual machine for execution, since it does not perform type verification). It has also be proved as safe as the defensive virtual machine, provided the bytecode verification of the executed program has been successful.

The tool Jakarta [5] has been design to generate, given abstraction functions from one virtual machine to another, the abstracted virtual machine as well as proofs of cross-validation. The abstraction process is guided by a script for com-

$$\begin{tikzcd}
\texttt{d\_state} \arrow[r, "\texttt{d\_exec}"] \arrow[d, "\texttt{alpha\_da}"] & \texttt{d\_rstate} \arrow[d, hook, "\preceq\texttt{alpha\_da\_rs}"] \\
\texttt{a\_state} \arrow[r, "\texttt{a\_exec}"] & \texttt{(set a\_rstate)}
\end{tikzcd}$$

Fig. 3. Commutative diagram of defensive and typed execution

plex cases, however only 150 lines of script are needed to produce the expected results for the typed virtual machine from the 5,000 lines long defensive virtual machine. The offensive virtual machine can also be obtain with Jakarta and cross validation results follow a commuting diagram similar to the typed virtual machine. However, for this diagram, the commutation is limited to the cases where the defensive execution does not lead to a type error, since the offensive virtual machine assumes that such errors do not happen. It can also be written with the following formula:

```
∀ (s:d_state), d_exec s ≠ d_Abnormal type_error →
                  alpha_do_rs (d_exec s) = o_exec (alpha_do s)
```

where `alpha_do`, `alpha_do_rs` and `o_exec` are similar to `alpha_da`, `alpha_da_rs` and `a_exec` but for the offensive virtual machine.

Abstract Definition and Construction. The formalization of the bytecode verifier relies on the modules system of Coq. It offers a refined model of the various notions (transition system, fixpoint structure, bytecode verifier, abstract virtual machine, etc.) involved to obtain the bytecode verifier for the defensive virtual machine.

A *bytecode verifier* is given by a type `state` of *states*, an *execution relation* `exec` over states, a set `err` of error states and a predicate `check` such as:

```
forall a:state, (check a) → ¬(bad a).
```

where a state is `bad`, if it is possible to reach from it an error state by successive transitions of the execution relation. Thus the predicate `check` rejects all states that lead by execution to an error state.

The standard way to build such a bytecode verifier is to endow the type of states with a order that does not admit infinite ascending chains, and for which execution is increasing (to guarantee termination), and such that error states are upwards closed. If furthermore execution is deterministic, one can compute for every state `a`, the least fixpoint `b` upper `a`. To do so, we define for every state `a` the least fixpoint `lfp a` below it as:

$$\texttt{lfp a} = \begin{cases} \texttt{a} & \text{if } \texttt{exec a = a} \\ \texttt{lfp (exec a)} & \text{otherwise} \end{cases}$$

Then, we define `check a` as `(err (gfp a))`. As execution is monotone and `lfp a` is the least fixpoint upper `a`, it is clear that such a checking is sufficient to guarantee that `a` is not a bad state.

In a first step, the function `exec` from the above construction is instantiated to the execution function from the typed virtual machine running over the corresponding stackmap for the chosen analysis (monovariant, polyvariant described in Section 2.3). This leads to a verification based on a single method for the typed execution. However, this result can be extended to a result about the defensive execution, by verifying individually each method of the program being verified, by using cross-validation results between typed and defensive virtual machines, and by appealing to a complex invariant between the typed and defensive virtual machine for the case of instructions that change the current frame such as invocations, returns or instructions that raise an exception. Finally, using this bytecode verifier for the defensive virtual machine and results of cross-validation between defensive and offensive virtual machines (including some extras properties on method invocation), we can easily obtain the following expected property:

```
forall s:d_state, (check s) →
(alpha_do_rs (d_exec s)) = (o_exec s).
```

If `s` is the initial state of a program, this property does guarantee that if the verification of the program has been successful, than defensive and offensive execution coincide. This is captured formally by introducing functions that perform several steps of execution, and by showing

```
forall s:d_state, forall n:nat, (check s) →
(alpha_do_rs (d_exec+ n s)) = (o_exec+ n s).
```

As summarized in Figure 4, to take advantage of the bytecode verification framework, the user must provide:

– a defensive virtual machine;
– the definition of abstraction functions for Jakarta abstraction scripts, that are used to construct the abstract virtual machine and an offensive virtual machine. Scripts may contain some minimal amount of proof information to carry cross-validation;
– a formal proof of the correctness w.r.t. bytecode verification of method invocation and exception handling;
– a choice of an analysis and of the corresponding history structures.

Then the user obtains an offensive virtual machine, several bytecode verifiers, and a proof that these bytecode verifiers are correct, in the sense that they will reject programs that go wrong on the defensive virtual machine, and that the offensive and defensive virtual machines coincide on programs that are accepted by bytecode verification.

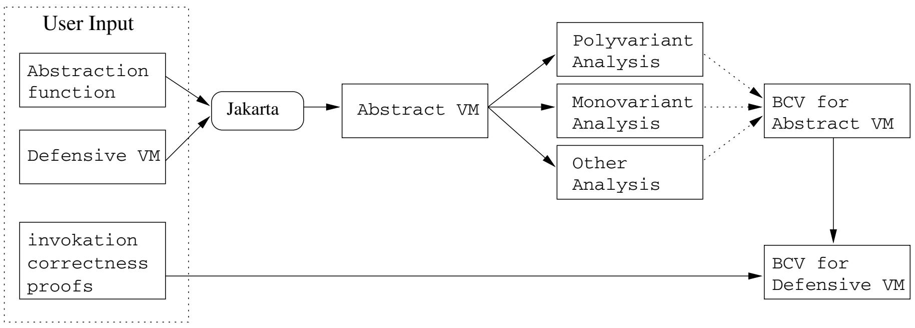


Fig. 4. Construction of verified bytecode verifier

4 Application Validation

While the correctness of the runtime environment is an essential to guarantee the overall security of trusted personal devices, no device can be deemed secure without ensuring that applications do not behave maliciously, in particular with respect to API usage.

The validation of applications against security policies can be addressed at different levels:

– one can enhance existing security architectures to enforce security properties not addressed by current architectures, in particular confidentiality and availability. Verification can be performed by enhanced bytecode verification mechanisms;
– one can abandon the realm of type systems and its associated benefits and choose develop logical methods for specifying and verifying either automatically or efficiently a specific class of security properties. Verification can be performed by (possibly efficient and hence incomplete) logic-based proof inference mechanisms;
– one can exploit the expressive power of logical methods to require that applications, or at least sensitive fragments of applications, are subjected to functional verification, i.e. to verifications that establish their correctness in terms of functionality as well as security.

The different levels may be combined, e.g. by using alias type systems to improve the modularity of functional verification [72]. For the clarity of presentation, we never introduce these different levels separately. In Section 4.1, we discuss possible applications of static analyses and type systems for guaranteeing the security of applications. In Section 4.2, we consider logical verification techniques for verifying specific security properties as well as functional correctness. The paragraph takes the point of view of code developers who want to increase their confidence in the code they develop, and therefore focuses on source program verification; nevertheless we briefly discuss the point of view of the code consumer at the end of this section.

4.1 Enhanced Type Systems and Static Analyses

Program analysis techniques such as type systems and static analysis provide a well-established means to enforce program properties at compile-time or load-time. In the domain of Java-enabled devices, the prime example of program analysis based on type systems is bytecode verification, which is discussed in Section 3.4. In addition, there are several proposals of type systems and static analysis for JVM programs, in particular for eliminating dynamic checks that incur a loss of performance at execution time, and for enforcing security properties that are not guaranteed by the Java security mechanisms.

Static Analyses for Access Control. While the runtime penalty incurred by dynamic access control is acceptable in practice, it is desirable to detect statically that an application may attempt to violate the rules of the firewall, since such attempts will result in a security exception that may block the card. D. Caromel, L. Henrio and B. Serpette [22] have proposed a static analysis that detects statically whether an application may raise a security exception, whereas M. Eluard and T. Jensen [40] have proposed a similar but finer analysis that covers more precise sharing policies. W. Dietl, P. Müller and A. Poetzsch-Heffter [36] manage a similar effect using a type system adapted from ownership type systems. In order for their approach to be practical, downcasts allow to add information to references that can belong to any context into more specific references, e.g. the reference belongs to the currently active context, or is an entry point (of course, such additional information must be verified). The information provided by downcasts is used in type checking, thus the correctness of the type system can only be achieved by combining type checking with runtime checks that verify at that the information associated to downcasts is correct.

An example of a rejected program for these analysis can be built from the program of Figure 2: Suppose that the applet `Alice` contains a public method `baz`, then inside the method `bar` from applet `Bob` a call to `AliceObj.baz()` would lead to a security exception since this `Alice` and `Bob` belong to different contexts and the invoked method does not belong to the shareable interface `SI`.

In a series of articles, T. Jensen and his co-workers [13,14] propose a method to check control-flow properties of Java applets. Their method relies on constructing a finite-state automaton that approximates of the control-flow graph of an application, and checking the automaton against security properties expressed in temporal logic. The method is applicable to stack inspection as well as other properties. Building up on earlier work by F. Pottier, C. Skalka and S. Smith [82], T. Higushi and A. Ohori [50] propose a static type system for access control in the JVM, and develop a sound inference algorithm to verify statically that programs respect the access control policy specified by the type system.

Static Analyses for Secure Information Flow. The firewall mechanism provides a means to control which principals access but does not guarantee that confidential information will not leak to unauthorized principals [71]. In order to avoid principals (that may access information legitimately) to pass the information unduly, it is desirable to devise security mechanisms that enforce stronger confidentiality policies such as non-interference, a high-level security property that guarantees the absence of illicit information flows during a program execution. Non-interference assumes that the variables in a program are either public (low) or secret (high), and requires that the initial values of secret variables do not influence the final values of public variables.

A. Myers [73] and A. Banerjee and D. Naumann [2] propose static enforcement mechanisms for guaranteeing non-interference of Java programs, using extended programs in which methods are declared with security signatures and fields are declared with their security level (in fact, [2] combines information flow and access control but we gloss over the issue here). A example of class declaration in the language of [2] is:

```
class PatientRecord extends Object {
name : (string, L);
hivstatus : (bool, H);
drug : (string, H);

(void, L) setDrug >-H→ (string, L) {
  ...
  }
...
}
```

where the text `>-H->` in the declaration of the method `setDrug` indicates that the method has a high heap effect, i.e. will modify high parts of the heap that store the values of high fields. While the type system of [2] has been proved sound in the sense that typable programs are non-interfering, the type system of [73], which exhibits a richer set of features, lacks a proof of soundness.

An information flow type system for a representative fragment of the JVM that includes classes, methods, and exceptions is given in [9]; the type system is compatible with bytecode verification, and is sound, in that it rejects non-interfering programs. The examples below illustrate some of the illicit information flows that can occur in a stack-based language, and that are detected by the type system of [9]:

```
1 sload $y_H$            1 spush 3
2 if_eq 6                2 spush 4
3 spush 0                3 sload $y_H$
4 sstore $x_L$           4 if_eq 6
5 goto 8                 5 sstore $y_H$
6 spush 1                6 sstore $x_L$
7 sstore $x_L$           7 return
8 return
```

These programs are example of indirect flows, since the final value of the low variable x_L depends on the initial value of the high variable y_H. The problem is caused in the first case by an assignment to x_L in the scope of a branching instruction, and in the second case by an instruction that manipulates the operand stack in the scope of a branching instruction.

To prevent such illicit flows, the type system of [9] manipulates typed states that include security environments, i.e. maps that assign a security level to each program point, and allows branching instructions to update the security environment, e.g. if they branch over a high value.

Although the prevention of illicit information flows is an important concern for the smartcard industry [15,66], and although there have been important achievements in the design of static enforcement methods for information flow security policies [87], the methods have not found substantial applications in practice, partly because information flow policies based on non-interference are too rigid and do not authorize information release, whereas security sensitive applications often release deliberately some amount of sensitive information.

Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases.

In a recent survey [88], A. Sabelfeld and D. Sands provide an overview of relaxed policies that allow for some amount of information release, and a classification along several dimensions, for example who releases the information, and what information is released. While several information flow type systems have been developed to accommodate some dimensions of information release, it is likely that providing flexible static enforcement mechanisms that integrate the different dimensions of declassification will contribute significantly towards a wider use of information flow type systems.

Other Static Analyses and Type Systems. The paragraphs above illustrate some prominent applications of type systems for enforcing security of JVM applications but the scope of properties enforceable by type systems is much wider: for example, G. Schneider *et al* [21,90] have recently developed an efficient analysis to estimate memory usage for Java smartcards. Advanced type systems can also be used to enforce safety policies or to justify aggressive optimization strategies. Existing analyses for JVM programs include array-out-of-bounds analysis [97] using restricted dependent types [96], exception analysis [56] and escape analysis [16], as well as type systems for concurrent fragments of the JVM [41,60].

Implementations. It is noticeable that advanced type systems for the JVM have remained at the level of prototype implementations, and have not found their way in security architectures. While the situation is partially a natural consequence of the exploratory nature of some type systems, P. Fong [44] suggests that the situation may also result from a more fundamental limitation of the Java platform, namely that the verification architecture of the JVM is not designed to support extensibility, and advocates the design of extensible protection mechanisms that can be used to accommodate mechanisms tailored towards enforcing application-specific properties such as confidentiality, access control, or resource management. Building on earlier work on proof linking [45], he proposes an architecture that supports pluggable verification modules, and illustrates the principles of his approach by implementing an access control type system as an instance of a pluggable verification module.

4.2 Logical Verification of Security Properties

Logical Verification Techniques. Research into program logics has a long history, dating back to the seminal work by Floyd [43] and Hoare [51] on program logics and by Dijkstra [37] on weakest precondition calculi in the late 1960s and early 1970s. There has been steady progress since these early days, resulting in tool-supported program logics for realistic programming languages. In particular the last 10 years have seen a burst of activity around Java program verification which has culminated in the realization of verification environments for Java programs. Many of these environments use as specification language the Java

Modeling Language, which we describe below, and can be used for verifying security properties as well as functional properties of Java programs.

Java Modeling Language. The Java Modeling Language [55] (JML) is a behavioral interface specification language designed for Java. It relies on the design by contract approach [69] to guarantee that a program satisfies its specification during runtime. These specifications are given as annotations of the Java source file. More precisely, they are included as special Java comments, either after the symbols //@ or enclosed between /*@ and @*/. For example, the general schema for the annotation of a method is the following:

```
/*@ behavior
  @   requires <precondition>;
  @   ensures <postcondition if no exception raised>;
  @   signals (E) <postcondition when exception E raised>;
  @   assignable <modified fields and variables>;
  @*/
```

where `requires` specifies the conditions on variables, fields and method parameters at the beginning of the method call so that the conditions after `ensures` hold at the end of the method call and the conditions after `signals(E)` hold if an exception is raised and not caught inside the analyzed method. The underlying model is a an extension of Hoare-Floyd logic: if the precondition holds at the beginning of the method call, then postconditions (with and without exceptions) will hold after the call. The `assignable` clause specifies side-effect affected variables and is used during the weakest precondition calculus for method invocations.

Preconditions and postconditions express first-order logic statements, with a syntax following the Java syntax. Thus, they can easily be written by a programmer. The Java syntax is enriched with special keywords: `\result` and `\old(<expr>)` to denote respectively the return value of a method, and the value of a given expression before the execution of the considered method; and `\forall`, `\exists`, $==>$ to denote respectively universal quantification, existential quantification and logical implication.

Besides methods specification, it is also possible to annotate a program with class invariants (predicates on the fields of a class that hold at any time in the class) using the keyword `invariant`, loop invariants (inside the code of a method with loops) using the keyword `loop_invariant`, and assertions (that must hold at the given point of the program) using the keyword `assert`.

Finally, when annotating a program, it might be useful to introduce new variables to keep track of certain aspects or computations. Instead of adding them to the program itself, thus adding new code, it is possible to define variables that will only be used for specification. These variables, called *ghost* variables, are defined in a JML annotation with the keyword `ghost` and assigned to a Java expression with the keyword `set`.

Styles of Specification. Due to its expressiveness and versatility, the JML specification language supports several styles of specifications; the choice of one style of specification over the others depends on the purpose of the verification effort. In a nutshell, one can either opt for lightweight specifications in which one introduces enough annotations to reason about some specific safety property, such as the absence of exceptions, or heavyweight specifications where functional behavior is considered. There is of course a great liberty in how “lightweight” or “heavyweight” a specification should be, and different styles can be used in different parts of an application.

In addition, one may opt for defensive specifications, in which methods are annotated with preconditions that prevent exceptions to occur, or offensive specifications, which use appropriate clauses to specify exceptional postconditions.

Verification Techniques and Tools. JML specifications correctness can be verified either during runtime or statically [18]. To be verified during runtime, the source code must have been compiled using `jmlc`, which is a enhanced Java compiler for JML annotated code. This compiler adds to the generated program assertions checking instructions corresponding to the JML specifications of the program: preconditions, postconditions and loop or class invariants. An exception is raised during the execution if a JML condition fails. The JML runtime assertion checker can be used for unit testing [26].

For the static verification of Java programs, several tools are available using (variations of) JML as specification language. These tools adopt different compromises between soundness and automation, and thus it is useful to use them in combination, starting from automatic but unsound tools, and pursuing with sound but interactive tools. Among these tools, ESC/Java2 [28] offers the higher level of automation as it does not require any user interaction and relies on the Simplify automatic prover. It is particularly useful for checking null pointers or array bounds limits; however it is unsound and incomplete. Other static verification tools such as JACK [20], Jive [70], Krakatoa [65] and Loop [11] generate proof obligations that can be discharged using proof assistants or automatic provers. These tools are sound but require user interaction.

Model-checking techniques provide yet another means to verify the correctness of Java programs against their JML annotations. Bogor [85] exploits such techniques to provide automatic verification of concurrent Java programs with JML annotations.

Annotation Assistants. Program verification using logics may require substantial amounts of annotations in programs, and the costs of annotating programs can become prohibitively high as programs increase in size. Automated support for annotating programs is of great benefit to allow program verification to scale to larger programs.

There are several tools and techniques for inferring annotations. One technique consists in using weakest precondition calculi, possibly in variant form, to generate defensive specifications that prevent run-time exceptions. Another technique to infer annotations is abstract interpretation, which can be used for instance to infer constraints on the range of integer fields, or loop invariants [77], class invariants [64] or object invariants [25]. A third technique consists in instrumenting an existing static analysis to generate annotations related to the property checked by the analysis. These techniques are implemented in some of the aforementioned tools, or in separate tools. In the next paragraphs we illustrate how some annotation assistants can be used to specify security properties of Java applications.

Limitations of JML Technology. Despite having shown its usefulness of a variety of case studies, the JML technology is still under development, and many technical issues remain to be solved. For example, JML is currently not appropriate for reasoning on complex data-structures such as linked-lists or trees because no global property on these structures can be stated in JML. This limitation of JML is related to the first-order logic on which JML is based and that prevents complex quantification over structures or predicates. Also, JML does not allow to in specifications external functions written in a back-end tool, or to use back-end tools to reason about program termination. Another severe restriction of current JML verification tools is the limited support they provide for reasoning about concurrent programs. There are however some ongoing efforts to extend JML with support for reasoning about multi-threaded programs [86].

A final drawback of the JML approach and of the weakest precondition calculus is the difficulty of controlling the shape and size of generated proof obligations. While some techniques have been developed to avoid the size of proof obligations, verifying the functional correctness of some programs, even of modest size, can lead to very complex and unpalatable proof obligations.

High-Level Security Properties

Security Rules. When programming applications using the Java technology, developers are required to follow security rules that pertain to the programming idioms used for developing the applications, and that are intended to complement the security mechanisms provided by the platform. Such security rules cover a wide range of properties, including safety policies not guaranteed by the platform, security properties related to confidentiality or resource control, as well as properties that ensure a correct and legitimate usage of the API. Examples of such properties are detailed below.

Exception Safety. Exception safety is an important concern when developing applications, and guidelines for programming Java applications often consider several forms of exception safety. For example:

- *No runtime exceptions at top-level.* The presence of runtime exceptions at top-level constitutes one common form of programming errors that is undetected by the JavaCard platform security mechanisms. If raised, such exceptions should be caught during the program execution (unlike ISO exceptions whose presence at top-level should not be considered as a programming error).
- *No uncaught exception in transactions.* A first step towards well-formed transactions, defined below, is to guarantee that exceptions do not escape transactions (i.e. that exceptions thrown inside a transaction should be caught inside this transaction).

API Usage. While a careful design of the API is crucial for constraining the interactions between applications and its environment on the card, the security of the device can only be guaranteed if applications perform a correct and legitimate usage of the API. Therefore, a large number of security rules focus on API usage.

- *Well-formed transactions.* In order to maintain the card in a coherent state, the API provides a transaction mechanism that guarantees the atomicity of updates performed within transactions, see Section 2.2. This rule requires that all calls to `beginTransaction` are matched by exactly one call to `commitTransaction` or `abortTransaction`, and conversely.
- *Bounded retries.* No authentication may happen within a transaction. The rule is designed to deter attacks that exploit the atomicity properties of transactions in combination with timing leaks in the implementation of the authentication mechanisms. In a nutshell, if authentication is done within a transaction and response times to authentication challenges are longer in case of a negative reply (say t_0 for a positive reply and $t_0 + t$ for a negative reply), it is possible for the user to pull out the card between t_0 and $t_0 + t$ in the absence of reply from the card (and hence if authentication failed). By pulling out the card, the user forces the card to roll back to the state prior to starting the transaction, and in particular to reset the retry counter. Hence the user allows himself an unbounded number of retries. Beyond its anecdotal nature, this example illustrates that security rules can be crafted to account for vulnerabilities that would be difficult to capture at a more semantical level.

Security guidelines for application developers may also include more specific security rules about API usage, including:

- legitimate and controlled use: instances of such rules include forbidding or restricting calls to given methods, e.g. forbidding GSM applications to call the method that trigger the sending of SMS messages, or requiring that calls to such methods are only performed in authenticated mode;
- privacy: in case of a GSM application requiring access to some positioning system (such as Global Positioning System GPS) to customize their services, security rules may be set to guarantee the application does not divulge the location of the phone and its owner to an unauthorized parties.

While falling short of providing a solid foundation to software security, such rules provide partial guarantees for properties that are difficult to capture formally, and are important in practice because they embody the know-how of security experts.

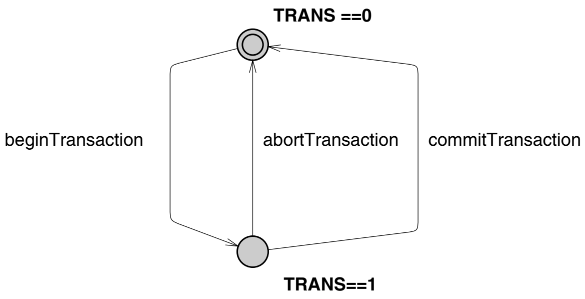
Enforcing Security Rules. Verification techniques based on JML provide an effective means to enforce many security rules for Java applications. In particular, M. Pavlova *et al* [79] propose a general method for guaranteeing a correct usage of the API:

1. the developer express the security rule in some appropriate formalism. Following an established practice popularized by S. Schneider [89], security properties are expressed as automata that constrain the usage of API methods. Intuitively, security automata specify at an abstract level, and independently of any program, preconditions that must hold at certain points during execution[1];
2. JML annotations are synthesized from the automaton and attached to those API methods whose behavior is specified by the automaton. Then JML annotations are propagated throughout the program, and proof obligations are generated from the annotated program. The propagation of the JML annotations is performed by some ad-hoc propagation algorithm, and proof obligations are generated by a verification condition generator. Currently, the propagation of annotations and the generation of proof obligations is performed in two different passes, although it is possible to perform both tasks in a single task due to the similarities between the verification condition generator and the propagation algorithm;
3. proof obligations are dispatched to automatic provers or interactive proof assistants, where they can be discharged by decision procedures or through user interaction. If all proof obligations can be discharged, then the application respects the security property. As a corollary, it follows that inserting runtime checks to monitor that the application only performs "allowed transitions" is superfluous.

The method has been implemented in the JACK verification environment and applied to several case studies from the smartcard domain. The overall conclusion is that the method contributes to carrying out formal security analyses and to improving the quality of applications, while remaining reasonably accessible to developers which are not familiar with formal techniques. Nevertheless the method is by no means complete. For example, B. Jacobs, C. Marché and N. Rauch [52] report on an experiment that goes beyond the technique of [79] and involves a substantial amount of work, both in terms of (non-automated) specification and interactive verification, for checking exception safety of a commercial smartcard applet.

Example. We illustrate how the method can be used to enforce atomicity properties, and in particular that transactions are well-formed. In order to specify that transactions are well-formed, we use an automaton involving one variable `TRANS` initialized to 0 and used to record whether or not there is an ongoing transaction, that corresponds to the two states `TRANS==0` and `TRANS==1`. Then the automaton contains transitions that correspond to correct calls of the methods for beginning, committing and aborting a transaction:

[1] This first step of the method is not reported in *loc cit*, its implementation being posterior to its publication.



The first step towards verifying the well-formedness property is to declare the variables that appear in the automaton with their initial values, here:

```
/*@ static ghost int TRANS == 0; @*/
```

and to generate so-called core annotations for the API methods that appear in the automaton, which in the case of the method `beginTransaction` is:

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 1; @*/
public static native void beginTransaction()
                          throws TransactionException;
```

while in the case of `commitTransaction` the core annotation is:

```
/*@ requires TRANS == 1;
  @ assignable TRANS;
  @ ensures TRANS == 0; @*/
public static native void commitTransaction()
                          throws TransactionException;
```

Then, the second step is to propagate the annotations in the application being verified. We consider two code fragments.

In the first code fragment, we have a method `m`, whose only method calls are those shown, and which does not contain any set annotations.

```
public static void m() { ... // some computations
           JCSystem.beginTransaction();
           ... // some computations within transaction
           JCSystem.commitTransaction(); }
```

The problem is to annotate `m` in such a way that the precondition of `m` guarantees the precondition of `beginTransaction`, and dually that the postcondition of `m` follows from the postcondition of `commitTransaction`. As we assume that `TRANS` is not modified by the code that precedes the call to `beginTransaction` or that follows the call to `commitTransaction`, the precondition of `m` is inherited from that of `beginTransaction`, and the postcondition of `m` is inherited from that of `commitTransaction`. Our algorithm to propagate annotations will generate exactly such a specification:

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 0; @*/
public static void m() { ... // some computations
           JCSystem.beginTransaction();
           ... // some computations within transaction
           JCSystem.commitTransaction(); }
```

Note that one could device finer propagation algorithms that account for the computations performed by m, but the resulting specification of m would become cluttered. Upon completion of the annotation phase, one must proceed with the verification of the annotated method. For the method above, the proof obligations will include a proof obligation that the postcondition of m is guaranteed by the postcondition of commitTransaction, a proof obligation that the precondition of m ensures the precondition of beginTransaction, and a proof obligation that the postcondition of beginTransaction ensures the precondition of commitTransaction. These proof obligations can be discharged automatically.

The second code fragment aims at illustrating that verification environments can also provide mechanisms to trace the potential source of violations of security properties. If we apply our method to the code fragment:

```
public static void m() { ... // some computations
        JCSystem.beginTransaction();
        ... // some computations within transaction
        JCSystem.beginTransaction();
       ... // some computations within nested transaction
        JCSystem.commitTransaction();
        ... // some computations within transaction
        JCSystem.commitTransaction(); }
```

we shall obtain 5 proof obligations including a proof obligation that (in essence) TRANS == 1 implies TRANS == 0, which corresponds to the proof obligation that ties the postcondition of (the first call to) beginTransaction with the precondition of (the second call to) beginTransaction. Verification environments such as Jack highlight the code fragment that corresponds to the unprovable proof obligation, and thus indicate the source of the violation of the proof obligation. As pointed in [61], the verification condition generation mechanism of Jack is particularly well suited for linking unprovable proof obligations with problematic code fragments, but the authors of [61] show that it is possible to achieve a similar effect for other forms of verification condition generators that avoid generating an exponential number of proof obligations [42].

Secure Information Flow. Due to its particular nature, a major concern of smartcards applications is to guarantee confidentiality and integrity of data. Section 4.1 advocates to address confidentiality issues from the perspective of non-interference, and indicates that information flow type systems can be used for enforcing non-interference. However, information flow type systems are extremely conservative and reject many secure programs.

Thus several works such as [32,6] suggest the use of program logics such as dynamic logic or Hoare logic to verify non-interference. In these frameworks, non-interference is reduced to a property about a single program execution using self-composition: basically, a program P is non-interfering if, given two sets of inputs that only differ by the values of secret data, the successive execution of P for each of these set leads to observably equal public outputs.

More formally, as presented in [6], let $\vec{l}$ and $\vec{h}$ be the sets of respectively public (low) and secret (high) variable of a simple (without mutable structures) imperative program P, let $\vec{l'}$ (resp. $\vec{h'}$) be a renaming with fresh names of the variables of $\vec{l}$ (resp. $\vec{h}$), then $P(\vec{l}, \vec{h})$ is non-interferent if before the execution $\vec{l}$ and $\vec{l'}$ are such that $\vec{l} = \vec{l'}$ then after the execution of $P(\vec{l}, \vec{h}); P(\vec{l'}, \vec{h'})$, where `;` is the usual sequential composition, the equality $\vec{l} = \vec{l'}$ still holds. Or equivalently, the following Hoare triple must be valid:

$$\{\vec{l} = \vec{l'}\}P(\vec{l}, \vec{h}); P(\vec{l'}, \vec{h'})\{\vec{l} = \vec{l'}\}$$

Although the above definition of non-interference with self-composition uses Hoare logic, verifying this property for a given program is not possible with standard JML tools. Indeed, specifications relate to the self-composed program, not the original one, make use of renamed variables and special care must be taken for method invocation. In [39], the Krakatoa tool and JML have been extended to allow specifications for non-interference, without changing the underlying weakest precondition calculus of the tool.

As mentioned in Section 4.1, non-interference is too strict since it forbids any computation on secret variables observable on public variables, even those expected for the normal behavior of smartcards, such as PIN code verification. One advantage of JML is that it provides a precise relationship between inputs and outputs of the program. Thus, it becomes possible to capture information release using JML. An example of information release, expressed with the extended version of JML presented in [39] is given below. This example deals with PIN code verification. The secret variable `pin` stores the actual PIN code, the public input variable `in` represents the attempted code, and the public output variable `acc` reveals whether `in` and `pin` agree. The specification of non-interference with declassification follows: tested with two different sets of input that are both valid or both invalid tries, the algorithm must give identical result for the access.

```
/*@ public normal_behavior
  @ requires_ni (\ni1(in)==\ni1(pin))<==>(\ni2(in)==\ni2(pin))
  @ ensures_ni \ni1(acc) == \ni2(acc);
  @*/
void pin_verification(int in) {
  if (in == pin)
    acc = true;
  else
    acc = false;
}
```

In this code, `\ni1(<var>)` (resp. `\ni2(<var>)`) correspond to the renamed variables that appear in self composition, `<==>` is logical implication, and `requires_ni` (resp. `ensures_ni`) is a keyword to introduce pre-condition (resp. post-condition) of the Hoare triple in self composition specifications.

Resource Consumption. Controlling resource consumption of downloaded applications is a compulsory measure for preventing denial of service attacks on devices with restricted resources. Yet current security architectures do not provide any mechanism to control resource consumption. In this section, we illustrate how JML can be used to specify and verify memory usage of Java applications.

Principles. Verification techniques based on JML are also appropriate for performing a precise analysis of resource consumption for Java programs. The basic idea, described in [8], consists in:

1. using a ghost variable `Mem` that provides an upper bound for the memory consumed by the program at any given program point;
2. attaching to each method `m` a postcondition that predicts its memory consumption. The prediction can be express at different levels of granularity. For example, the estimate may be global, or it may account for the value of inputs, or it may distinguish between normal and abnormal termination, or between the different types of memory;
3. verifying for each method that the predicted memory consumption is indeed an upper bound of the memory consumption, by inserting immediately after every bytecode that allocates memory a ghost assignment that increments of `Mem` by the amount of memory consumed by the allocation, and verifying the resulting annotated program.

The correctness of the approach can be derived from the correctness of logical verification techniques. As a consequence, every run of a method will not consume more than `km` memory units, provided the corresponding annotated method has been proved correct. Of course, the correctness of the approach hinges on the fact that the programmer correctly specified the amount of memory consumed by each allocation.

The approach is practical in that it allows to specify and enforce precise memory consumption policies for applications that may involve such features as exceptions, subroutines, and recursive methods. Its practicality can be further enhanced by annotation assistants which infer appropriate JML annotations about memory consumption. Of course, such assistants are necessarily incomplete an user interaction may be required for exploiting the full power of JML. In addition, user interaction can be required to discharge proof obligations generated from the annotated applet.

While our approach is purely static, L.-A. Fredlund [46] suggests to use a runtime monitor to control the execution of JavaCard applets whereas A. Chander *et al* [24] propose to control memory consumption with an hybrid approach that combines static and dynamic verifications. A further difference between our approach and that of Chander *et al* is that they do not require user interaction; while avoiding user interaction does in principle restrict the scope of their method, they show the applicability and effectiveness of their tool on a substantial example.

Example. One can specify and verify that the method `m` below will not use more than `km` memory units

```
public void m(A a) {
  if (a == null) {
    a = new A();
  }
  a.b = new B();
}
```

by annotating the program as follows

```
//@ ensures Mem <= \old(Mem) + km;
public void m(A a ) {
  if (a == null) {
    a = new A();
    //@ set Mem += ka;
  }
  a.b = new B();
  //@ set Mem += kb;
}
```

and verifying that the annotated method is correct provided `ka`+`kb` $\leq$ `km`. Due to the simplicity of the code (e.g. it does not contain loop nor recursive methods), the annotation assistant is able to infer the specification of the method, once given (over)estimates `ka` and `kb` of the memory consumed by the allocation of an instance of class `A` and `B` respectively.

Other Quantitative Properties. Due to their constrained resources, trusted personal devices are very vulnerable to denial-of-service attacks, and it is therefore important to control many facets of the resource usage made by applications. In addition to memory consumption, which is discussed above, security policies may concern such resources as library calls, communication channels, bandwidth, and power consumption. For some resources, it is possible to adapt the technique described above.

Verification from the Code Consumer Perspective. Throughout this section, the problem of applet validation has been addressed from the perspective of the application developer who wants to gain confidence in the quality of the software it develops. Thus, we have focused on applet validation at source code level. Such a focus has been quite common in the area of smartcards, where currently the card issuer has control over the applications loaded on the card, and is likely to have access to the applications source code.

The situation is somewhat different in other application areas of trusted personal devices, and in particular in the area of telecommunications where operators are faced with the possibility of offering their customers new services by deploying applications originating from an untrusted software company. The issue here is that operators, while worried about the negative impact on business if the code is malicious or simply erroneous, do not have access to the source code of applications both for intellectual property reasons (the software company does not want to disclose its source code). In such a scenario, the issue arises of verifying bytecode applications.

Fortunately, logical techniques such as those presented above are also applicable to bytecode level, and can be used by the code consumer to check that the application that he downloads is secure. For example, C. Quigley [84] considers program logics for Java bytecode, as do F. Bannwart and P. Müller [3]. Further, M. Wildemoser and T. Nipkow [95] have formalized in Isabelle/HOL a verification condition generator for a fragment of Java bytecode, and shown its correctness w.r.t. an operational semantics. Motivated by the prospect of bringing the benefits of source code verification to the code consumers, L. Burdy and M. Pavlova [19] have also developed a verification condition generator for Java bytecode programs. The verification condition generator, which has been integrated in JACK, outputs proof obligations from extended class files that can either be obtained manually by inserting annotations into user-defined attributes, or by a compiler that compiles JML annotated Java programs into extended class. In both cases, annotations are written in the Bytecode Modeling Language (BML), which may be considered as the counterpart of JML for bytecode. Rather similar work for C# has taken place in the context of the Spec# project [4], which has defined an extension of C# with annotations and type support for nullity discrimination. Such annotated programs are then compiled with their specifications to extended .NET files, which can be run using the .NET platform. Specifications are checked at run-time or verified using the Boogie static checker.

Bringing the benefits of source code verification to code consumers not only requires to develop logical methods to reason about bytecode programs, but also to develop mechanisms that allow code consumers to check efficiently that programs are correct. In their work of Proof Carrying Code, G. Necula and P. Lee [75,74] propose a general solution to this problem by requiring that components should come equipped with a certificate which can be used by the consumer to verify statically that the components are correct. They also advocate the idea of certifying compilation [76], where the focus is on safety properties which can be proven automatically through an extended compiler that synthesizes annotations from the information it gathers about a program, and a checker that discharges proof obligations generated by the verification condition generator.

Some of the security properties described above cannot be verified automatically without a loss of precision, and thus as a result of favoring automatic verification, certifying compilation does not fully exploit the flexibility and expressiveness provided by logical verification techniques. One complementary approach to certifying compilation consists in exploiting the results of source code verification, and constructing certificates for bytecode programs from certificates for the corresponding source code programs. In some recent unpublished work, G. Barthe, T. Rezk and co-workers have showed for a restricted fragment of Java that proof obligations are almost preserved by non-optimizing compilers; thus in this context it is possible to reuse certificates almost directly. Preservation of proof obligations by compilation is destroyed by simple program optimizations; nevertheless preliminary experiments suggest that it is possible to construct from certificates for source code programs certificates for bytecode programs obtained by optimizing compilation.

Types vs. Logics. While Section 4.1 focused on the use of type systems to guarantee that applications respect security policies related to confidentiality and memory consumption, this section shows that logical verification methods can be used to enforce the same policies. The main benefits of using logical verification techniques are: expressiveness (logics support customizable security policies), versatility (the same logic can be used for a great variety of analyses, thus allowing analyses to be combined), and precision (logics can be used to provide precise statements of program behavior).

In order to improve the quality of their static analyses, several researchers are now exploring the possibility of making a systematic use of logical verification methods to verify program properties that are traditionally checked by static analyses, for example ownership [35]. This line of work seems promising, and opens the perspective for precise static analyses. However such program analyses based on logic may require user interaction (for discharging the proof obligations generated by the analysis of the program), especially if they seek to exploit the full power of logic in an analysis that combines expressiveness and precision. Unfortunately, requiring substantial user interaction is an obstacle to the scalability of this method.

On the other hand, type systems benefit from a combination of three important features: simplicity (types can be viewed as a particularly simple form of assertions), automation (type systems compute a decidable approximation of the property to be enforced) and scalability (type systems are compositional and allow to reduce the verification of a complex program into simpler verification tasks). They are thus amenable to on-device checking. In order to get the best of both worlds, it seems therefore relevant to gain a more systematic understanding of the relationship between program logics and type systems, and eventually to combine both verification methods in a single technique that remains precise while avoiding excessive user interaction.

Application Correctness. Enforcing security properties of JVM applications is an important step towards guaranteeing the security of trusted personal devices. In most situations, verifying that applications adhere to a security policy will provide sufficient guarantees (under the assumption that the platform is correctly implemented). In some situations, one may however be interested in achieving a higher degree of reliability by showing that an application, or a fragment of it, has the expected functionality. As a fully fledged behavioral interface specification language, JML offers the possibility to verify and specify the functional behavior of Java applications, and many of the JML tools used for applet validation can also be used to verify functional specifications.

The most immediate application of functional verification is to establish that some fragment of a Java application is correct w.r.t. its intended behavior. For example, C.B. Breunesse *et al* [17] show that decimal arithmetic is correctly implemented in an electronic purse application. Below we briefly describe two further applications, and point to some of the issues involved.

JavaCard API. Section 3.3 provides a general methodology to reason about the Java API, for example to prove that some method from the API has some expected functional behavior. Such a verification is valuable from the point of view of guaranteeing the correct design of the API, but leaves open the issue of the implementation of the API, which may not coincide with its description in the formal model. Therefore one must ensure that the API is correctly implemented. Since a substantial part of the API is implemented in Java, it is possible to use JML and its associated verification techniques for this purpose.

E. Poll and its co-workers [12,68] have developed reference JML specifications of the JavaCard API, and verified formally the correctness of some API methods against their specification. For example, they establish the correctness of the methods in the class AID, and in particular of the method `getBytes` which is called to get the AID bytes encapsulated within AID object. The method takes two parameters `dest` which represents the byte array to copy the AID bytes, and `offset` which represents within `dest` where the AID bytes begin, and returns the length of the AID bytes. Its code and specification are given in Figure 5. The JML annotations aims at specifying that bytes representing the AID are accurately copied to the right position in the given array `dest` and at constraining the behavior of the method if an exception is raised (by `arrayCopy`). The corresponding proof obligations will directly follow from the specifications of `arrayCompare` and `arrayCopy` used in the annotations and in the code.

The implementation of the method `getBytes` contains a call to the native method `arrayCopy`; in order to reason about the correctness of the method `getBytes`, one must therefore dispose of a JML specification of `arrayCopy`. As native methods are not implemented in Java, we cannot use JML tools to verify them against their specifications, and thus we must trust that the specification of native methods is faithful to the implementation. Thus the correctness of the method `getBytes` will be established under the assumption that the specification of the native method is correct. The correctness proof is easy; it should be pointed however that generating the verification conditions that guarantee the correctness of the method involves subtle semantical issues, including the semantics of method calls in specifications [33].

Java-Based Components. Trusted personal devices are evolving from dedicated devices with a specific usage into general purpose devices that must provide users with a uniform access to multiple services. Such an evolution raises some

```
/*@ public behavior
  @ requires dest != null;
  @ requires dest != theAID;
  @ requires 0 <= offset;
  @ requires offset+theAID.length <= dest.length ;
  @
  @ assignable dest[offset..offset+theAID.length-1];
  @
  @ ensures \result == theAID.length;
  @ ensures Util.arrayCompare(theAID,(short)0,
  @         dest,offset,(short)theAID.length) == 0;
  @
  @ ensures
  @         (\forall short i; 0 <= i && i < theAID.length
  @                       ==> theAID[i] == dest[offset+i]);
  @
  @ signals (NullPointerException) dest == null;
  @ signals (ArrayIndexOutOfBoundsException) dest != null;
  @ signals (ArrayIndexOutOfBoundsException)
  @         (0>offset || offset + theAID.length > dest.length);
  @*/

public byte getBytes (byte[] dest, short offset)
  throws ArrayIndexOutOfBoundsException, NullPointerException {
  Util.arrayCopy(theAID, (short) 0,
                 dest, offset, (short) theAID.length);
  return (byte) theAID.length;
}
```

Fig. 5. JML specification of getBytes method from AID class

technical difficulties: in particular, trusted personal devices are heterogeneous in their computational infrastructure (operating systems, communication protocols, libraries) and resources (memory, power autonomy, connectivity), and due to physical, technological and economical constraints, it is not possible to install on such devices all functionalities that may be needed *a priori* by potential applications. One promising solution to overcome this limiting factor is to embed on devices extensible virtual machines that can be extended with the computational infrastructure, platform or libraries, needed to execute the required services. Such a solution also enables remote system upgrades that allow device issuers to perform maintenance over the network in situations that would otherwise require devices to be recalled. Such remote updates are an attractive possibility (although there are some concerns about negative implications of issuers having remote control over devices) and form an integral part of the deployment model in application domains such as telephone networks and digital video infrastructures. The adoption of a component-based approach in the development of runtime environments and security architectures for trusted personal devices, and the perspective of downloading such components dynamically, raise important security issues as the security of a device might be fatally compromised upon installing a faulty or malicious component. In such instances, it becomes important to be assured of the functional correctness of incoming programs prior to their loading on device.

Many current projects to develop modular virtual machines adopt Java as their programming language [94,54,78], making it conceivable to use JML verification tools to establish the functional correctness of key components. Preliminary experiments suggest that it is feasible to prove component correctness in toy examples, e.g. to prove the correctness of a bytecode verifier for a toy assembly language. However, even dealing with toy examples involves a number of difficulties that range from JML semantics to scalability of JML verification techniques. Under such circumstances, proving formally the correctness of a realistic component remains beyond current state of the art, and an exciting challenge for JML verification technology.

5 Perspective

Smart cards have proved an ideal application domain for formal methods, and the last few years have seen substantial achievements both in the area of platform verification and application validation. In spite of such scientific progress, formal methods have not gained a widespread acceptance in industry, and the use of formal methods in the smart cards and trusted personal devices industry is quite often confined to R&D laboratories. According to a recent roadmap [1], cost effectiveness and scalability remain two bottlenecks for a wider use of formal methods in the smartcard industry; see also [59] for another industrial perspective on this issue. Addressing these bottlenecks is an important engineering challenge to be tackled by the formal methods community.

In addition, the increasing complexity and connectivity of trusted personal devices raise new challenges and opportunities for formal methods. Some specific technical challenges have already been mentioned in this chapter: extending platform verification and applet verification techniques to cover multi-threading, developing a systematic encoding of security properties in interface specification languages such as JML, integrating type systems and logical verification techniques, establishing the functional correctness of system components. These specific challenges will serve as stepping stones towards the much greater challenge of achieving a better integration of formal methods in security architectures for trusted personal devices, and eventually security architectures for large networks of Java-enabled devices. The forthcoming European project "MOBIUS: Mobility, Ubiquity, and Security" [83] will aim at addressing many of these challenges and opportunities in the context of large and distributed networks of Java-enabled devices that aim at providing services globally, uniformly, and securely.

Acknowledgments. The authors are partially supported by the European IST project INSPIRED, the RNTL project CASTLES and the ACI Sécurité SPOPS and GECCOO. The work reported here has been partially funded by the European IST projects PROFUNDIS and VERIFICARD, by the European thematic network RESET, and by the INRIA research actions S-Java and MODOCOP.

The authors are grateful to Julien Forest and Tamara Rezk for their comments on an earlier version of the chapter, and to the former and present members of the EVEREST and LEMME teams at INRIA Sophia-Antipolis for stimulating interactions over the years.

References

1. Roadmap for European Research on Smartcard Technologies. `http://www.ercim.org/reset`
2. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.
3. F. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Proceedings of Bytecode'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005.
4. M. Barnett, K.R.M. Leino, and W. Schulte. The spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 50–71. Springer-Verlag, 2005.
5. G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Jakarta: tool-assisted specification and verification of the JavaCard Platform. *Journal of Automated Reasoning*, 2006. To appear.
6. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.
7. G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. In *Proceedings of FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 99–113. Springer-Verlag, 2004.
8. G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In B. Aichernig and B. Beckert, editors, *Proceedings of SEFM'05*. IEEE Press, 2005.
9. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
10. D. Basin, S. Friedrich, and M. Gawkowski. Bytecode Verification by Model Checking. *Journal of Automated Reasoning*, 30(3-4):399–444, December 2003.
11. J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.
12. J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'00*, volume 2041 of *Lecture Notes in Computer Science*, pages 137–150. Springer Verlag, 2001.
13. F. Besson, T. Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proceedings of PPDP'02*, pages 76–87. ACM Press, 2002.
14. F. Besson, T. Jensen, D. Le Métayer, and T.Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

15. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
16. B. Blanchet. Escape analysis for java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.
17. C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and Verifying a Decimal Representation in Java for Smart Cards. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 304–318. Springer-Verlag, 2002.
18. L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005. To appear.
19. L. Burdy and M. Pavlova. Annotation carrying code. Manuscript, 2005.
20. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME'03*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
21. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Proceedings of FM'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
22. D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java card object sharing. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2001.
23. L. Casset, L. Burdy, and A. Requet. Formal Development of an Embedded Verifier for JavaCard ByteCode. In *Proceedings of DSN'02*. IEEE Computer Society, 2002.
24. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag, 2005.
25. B.-Y.E. Chang and K.R.M. Leino. Inferring object invariants. In A. Cortesi and F. Logozzo, editors, *Proceedings of AIOOL'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005. To appear.
26. Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP'02*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, 2002.
27. A. Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, 2004.
28. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML — progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
29. Connected Limited Device Configuration (CLDC) and the K Virtual Machine (KVM). `http://java.sun.com/products/cldc`
30. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
31. Common Criteria. `http://www.commoncriteria.org`
32. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Informal proceedings of WITS'03*, 2003.

33. A. Darvas and P. Müller. Reasoning About Method Calls in JML Specifications. Manuscript, 2005.
34. D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *Proceedings of WIESS'02*. Usenix Association, 2002.
35. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.
36. W. Dietl, P. Müller, and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 129–150. Springer-Verlag, 2005.
37. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
38. G. Dufay. *Vérification formelle de la plateforme JavaCard*. PhD thesis, Université de Nice Sophia-Antipolis, 2003.
39. G. Dufay, A. Felty, and S. Matwin. Privacy-Sensitive Information Flow with JML. In R. Nieuwenhuis, editor, *Proceedings of CADE'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. To appear.
40. M. Eluard and T. Jensen. Secure object flow analysis for java card. In *Proceedings of CARDIS'02*, pages 97–110. USENIX Association, 2002.
41. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI'03*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.
42. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of POPL'01*, pages 193–205. ACM Press, 2001.
43. R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
44. P. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Proceedings of OOPSLA'04*, pages 404–418. ACM Press, 2004.
45. P. Fong and R. Cameron. Proof linking: modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 9(4):379–409, October 2000.
46. L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In K. Havelund and G. Rosu, editors, *Proceedings of RV'04*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 217–233. Elsevier Publishing, 2004.
47. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
48. P. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.
49. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
50. T. Higuchi and A. Ohori. A static type system for JVM access control. In *Proceedings of ICFP'03*, pages 227–237. ACM Press, 2003.
51. C. A. R. Hoare. An axiomatic basis for computer programming. *Commununications of ACM*, 12(10):576–580, 1969.
52. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of AMAST'04*, volume 3116 of *Lecture Notes in Computer Science*, pages 241–257. Springer-Verlag, 2004.
53. JavaCard Technology. `http://java.sun.com/products/javacard`

54. Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`
55. JML Specification Language. `http://www.jmlspecs.org`.
56. J.W. Jo, B.M. Chang, K. Yi, and K.M. Choe. An uncaught exception analysis for Java. *Journal of systems and software*, 72(1):59–69, 2004.
57. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.
58. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
59. J.-L. Lanet. Are smart cards the ideal domain for applying formal methods? In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proceedings of ZB'2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 363–374, 2000.
60. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290(1):741–778, October 2002.
61. K.R.M. Leino, T. Millstein, and J.B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55:209–226, March 2005.
62. X. Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.
63. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
64. F. Logozzo. Automatic inference of class invariants. In G. Levi and B. Steffen, editors, *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer-Verlag, 2004.
65. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard Programs annotated with JML Annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
66. R. Marlet and D. Le Métayer. Security properties and java card specificities to be studied in the secsafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
67. C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Proceedings of MMMACNS*, volume 2052 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
68. H. Meijer and E. Poll. Towards a full formal specification of the java card. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 165–178. Springer-Verlag, 2001.
69. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.
70. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from `sct.inf.ethz.ch/publications`, 2000.
71. M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
72. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
73. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
74. G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
75. G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

76. G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344, 1998.
77. J.W. Nimmer and M.D. Ernst. Automatic generation of program specifications. In *Proceedings of ISSTA'02*, volume 27, 4 of *Software Engineering Notes*, pages 232–242. ACM Press, 2002.
78. OVM project. `http://www.ovmj.org/`
79. M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*. Kluwer, 2004.
80. Global Platform. See `http://www.globalplatform.org`
81. J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Proceedings of ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.
82. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, March 2005.
83. Mobius Project. `http://mobius.inria.fr`
84. C.L. Quigley. A Programming Logic for Java Bytecode Programs. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.
85. Robby, E. Rodríguez, M.B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model-checking framework. In K. Jensen and A. Podelski, editors, *Proceedings of TACAS'04*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420, 2004.
86. E. Rodriguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending jml for modular specification and verification of multi-threaded programs. In *Proceedings of ECOOP'05*, volume 3586 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
87. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
88. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press, 2005.
89. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.
90. G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, 2004.
91. I. A. Siveroni. Operational semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, 2004.
92. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
93. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.
94. Java In the Small Project. `http://www.lifl.fr/rd2p/jits/`
95. M. Wildmoser and T. Nipkow. Asserting bytecode safety. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2005.
96. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.
97. H. Xi and S. Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proceedings of CASCOON'99*, pages 110–125, November 1999.

Privacy-Preserving Database Systems

Elisa Bertino, Ji-Won Byun, and Ninghui Li

Department of Computer Science and Cerias,
Purdue University,
656 Oval Drive, West Lafayette, IN 47907
{bertino, byunj, ninghui}@cs.purdue.edu

Abstract. Privacy is today an important concern for both users and enterprises. Therefore, intense research is today being carried out on various aspects of privacy-preserving data management systems. In this paper, we focus on database management systems (DBMS) able to enforce privacy promises encoded in privacy languages such as P3P. In particular, in the paper, we first present an overview of the P3P language and outlines some of its critical aspects. We then outline the main requirements for a privacy-preserving DBMS and we discuss solutions related to the management of privacy-related meta-data, focusing on special category of meta-data information, that is, purpose information. Purpose information represents an important component of privacy statements and thus their effective management is crucial. We then discuss current solutions to to fine-grained access control in the context of relational database systems and identify relevant issues.

1 Introduction

Data represent today an important asset. We see an increasing number of organizations that collect data, very often concerning individuals, and use them for various purposes, ranging from scientific research, as in the case of medical data, to demographic trend analysis and marketing purposes. Organizations may also give access to the data they own or even release such data to third parties. The number of increased data sets that are thus available poses serious threats against the privacy of individuals and organizations. Because privacy is today an important concern, several research efforts have been devoted to address issues related to the development of privacy-preserving data management techniques.

A first important class of techniques deals with privacy-preservation when data are to be released to third parties. In this case, data once are released are not any longer under the control of the organizations owning them. Therefore, the organizations owners of the data are not able to control the way data are used. The most common approach to address the privacy of released data is to modify the data by removing all information that can directly link data items with individuals; such a process is referred to as data anonymization. It is important to note that simply removing identity information, like names or social-security-numbers, from the released data may not be enough to anonymize the data. There are many examples showing that even when such information is removed from the released data, the remaining data combined with other information sources may still link the information to the individuals it refers to. To overcome this problem, approaches based on generalization techniques have been proposed, the most well known of which is based on the notion of k-anonymity [27,28].

A second class of techniques deals specifically with privacy-preservation in the context of data mining. Data mining techniques are today very effective. Thus even though a database is sanitized by removing private information, the use of data mining techniques may allow one to recover the removed information. Several approaches have been proposed, some of which are specialized for specific data mining techniques, for example tools for association rule mining or classification systems, whereas other are independent from the specific data mining technique. In general all approaches are based on modifying or perturbing the data in some way; for example, techniques specialized for privacy preserving mining of association rules modify the data so to reduce the confidence of sensitive association rules. A problem common to most of those techniques is represented by the quality of the resulting database; if data undergo too many modifications, they may not be any longer useful. To address this problem, techniques have been developed to estimate the errors introduced by the modifications; such estimate can be used to drive the data modification process. A different technique in this context is based on data sampling [7]. The idea is to release a subset of the data, chosen in such a way that any inference made from the data has a low degree of confidence. Finally, still in the area of data mining, techniques have been developed, mainly based on commutative encryption techniques, the goal of which is to support distributed data mining processes on encrypted data [8]. In particular, the addressed problem deals with situations in which the data to be mined is contained at multiple sites, but the sites are unable to release the data. The solutions involve algorithms that share some information to calculate correct results, where the shared information can be shown not to disclose private data.

Finally, some efforts have been reported dealing with database management systems (DBMS) specifically tailored to support privacy policies, like the policies that can be expressed by using the well known P3P standard [29]. In particular, Agrawal et al. [1] have recently introduced the concept of Hippocratic databases, incorporating privacy protection in relational database systems. In their paper, Agrawal et al. introduce the fundamental principles underlying Hippocratic databases and then propose a reference architecture. An important feature of such architecture is that it uses some privacy metadata, consisting of privacy policies and privacy authorizations stored in privacy-policies tables and privacy-authorizations table respectively. There is strong need for development of privacy-preserving DBMS driven by the demand organizations have of complying with various privacy laws and requirements and of increasing user trusts [19]. The development of database technology entails however addressing many challenging issues, ranging from modeling to architectures, and may lead to the next-generation of DBMS.

In this paper we focus on the development of privacy-preserving DBMS because it poses several challenges with respect to both theory and architectures.

It is important to notice, however, that privacy-preserving DBMS may be combined with tools for data anonymization and privacy-preserving data mining in order to provide comprehensive platforms for supporting flexible and articulated privacy-preserving information management.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of P3P; this standard, even though has several limitations [19], is an important reference in the current privacy practices. Thus, it represens one of the starting points for development of privacy-preserving DBMS. In Section 3 we then discuss relevant requirements towards the development of privacy-preserving DBMS; in particular, we elaborate on some of the requirements arising from the support of P3P policies. We then survey some existing solutions and elaborate on some of those requirements in Section 4. We review some techniques of fine-grained access control and discuss some key challenges in Section 5, and we conclude the paper in Section 6.

2 Platform for Privacy Preferences (P3P)

The W3C's Platform for Privacy Preferences Project (P3P) [29] is one major effort to improve today's online privacy practices. P3P enables websites to encode their data-collection and data-use practices in a machine-readable XML format, known as P3P policies [12]. The W3C has also designed APPEL (A P3P Preference Exchange Language) [16], which allows users to specify their privacy preferences. Ideally, through the use of P3P and APPEL, a user's agent should be able to check a website's privacy policy against the user's privacy preferences, and automatically determine when the user's private information can be disclosed. In short, P3P and APPEL are designed to enable users to play an active role in controlling their private information. In this section, we provide a brief overview of P3P and APPEL and discuss some related issues.

2.1 Overview of P3P

Each P3P policy is specified by one `POLICY` element that includes the following major elements.

- One `ENTITY` element: identifies the legal entity making the representation of privacy practices contained in the policy.
- One `ACCESS` element: indicates whether the site allows users to access the various kind of information collected about them.
- One `DISPUTES-GROUP` element: contains one or more `DISPUTES` elements that describe dispute resolution procedures to be followed when disputes arise about a service's privacy practices.
- Zero or more `EXTENSION` elements: contain a website's self-defined extensions to the P3P specification.
- And one or more `STATEMENT` elements: describe data collection, use and storage. A `STATEMENT` element specifies the data (e.g. user's name) and the data categories (e.g. user's demographic data) being collected by the site, as well as the purposes, recipients and retention of that data.

```
<STATEMENT>
 <PURPOSE><admin required="opt-in"/></PURPOSE>
 <RECIPIENT><public/></RECIPIENT>
 <RETENTION><indefinitely/></RETENTION>
 <DATA-GROUP>
  <DATA ref="#user.home-info.postal"></DATA>
 </DATA-GROUP>
</STATEMENT>
```

```
stmt(
 purpose: {admin(opt-in)}
 recipient: {public}
 retention: {indefinitely}
 data: {#user.home-info.postal}
)
```

Fig. 1. An Example P3P Statement. The XML representation appears on the left side and a more succinct representation on the right side.

There are two kinds of P3P statements. The first kind contains the `NON-IDENTIFIABLE` element, which is used to indicate that either no information will be collected or information will be anonymized during collection. The second kind does not contain the `NON-IDENTIFIABLE` element; this is the commonly used one. For now, we will focus on the latter. A brief discussion of statements with `NON-IDENTIFIABLE` element is given later in this section.

Figure 1 provides an example of a P3P statement. Each such statement contains the following:

– One `PURPOSE` element, which describes for which purpose(s) the information will be used. It contains one or more pre-defined values such as current, admin, individual-analysis and historical. A purpose value can have an optional attribute ‘required’, which takes one of the following values: opt-in, opt-out, and always. The value ‘opt-in’ means that data may be used for this purpose only when the user affirmatively requests this use. The value ‘opt-out’ means that data may be used for this purpose unless the user requests that it not be used in this way. The value ‘always’ means that users cannot opt-in or opt-out of this use of their data. Therefore; in terms of strength of data usage, ‘always’ > ‘opt-out’ > ‘opt-in’. In Figure 1, `PURPOSE` is admin and the attribute ‘required’ takes the value opt-in.
– One `RECIPIENT` element, which describes with whom the collected information will be shared. It contains one or more pre-defined values such as ours, delivery and public. A recipient value can have an optional attribute ‘required’, which is similar to that of a `PURPOSE` element. In Figure 1, `RECIPIENT` is public.
– One `RETENTION` element, which describes for how long the collected information will be kept. It contains exactly one of the following pre-defined values: no-retention, stated-purpose, legal-requirement, business-practices and indefinitely. In Figure 1, the `RETENTION` value is indefinite.
– One or more `DATA-GROUP` elements, which specify what information will be collected and used. Each `DATA-GROUP` element contains one or more `DATA` elements. Each `DATA` element has two attributes. The mandatory attribute ‘ref’ identifies the data being collected. For example, ‘#user.home-info.telecom.telephone’ identifies a user’s home telephone number. The ‘optional’ attribute indicates whether or not the data collection is optional. A `DATA`

element may also contain a `CATEGORIES` element, which describes the kind of information this data item is, e.g., financial, demographic and health. In Figure 1, `DATA` is postal info.
- Zero or one `CONSEQUENCE` element, which contains human-readable contents that can be shown to users to explain the data usage practice's ramifications and why the usage is useful.

2.2 Issues in P3P

Since proposed, P3P has received broad attention from both industry and the research community, and has been gradually adopted by companies. On the other hand, the full deployment of P3P in enterprise information systems has raised many challenging questions. For example, P3P represents an enterprise's promise to users about its privacy practice. How can we ensure that an organization and its customers have a common understanding of these promises? P3P promises must be fulfilled in the services provided by enterprises. How can a company guarantee that its P3P policy is correctly enforced in those applications? We discuss some of the issues regarding the former question (privacy policy specification) in this section. The issues regarding the latter question (privacy policy enforcement) are discussed in the subsequent section.

The Lack of Formal Semantics in P3P. One major problem that hinders P3P adoption is that a P3P policy may be interpreted and represented differently by different user agents. Companies are thus reluctant to provide P3P policies on their websites, fearing that the policies may be misrepresented [11,25]. Quoting from CitiGroup's position paper [25], "The same P3P policy could be represented to users in ways that may be counter to each other as well as to the intent of the site." "... This results in legal and media risk for companies implementing P3P that needs to be addressed and resolved if P3P is to fulfill a very important need."

For instance, consider the statement in Figure 1. In the statement, the three components (purpose, recipient and retention) all refer to the same data item '#user.home-info.postal'; however, for the statement to have a precise meaning, one must also determine how these components interact. We consider two interpretations. In the first interpretation, all three components are related, i.e., the purpose, the recipient and the retention are about *one* data usage. In Figure 1, the postal information will be used for the admin purpose (technical support of the website and its computer system); the information will be shared with the public and will be stored indefinitely. For this statement, this interpretation seems counterintuitive, because there is no need to share the data with the public for the admin purpose. Furthermore, it is not clear whether this data usage is required or optional, since the 'required' attribute has the 'opt-in' value for purpose but the default 'always' value for recipient. The explanation for this statement, provided by one of the P3P architects [10], is that the data item '#user.home-info.postal' will always be collected and shared with the public. Additionally, if the user chooses to opt-in, their postal information will be used for the admin purpose. In other words, whether the individual's postal information will be shared with the public does not depend upon whether or not the information is used for the admin purpose.

This leads us to the second interpretation, in which purpose, recipient and retention are considered orthogonal. In this interpretation, a P3P statement specifies three relations: the purposes for which a data item will be used, the recipients with whom a data item will be shared, and how long the data item will be stored. Even though these relations are specified in the same statement, they are not necessarily about a single data usage. Given this *data-centric* interpretation, the following three P3P policies will have the same meaning in the sense that all relations contain a data component:

Example 1. Three P3P policies that have the same meaning.

```
Policy 1:
  stmt( data: {#user.home-info.telecom,
               #user.bdate(optional)},
        purpose: {individual-analysis,
                  telemarketing(opt-in)},
        recipient: {ours},
        retention: {stated-purpose} )
Policy 2:
  stmt( data: {#user.home-info.telecom,
               #user.bdate(optional)},
        purpose: {individual-analysis},
        recipient: {ours},
        retention: {stated-purpose})
  stmt( data: {#user.home-info.telecom,
               #user.bdate(optional)},
        purpose: {telemarketing(opt-in)},
        recipient: {ours},
        retention: {stated-purpose} )
Policy 3:
  stmt( data: {#user.home-info.telecom},
        purpose: {individual-analysis,
                  telemarketing(opt-in)},
        recipient: {ours},
        retention: {stated-purpose} )
  stmt( data: {#user.bdate(optional)},
        purpose: {individual-analysis,
                  telemarketing(opt-in)},
        recipient: {ours},
        retention: {stated-purpose} )
```

Part of this problem is caused by overlooking the need for a semantics in the initial design of P3P, leaving too much freedom for P3P policies to be misinterpreted and misrepresented by user agents. The fact that the same meaning may be encoded in several different ways makes it very difficult to correctly express privacy preferences in a syntax-based preference language such as APPEL. One representation can be accepted by a preference, but another representation could be rejected by the same preference.

Potential Semantic Inconsistencies in P3P Policies. In general, any combinations of the values for purpose, recipient and retention are allowed in P3P. However, in a practical setting, semantic dependencies arise naturally between these values, making some of the combinations invalid. A P3P policy using invalid combinations is thus semantically inconsistent. This problem has been recognized [9,24], and P3P's designers are beginning to address some of these conflicts [9]. Nonetheless, many places where potential conflicts may occur have not been previously identified. We now identify some additional classes of potential semantic inconsistencies in P3P.

– *A P3P policy may be inconsistent because multiple retention values apply to one data item.*
  P3P allows one data item to appear in multiple statements, which introduces a semantic problem. Recall that in each P3P statement, only one retention value can be specified, even though multiple purposes and recipients can be used. The rationale behind this is that retention values are mutually exclusive, i.e., two retention values conflict with each other. For instance, *no-retention* means that "Information is not retained for more than a brief period of time necessary to make use of it during the course of a single online interaction" [12]. And *indefinitely* means that "Information is retained for an indeterminate period of time" [12]. One data item cannot have both retention values. However, allowing one data item to appear in multiple statements makes it possible for multiple retention values to apply to one data item.
– *A statement may have conflicting purposes and retention values.*
  Consider a statement in a P3P policy that collects users' postal information for the purpose *historical* with retention *no-retention*. Clearly, if the postal information is going to be "... archived or stored for the purpose of preserving social history ...", as described by the *historical* purpose, it will conflict with *no-retention*, which requires that the collected information "... MUST NOT be logged, archived or otherwise stored" [12].
– *A statement may have conflicting purposes and recipients.*
  Consider a statement that includes all the purpose values (e.g., history, admin, telemarketing, individual-analysis, etc.) but only the recipient value *delivery* (delivery services). This does not make sense as one would expect that at least *ours* should be included in the recipients.
– *A statement may have conflicting purposes and data items.*
  Certain purposes imply the collection and usage of some data items. This has been recognized by the P3P designers and reflected in the guidelines for designing P3P user agents [9]. For example, suppose a statement contains purpose contact but does not collect any information from the categories physical and online. Then the statement is inconsistent because, in order to contact a user, "the initiator of the contact would possess a data element identifying the individual . . . . This would presuppose elements contained by one of the above categories" [9].

All semantic inconsistency instances must be identified and specified in the P3P specification. Completion of this work requires a detailed analysis of the vocabulary, ideally by the individuals who design and use these vocabularies.

Dealing with P3P Statements Having the `NON-IDENTIFIABLE` Element. A `STATEMENT` element in a P3P policy may optionally contain the `NON-IDENTIFIABLE` element, which "signifies that either no data is collected (including Web logs), or that the organization collecting the data will anonymize the data referenced in the enclosing `STATEMENT`" [12]. We call such statements non-identifiable statements.

From the above description, we see that the `NON-IDENTIFIABLE` element is used for two unrelated purposes in P3P. We argue that using the `NON-IDENTIFIABLE` element to signify that no data is collected is inappropriate. Intuitively, if a statement with the `NON-IDENTIFIABLE` element contains the `DATA-GROUP` element, it means that the data collected in this statement is anonymized. If such a statement does not have the `DATA-GROUP` element, it means that no data is collected. However, this statement is meaningless when the policy contains other statements that collect and use data. In general, the fact that a policy does not collect any data should not be specified at the level of a `STATEMENT` element; instead, it should be specified at the level of a `POLICY` statement. For instance, it seems more appropriate to use a separate sub-element (or an attribute) for the `POLICY` element to denote that a policy collects no data.

Another issue that arises from having the `NON-IDENTIFIABLE` element is that a data item may appear both in normal statements and in non-identifiable statements. In this situation, it is not clear whether the data item is anonymized upon collection. According to Cranor [10], it may be possible that: "a company keeps two different unlinkable databases and the data is anonymized in one but not the other."

The most straightforward way seems to annotate an anonymized data item so that it is different from a normal data item, e.g., one may use '#user.home-info' to denote a normal data item and '#@.user.home-info' to denote an anonymized version of the same data.

2.3 An Overview of APPEL

Privacy preferences are expressed as a *ruleset* in APPEL. A ruleset is an ordered set of rules. An APPEL evaluator evaluates a ruleset against a P3P policy.[1] A rule includes the following two parts:

– A behavior, which specifies the action to be taken if the rule fires. It can be *request*, implying that a P3P policy conforms to preferences specified in the rule body and should be accepted. We call this an *accept* rule. It can be

[1] The APPEL specification allows arbitrary XML elements not related to P3P to be included together with the P3P policy for evaluation. Since the users may not even know about them, it is not clear how they write preferences dealing with them. In this paper, we assume that only a P3P policy is evaluated against a ruleset.

block, implying that a P3P policy violates the user's privacy preferences and should be rejected. We call this a *reject* rule. It can also be *limit*, which can be interpreted as accept with warning.
- A number of expressions, which follow the XML structure in P3P policies. An expression may contain subexpressions. An expression is evaluated to TRUE or FALSE by matching (recursively) against a target XML element. A rule *fires* if the expressions in the rule evaluate to TRUE.

Every APPEL expression has a *connective* attribute that defines the logical operators between its subexpressions and subelements of the target XML element to be matched against. A connective can be: *or*, *and*, *non-or*, *non-and*, *or-exact* and *and-exact*. The default connective is *and*, which means that all subexpressions must match against some subelements in the target XML element, but the target element may contain subelements that do not match any subexpression. The connective *and-exact* further requires that any subelement in the target match one subexpression. The *or* connective means that at least one subexpression matches a subelement of the target. The *or-exact* connective further requires that all subelements in the target matches some subexpressions.

When evaluating a ruleset against a P3P policy, each rule in a ruleset is evaluated in the order in which it appears. Once a rule evaluates to true, the corresponding behavior is returned.

2.4 Issues in EPPAL

Many authors have noted that APPEL is complex and problematic [2,13,14,30]. In this section, we analyze APPEL's pitfalls and the rationales for some of the design decisions embedded in APPEL. The main objective for our analysis is to ensure we design a preference language that avoids the APPEL's pitfalls but preserves the desirable functionalities in APPEL. The following subsections examine the pitfalls and limitations of APPEL.

Semantic Inconsistencies of APPEL. Because of APPEL's syntax-based design, two P3P policies that have the same semantic meanings but are expressed in syntactically different ways may be treated differently by one APPEL ruleset. This deficiency in APPEL has been identified before [15]. We now show an example APPEL rule.

```
<appel:RULE behavior="block">
 <p3p:POLICY>
  <p3p:STATEMENT>
   <p3p:DATA-GROUP>
    <p3p:DATA ref=
     "#user.home-info.telecom"/>
    <p3p:DATA ref="#user.bdate"/>
   </p3p:DATA-GROUP>
  </p3p:STATEMENT>
 </p3p:POLICY>
</appel:RULE>
```

This is a reject rule, since the behavior (on line 1) is “block”. The body of this rule has one expression (lines 2-10) for matching a P3P policy. This expression contains one subexpression (lines 3-9), which in turns contain one subexpression (lines 4-8). The outmost expression (lines 2-10) uses the default **and** directive; therefore, it matches a P3P policy only if the policy contains at least one statement that matches the enclosed expression (lines 3-9). Overall, this APPEL rule says that a P3P policy will be rejected if it contains a `STATEMENT` element that mentions both the user’s birthday and the user’s home telephone number.

This rule rejects Policies 1 and 2 in Example 2, but not Policy 3. In Policy 3, the two data items are mentioned in different statements and no statement mentions both data items. This is clearly undesirable, as the three statements have the same semantics. This problem is a direct consequence of that fact that APPEL is designed to query the representation of a P3P policy, rather the semantics of the policy.

The same problem exists in XPref [2], since it is also syntax-based.

The Subtlety of APPEL’s Connectives. The meaning of an APPEL rule depends very much on the connective used in the expressions. However, the connectives are difficult to understand and use. The APPEL designers made mistakes using them in the first example in the APPEL specification [16]. Consider the following example taken from [16].

Example 2. The user does not mind revealing click-stream and user agent information to sites that collect no other information. However, she insists that the service provides some form of assurance.

The APPEL rule used in [16] for the above example is as follows:

```
01 <appel:RULE behavior="request"
02   description="clickstream okay">
03 <p3p:POLICY>
04  <p3p:STATEMENT>
05   <p3p:DATA-GROUP
06     appel:connective="or-exact">
07    <p3p:DATA
08     ref="#dynamic.http.useragent"/>
09    <p3p:DATA
10     ref="#dynamic.clickstream.server"/>
11   </p3p:DATA-GROUP>
12  </p3p:STATEMENT>
13  <p3p:DISPUTES-GROUP>
14   <p3p:DISPUTES service="*"/>
15  </p3p:DISPUTES-GROUP>
16 </p3p:POLICY>
17 </appel:RULE>
```

The above APPEL rule is an accept rule; its body has one outmost expression (lines 3-16) to match a P3P policy. The expression contains two subexpressions, matching different elements in a policy. The expression denoted by the p3p:POLICY element (lines 3–16) does not have the 'connective' attribute; therefore, the default **and** connective is used, which means that as long as the two included expressions, i.e., p3p:STATEMENT (lines 4-12) and p3p:DISPUTES-GROUP (lines 13-15), match some parts in the P3P policy, the rule accepts the policy. The expression denoted by p3p:DATA-GROUP uses the **or-exact** connective, it matches a DATA-GROUP element if the DATA elements contained in the element is a non-empty subset of {#dynamic.http.useragent, #dynamic.clickstream.server}.

Overall, this rule means that a P3P policy will be accepted if it contains a STATEMENT element that mentions only the two specified data items and a DISPUTES-GROUP element.

Observe that this rule does not express the preference, as it does not take into consideration the fact that a P3P policy can have multiple statements. Subsequently, a policy that only mentions "#dynamic.http.useragent" in the first statement can be accepted by the rule, even if the next statement collects and uses other user data.

One may try to fix this problem by using the **and-exact** connective on line 2, which means that each element contained in the P3P policy must match one of the expressions within the APPEL rule. For such a rule to work as intended, the p3p:POLICY expression (lines 3–16) must contain two additional sub-expressions: p3p:ENTITY and p3p:ACCESS. Otherwise, no P3P policy will be accepted because of the existence of ENTITY and ACCESS elements. However, this fix still does not work. A P3P policy may optionally contain an EXTENSION element. Even when such a policy collects only '#dynamic.http.useragent' and '#dynamic.clickstream.server', it will not be accepted by the above rule, due to the semantics of **and-exact**. On the other hand, including a sub-expression p3p:EXTENTION cannot fix the problem, as a policy without extensions will not be accepted in this case.

Another approach to fix the problem is to use the **or-exact** connective on line 3 and to include subexpressions for p3p:ENTITY, p3p:ACCESS and p3p:EXTENSION. Recall that the **or-exact** connective means that all elements in the policy must match some subexpressions, but not every subexpression is required to match some element in the policy. However, this means that the p3p:DISPUTES-GROUP element also becomes optional. A P3P policy that does not have the DISPUTES-GROUP element will also be accepted. This is not the preference described in Example 2.

In fact, as far as we can see, there is no way to correctly specify the preference in Example 2 in APPEL. One source of difficulty is that one has to intermingle statements and other aspects (e.g., dispute procedures) of a P3P policy in a preference rule. This is caused by APPEL's syntactic nature and the fact that statements and dispute procedures are all immediate children of a POLICY element. If conditions about data usages and other aspects of P3P policies may be specified separately, it is possible to specify the conditions on data usage in Example 2 using the **or-exact** connective.

3 Requirements Towards the Development of Privacy-Preserving DBMS

Languages for specification of privacy promises, such as P3P, represent only one of the components in a comprehensive solution to privacy [3]. It is crucial that once data are collected, privacy promises be enforced by the information systems managing them. Because in today information systems, data are in most cases managed by DBMS, the development of DBMS properly equipped for the enforcement of privacy promises and of other privacy policies is crucial. Here we discuss a set of requirements towards the development of such DBMS. Some of those requirements, as the support for purpose meta-data and privacy obligations, derive directly from P3P. Other requirements are not directly related to P3P; however, they are crucial for the development of DBMS able to support a wide range of privacy policies, going beyond the ones strictly related to P3P. In the discussion, we use the terms subject to denote the active entities, trying to gain accesses to the data, and subjects to denote the passive entities, that are to be protected.

Support for Rich Privacy Related Meta-data. An important characteristic of P3P is that very often privacy promises, that is, statements specifying the use of the data by the party collecting them, include the specification of the intended use of the data by the collecting party as well as other information. Examples of this additional information are how long the data will be kept and possible actions that are to be executed whenever a subject accesses the data. Supporting this additional information calls for the need of privacy-specific meta-data that should be associated with the data, stored in the database together with the data, and send with the data whenever the data flow to other parties in the system. Metadata should be associated with the data according to a range of possible granularities. For example in a relational database, one should be able to associate specific metadata with an entire table, with a single tuple, or even with a column within a single tuple. Such flexibility should not however affect the performance; thus we need to develop highly efficient techniques for managing these metadata in particular when dealing with query executions. Query executions may need to take into account the contents of such metadata in order to filter out from the data to be returned, the data that cannot be accessed because of privacy constraints.

Support for Expressive Attribute-Based Descriptions of Subjects. We see an increasing trend towards the development of access control models that relies on information concerning subjects. Examples of such models are represented by trust negotiation systems [4,18], that use credentials certifying relevant properties of subjects. Such access control models are crucial in the context of privacy because they provide a high-level mechanism able to support a very detailed specification of the conditions that subjects must verify in order to access data. As such, fine-grained privacy-preserving access control policies can be supported. They also make it easy to formulate and maintain privacy policies and verify their correctness. Moreover, such high-level models can provide better support for interoperability because they can, for example, easily integrate with SAML assertions. However, current database technology is very poor in the representation of subjects. At the best current DBMS provide support for roles in the context of the well-known role-based access control (RBAC) model [3]. However, apart from this, DBMS do not provide the possibility of specifying application-dependent user profiles for use in access control and privacy enforcement. It can be argued that such profiles should perhaps be built on top of the DBMS or even be supported externally. However, in such a case, it is not clear how efficient access control and privacy enforcement could be supported. It also important to notice that RBAC does not support subject attributes. Extensions of RBAC models supporting such a feature should be devised.

Support for Obligations. Obligations specify privacy-related actions that are to be executed upon data accesses for certain purposes. There is a large variety of actions that can be undertaken, including modifications to the data, deletion of the data, notifications of data access to the individual to whom the data are related or to other individuals, insertion of records into privacy logs. These obligations should be possibly executed, or at least initiated by, the DBMS because their execution is tightly coupled with data accesses. An important issue here is the development of expressive languages supporting the specification obligations, and analysis tools to verify the correctness and consistency of obligations. A viable technology to support obligations is represented by trigger mechanisms, currently available in all commercial DBMS. The main question is however whether current trigger languages are adequate to support the specification of obligations.

Fine-Grained Access Control to Data. The availability of a fine-grained access control mechanism is an important requirement of a comprehensive solution to privacy. Conventional view mechanisms, the only available mechanism able to support in some ways a very fine granularity in access control, have several shortcomings. A naive solution to enforce fine-grained authorizations would require specifying a view for each tuple or subset of a tuple that are to be protected. Moreover, because access control policies are often different for different users, the number of views would further increase. Furthermore, applications programs would have to code different interfaces for each user, or group of users, because queries and other data management commands would need to use for each user, or group of users, the correct view. Modifications to access control policies would also require creation of new views with consequent modifications to application programs. Alternative approaches that address some of those issues have been proposed that are based on the idea that queries are written against base tables and then automatically re-written by the system against the view available to the user. These approaches do not require to code different interfaces for different users, and thus address on of the main problems in the use of conventional view mechanisms. However, they introduce other problems, such as inconsistencies between what the user expects to see and what the systems returns; in some cases, they return incorrect results to queries rather than rejecting them as unauthorized. Different solutions thus need to be investigated. These solutions must not only address the specification of fine-grained access control policies but also their efficient implementation in current DBMS.

Privacy-Preserving Information Flow. In many organizations, data flow across different domains. It is thus important that privacy policies related to data "stick" with the data when these data move within an organization or across organizations. This is crucial to assure that if data have been collected under a given privacy promise from an individual, this promise is enforced also when data are passed to parties different from the party that have initially collected them. Information flow has been extensively investigated in the past in the area of multi-level secure databases. An important issue is to revisit such theory and possibly extend it for application in the context of privacy.

Protection from Insider Threats. An important problem that so far has not received much attention is related to the misuse of privileges by legitimate subjects. Most research in the past has been devoted to protection from intrusions by subjects external to the systems, against which technologies like firewalls can provide a certain degree of protection. However, such approaches are not effective against users that are inside the firewalls. A possible technique that can be employed to start addressing such problem is based on the use of subject access profiling techniques. Once the profile of the legitimate accesses has been defined, such profile can be used to detect behavior that is different. Developing such an approach requires however investigating several issues, such as specific machine learning techniques to use, efficiency and scalability. Also, the collection of user profiles may in turn introduce more privacy problems, because users of the system may be sensible to the fact that all their actions are being monitored.

In the rest of this paper, we elaborate on some of those requirements and discuss possible solutions.

4 Purpose Management and Access Control

In this section, we present three privacy-centric access control models from recent literatures. Prior to proceed, we note that privacy protection cannot be easily achieved by traditional access control models. The key difficulty comes from the fact that privacy-oriented access control models are mainly concerned with which data object is used for which purpose(s) (i.e., the intent(s) of data usage), rather than which user is performing which action on which data object as in traditional access control models. Another difficulty of privacy protection is that the comfort level of privacy varies from individual to individual, which requires access control be fine-grained. Thus, the main challenge of privacy protecting access control is to provide access control based on the notion of purpose, incorporating data subjects' preferences if necessary, at the most fine-grained level. In the remainder of this section we discuss two approaches to the management of purpose information and their use in access control.

4.1 Purpose Based Access Control

Our work in [5,6] presents a comprehensive approach to purpose management, which is the fundamental building block on which purpose-based access control can be developed. Our approach is based on intended purposes, which specify the intended usage of data, and access purposes, which specify the purposes for which a given data element is accessed. We also introduce the notion of purpose compliance, which is the basis for verifying that the purpose of a data access with the intended purposes of the data.

Another important issue addressed in this work is the data labeling scheme; that is, how data are associated with intended purposes. We address this issue in the context of relational data model. The main issue here is the granularity of data labeling. We propose four different labeling schemes, each providing a different granularity. We also exploit query modification techniques to support data filtering based on purpose information.

Evidently, how the system determines the purpose of an access request is also crucial as the access decision is made directly based on the access purpose. To address this issue, we present a possible method for determining access purposes. In our approach, users are required to state their access purposes along with the data access requests, and the system validates the stated access purposes by ensuring that the users are indeed allowed to access data for the particular purposes.

Definition of Purposes. In privacy protecting access control models, the notion of purpose plays a central role as the purpose is the basic concept on which access decisions are made. In order to simplify the management, purposes are organized according to a hierarchical structure based on the principles of generalization and specialization, which is appropriate in common business environments. Figure 2 gives an example of purpose tree, where each node is represented with the conceptual name of a purpose.

Intuitively, an access to a specific data item is allowed if the purposes allowed by privacy policies for the data include or imply the purpose for accessing the data. We refer to purposes associated with data and thus regulating data accesses as *Intended Purposes*, and to purposes for accessing data as *Access Purposes.*

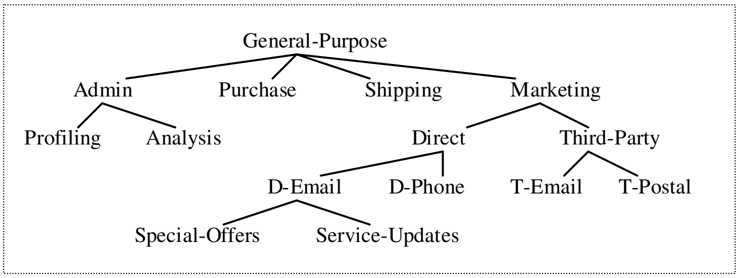


Fig. 2. Purpose Tree (From [5])

Intended purposes can be viewed as brief summaries of privacy policies for data, stating for which purposes data can be accessed. When an access to data is requested, the access purpose is checked against the intended purposes for the data.

In our model intended purposes support both positive and negative privacy policies. An intended purpose consists of two components: *Allowed Intended Purposes* and *Prohibited Intended Purposes.* This structure provides greater flexibility to the access control model. Moreover, by using prohibited intended purposes, one can guarantee that data accesses for particular purposes are never allowed. Conflicts between the allowed intended purposes and the prohibited intended purposes for the same data item are resolved by applying the denial-takes-precedence policy where prohibited intended purposes override allowed intended purposes.

An access purpose is the purpose of a particular data access, which is determined or validated by the system when the data access is requested. Thus, an access decision can be made based on the relationship between the access purpose and the intended purposes of data. That is, an access is granted if the access purpose is entailed by the allowed intended purposes and not entailed by the prohibited intended purposes; in this case we say the access purpose is *compliant to* the intended purpose. The access is denied if any of these two conditions fails; we then say that the access purpose is *not compliant to* the intended purpose.

Data Labeling Model. In order to build an access control model based on the notion of purpose, we must consider a specific data model and based on this model devise a proper labeling scheme. A major question here is at what level of granularity intended purposes are associated with data.

It is clear that in order to make the best use of data while at the same time ensure that data providers feel comfortable, the labeling model should allow the assignments of intended purposes with data at the most fine-grained level. That is, we should be able to assign an intended purpose to each data element in every tuple; e.g., for each attribute and for each data provider. It is also possible that a data provider allows or prohibits access to his/her entire record (i.e., address), not to individual sub-elements (i.e., street, city, state, etc.). This means that it is not always necessary to associate each data element with an intended purpose. However, intended purpose for the address information can vary depending on each individual. To address these concerns, the labeling model should allow the assignment of intended purpose to each tuple of a relation.

However, this most fine-grained approach is not efficient in terms of storage and is not always necessary. For instance, it is possible that there exists some information for which corresponding privacy policies are mandated by enterprises or by laws; i.e., data providers do not have a choice to opt-out from the required intended purposes. In such cases, the data elements in each column in the relation have the identical intended purpose. Thus, in order to avoid any redundant labeling, intended purposes should be assigned to each attribute of a relation using an auxiliary table, e.g., *privacy policy table.* Furthermore, it is possible that the intended purposes of every attribute in a relation be identical.

Such cases occur when information in a relation is meaningful as a whole tuple, but individual elements or tuples do not have any usefulness. In this case, the intended purposes are assigned to the entire relation by using a single entry in the privacy policy table.

In summary, in order to provide privacy protection in a storage efficient way, intended purpose should be assigned to every relation, to every tuple in every relation, to every attribute in every relation, or to every data element in every relation.

Access Control Using Query Modification. Privacy-preserving access control mechanisms must ensure that a query result contains only the data items that are allowed for the access purpose of the query. In other words, the system must check the intended purpose of each data element accessed by the query and filter out its value if the access purpose is not compliant with the intended purpose of the data element. In our approach, this fine-grained access control is achieved using query modification [26]. Our query modification algorithm is outlined in Figure 3. Note that this method is invoked only if the access purpose of the query is verified to be acceptable by the validate function. If the access purpose is not acceptable, then the query is rejected without further being processed.

In Lines 7 and 9 the compliance checks for relations with the relation- or attribute-based labeling schemes are executed statically by the query modification method. On the other hand, the compliance checks for relations with the tuple- or element-based labeling schemes are performed during query processing by the predicates which are added by the query modification algorithm (Lines 15 and 17).

The query modification algorithm checks both the attributes referenced in the projection list and the attributes referenced in predicates (Line 3). As the attributes in the projection list determine what data items will be included in the result relation of a query, it may seem enough to enforce privacy policy based only on the attributes in the projection list. However, the result of a query also depends on the predicates, and not enforcing privacy constraints on the predicates may introduce inference channels. For example, consider the following query:

```
SELECT name
FROM Customer
WHERE income > 100000
FOR Third-Party.
```

Suppose that according to the established privacy policies, *name* can be accessed for the purpose of *Third-Party*, but *income* is prohibited for this purpose. If the privacy constraint is not enforced on the predicates, this query will return a record containing the names of customers whose income is greater than 100,000. This is highly undesirable as this result implicitly conveys information about the customers' income. Note that if the privacy policy is enforced at the predicate level, such inference channels cannot be created.

```
Comp_Check (Number ap, Number aip, Number pip)
Returns Boolean
1.    if (ap & pip) $\neq$ 0 then
2.       return False;
3.    else if (ap & aip) = 0 then
4.       return False;
5.    end if;
6.    return True;

Modifying_Query (Query Q)
Returns a modified privacy-preserving query Q'
1.    Let $R_1$, ..., $R_n$ be the relations referenced by Q
2.    Let P be the predicates in WHERE clause of Q
3.    Let $a_1$, ..., $a_m$ be the attributes referenced in both the projection list and P
4.    Let AP be the access purpose encoding of Q
5.    for each $R_i$ where i = 1, ..., n do
6.       if ($R_i$ is relation-based labeling AND Comp_Check (AP, $R_i$.aip, $R_i$.pip) = False) then
7.          return ILLEGAL-QUERY;
8.       else if $R_i$ is attribute-based labeling then
9.          for each $a_j$ which belongs to $R_i$ do
10.            if Comp_Check (AP, $a_j$.aip, $a_j$.pip) = False then
11.               return ILLEGAL-QUERY;
12.            end if;
13.         end for;
14.      else if $R_i$ is tuple-based labeling then
15.         add ' AND Comp_Check (AP, $R_i$_aip, $R_i$_pip)' to P ;
16.      else if $R_i$ is element-based labeling then
17.         for each $a_j$ which belongs to $R_i$ do
18.            add ' AND Comp_Check (AP, $a_j$_aip, $a_j$_pip)' to P;
19.         end for;
20.      else // $R_i$ is a relation without labeling
21.         do nothing;
22.      end if;
23.   end for;
24.   return Q with modified P;
```

Fig. 3. Query Modification Algorithm (From [5])

Notice that the provided algorithm filters out a tuple if any of its elements that are accessed is prohibited with respect to the given access purpose. For instance, consider the following query:

```
SELECT name, phone
FROM Customer
FOR Marketing.
```

Suppose there is a customer record of which the *name* is allowed for marketing, but the *phone* is prohibited for this purpose. Then our algorithm excludes the record from the query result. We note that in the environments where partially incomplete information is acceptable, the query modification algorithm can be easily modified to mask prohibited values with null values using the case expression in SQL.

Access Purpose Determination. An access purpose is the reason for accessing a data item, and it must be determined by the system when a data access is requested. Evidently, how the system determines the purpose of an access request is crucial as the access decision is made directly based on the access purpose. There are many possible methods for determining access purposes. First, the users can be required to state their access purpose(s) along with the requests for data access. Even though this method is simple, it requires complete trust on the users and the overall privacy that the system is able to provide entirely relies on the users' trustworthiness. Another possible method is to register each application or stored-procedure with an access purpose. As applications or stored-procedures have limited capabilities and can perform only specific tasks, it can be ensured that data users use them to carry out only certain actions with the associated access purpose. This method, however, cannot be used for complex stored-procedures or applications as they may access various data for multiple purposes. Lastly, the access purposes can be dynamically determined, based on the current context of the system. For example, suppose an employee in the shipping department is requesting to access the address of a customer by using a particular application in a normal business hour. From this context (i.e., the job function, the nature of data to be accessed, the application identification, and the time of the request), the system can reasonably infer that the purpose of the data access must be shipping.

In our work [6], users are required to state their access purposes along with the data access requests, and the system validates the stated access purposes by ensuring that the users are indeed allowed to access data for the particular purposes. To facilitate the validation process, each user is granted authorizations for a set of access purposes, and an authorization for an access purpose permits users to access data with the particular purpose. To ease the management of access purpose authorizations, users are granted authorizations through their roles. This method has a great deployment advantage as many systems are already using RBAC mechanisms for the management of access permissions. This approach is also reasonable as access purposes can be granted to the tasks or functionalities over which roles are defined within an organization. However, using an RBAC mechanism for the management of both access permissions and access purposes may increase the complexity of the role engineering tasks. To address this problem, we introduce a simple extension to RBAC. An important feature of our approach is that by integrating RBAC with attribute-based control, our extension simplifies the role administration and also provides increased flexibility. For more detailed information, users are directed to [6].

4.2 Limiting Disclosure in Hippocratic Databases

LeFevre et al. [17] presented a database architecture for enforcing limited disclosure [2] expressed by privacy polices, which is illustrated by Figure 4. In this section, we briefly review each component of their architecture.

[2] Limited disclosure is one of data privacy principles, which states that data subjects have control over who is allowed to see their personal information and for what purpose [1].

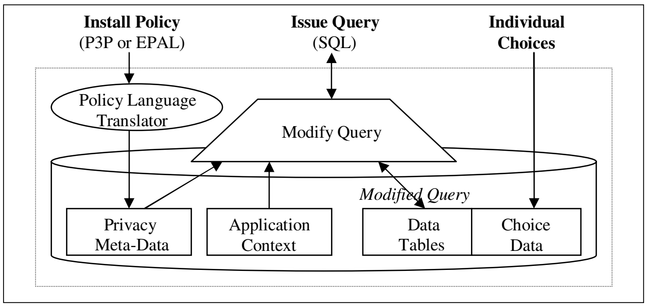


Fig. 4. Implementation architecture overview (From [17])

Privacy Policy Meta-language. A privacy policy, ⟨*data, purpose-recipient pair, condition*⟩ [3], describes to whom the data may be disclosed (i.e., recipients), how the data may be used (i.e., the purposes), and in what specific circumstances the data may be disclosed (i.e., conditions). For instance, a rule ⟨*address, solicitation-charity, optin=yes*⟩ requires that a data subject's address information can be accessed to a charity organization for the solicitation purpose if the subject has explicitly consented to this disclosure. Note that a condition predicate may refer to the data table T as well as any other data tables and the context environment variables (e.g., \$USERID). For example, a condition to govern the disclosure of patient data to nurses such that, for treatment, nurses may only see the medical history of patients assigned to the same floor can be expressed as follows:

```
EXISTS ( SELECT NurseID
    FROM Nurses
    WHERE Patients.floor = Nurses.floor
    AND $USERID = Nurses.NurseID)
```

Limited Disclosure Models. For enforcing cell-level limited disclosure, two models are introduced: *table semantics* and *query semantics* [17]. The table semantics model conceptually defines a view of each data table for each purpose-recipient pair, based on privacy policies. Then queries are evaluated against these views. On the other hand, the query semantics model takes the query into account when enforcing disclosure. Despite this subtle difference, the effect of them is the same in that both models mask prohibited values using *null* value. Below we provide the definitions of these two models, taken verbatim from [17].

– (***Table Semantics***) Let T be a table with n data columns, and let K be the set of columns that constitute the primary key of T. For a given purpose-recipient par P_j, the table T, seen as T_{P_j}, is defined as follows:

[3] It is assumed that privacy policies in P3P or EPAL are translated to this form and stored, prior to access control.

$$\{r \mid \exists\, t \in T \wedge \forall\, i,\, 1 \leq i \leq n$$
$$(r[i] = t[i] \text{ if } \mathit{eval}(t[i,j]) = \text{true},^{4}$$
$$r[i] = \text{null otherwise})$$
$$\wedge\, r[K] \text{ nonenull}\}$$

– (***Query Semantics***) Consider a query Q that is issued on behalf of some purpose-recipient pair P_j and that refers to table T. Query Semantics is enforced as follows:
  1. Every table T in the FROM clause is replaced by T_{P_j}, defined as follows:
  $$\{r \mid \exists\, t \in T \wedge \forall\, i,\, 1 \leq i \leq n$$
  $$(r[i] = t[i] \text{ if } \mathit{eval}(t[i,j]) = \text{true},$$
  $$r[i] = \text{null otherwise})\}$$
  2. Result tuples that are null in all columns of Q are discarded.

Privacy Policy Storage. The privacy policies (i.e., disclosure rules) are stored in the database as the *privacy meta-data*. These meta-data consists of *policy rules table* and *conditions table*. The policy rules table stores a policy rule as a tuple of ⟨RuleID, PolicyID, Purpose, Recipient, Table, Column, CondID⟩. The condition table stores conditional predicates each of which is identified by a CondID. For instance, a tuple ⟨r1, p1, p, r, t, d, c⟩ in the policy rules table means that the data in d of t is available to the user r for the purpose of p if the predicate identified by c in the conditions table satisfies.

Query Modification. Incoming queries are augmented with case statements [5] to enforce the disclosure rules and conditions specified by the privacy meta-data. For instance, consider the previous example, where nurses may only see the medical history of patients assigned to the same floor for treatment. Suppose a nurse issues a query

```
SELECT history FROM Patients
```

for the purpose of treatment. Then the query is rewritten to enforce the disclosure rules as follows:

```
SELECT
CASE WHEN EXIST
     (SELECT NurseID
      FROM Nurses
      WHERE Patients.floor = Nurses.floor
           AND $USERID = Nurses.NurseID)
THEN history ELSE null END
FROM Patients
```

[4] $\mathit{eval}(t[i,j])$ denotes the boolean result of evaluating the condition that governs the disclosure of data column i in the current row of T to the purpose-recipient pair j.

[5] In [17], a query modification algorithm using outer-joins is also presented.

Observe that the modified query replaces the prohibited history data with *null* values. Note also that the modified query accesses an additional patient data (i.e, "floor"). If this information is governed by another disclosure rule, then such a rule must be also enforced by checking the rule and adding the corresponding condition to the modified query.

5 Generalized Fine-Grained Access Control Models for Database Systems

In recent years, because of privacy requirements, we have seen a growing interest in fine-grained access control in databases; e.g., the VPD in Oracle, parameterized views in DB2 and the Hippocratic databases work. A motivation for fine-grained access control is to move access control from applications to databases, so that they cannot be bypassed and therefore high-assurance privacy can be achieved.

Fine-grained access control policies may be specified according to two different approaches. One approach, that we call view-based approach, uses views or parameterized views. In this approach, what a user is authorized to see is given by a set of views. Unlike in standard SQL, where the user is given access only to the views, the user is given access to the base table; however, the user is limited to access only those parts of the table that is authorized by the views. The other approach uses labeling, where each data element (e.g., a cell or a record) is labeled with information determining whether this element is authorized for a particular user (or a particular query). Purpose based access control discussed in Section 4 is an example of this latter approach. In this section, we review some existing works of the view-based approach and discuss the key challenges posed by fine-grained access control.

5.1 View-Based Approach

In the view-based approach, a user is given a set of views which then represents the parts of the database that are accessible to the user. This differs from the view mechanism in SQL in that users issue their queries against the base tables directly, not using the given views. In this section, we briefly review two works that exemplify the view-based approach.

Access Control in INGRES. Stonebraker and Wong [26] introduced query modification as a part of the access control system in INGRES. The basic idea of query modification is that before being processed, user queries are modified transparently to ensure that users do not see more than what they are authorized to see. In their scheme, an access permission for a user is specified and stored as a view. Thus, each user is associated with a set of views which defines a permitted view of database for the user. When the user issues a query, the query modification algorithm searches for the views that are related to the query; i.e., the views whose attributes contain the attributes addressed by the query [6]. Then the qualifications (i.e., conditions in the WHERE clause) of such views are conjuncted with the qualification of the original query. For instance, consider a table *Employee* with attributes *name*, *department*, *salary.* Suppose a user Smith is allowed to see only information on himself in the *Employee* table. This restrictions is expressed [7] as:

```
SELECT name, department, salary
FROM Employee
WHERE name = ‘Smith’
```

Now suppose Smith wants to find out the salary information on the employees in the *Accounting* department and issues the following query:

```
SELECT name, salary
FROM Employee
WHERE department = ‘Accounting’
```

As the attributes of the given view contain the attributes of the query, the restriction is applied to the query and the query is automatically modified into:

```
SELECT name, salary
FROM Employee
WHERE department = ‘Accounting’ AND name = ‘Smith’
```

Thus, the modified query effectively limits the query result to the information of Smith only. Note that once a query is modified, the modified query can be processed without further access control; that is, the modified query is guaranteed never to violate any access restriction placed on the query issuer.

Motro's Approach. In [21], Motro points out some limitations of the query modification algorithm in INGRES. The main drawback pointed out is that in some cases the algorithm returns less than what the user is actually allowed to see. For instance, consider a relation A with attributes a_1, a_2 and a_3, and assume that a user is given a view permitting her to access the tuples of a_1 and a_2 as long as a condition C is satisfied. Then if the user tries to retrieve the tuples of a_1, a_2, or a_1 and a_2, the access restriction is applied to the query, and only the tuples that satisfy the condition C will be returned. However, when the user tries to retrieve the tuples of all a_1, a_2 and a_3, then the query will be denied as the view given to the user does not contain the all attributes of the query. To address this problem, Motro proposed an alternative technique. Similar to the scheme used in INGRES, views are used to represent statements of access

[6] Note that the attributes of a query include the attributes in both the SELECT clause and the WHERE clause.

[7] While QUEL [20] is used as the query language in the original paper, we use SQL for the convenience of readers.

permissions. However, the main difference is that granting a user permission to access a set of views $V = \{v_1, \ldots, v_m\}$ in Motro's scheme implies that permission is also granted to access any view derived from V. Thus, when the user issues a query Q, which is also a view, the system accepts Q if Q is a view that can be derived from V. In addition, if Q is not a view of V, but any subview of Q is, then the subview is accepted; that is, in the previous example where the user tries to access all a_1, a_2 and a_3, the tuples of a_1 and a_2 that satisfy C will be returned. In [21], this is accomplished as follows. The views that permit access are stored as "meta-relations" in the database. When a query is presented to the database, the query is performed both on the meta-relations and the actual relations. As applying the query on the meta-relations results in a view that is accessible to the user, this result is then applied to the actual query result, yielding the final result. For complete description and examples, readers are referred to [21].

Although the approach of Motro provides more flexibility to access control, his technique also poses some subtle problems. First, his algorithm may result in accepting multiple disjoint subviews of a query. However, it is not trivial how these subviews are presented to users. Another shortcoming of his approach is that the logical structure of actual query result may be different from the expected structure; that is, a query retrieving three attributes may return tuples with only two attributes. This is highly undesirable or unacceptable to most database applications.

5.2 Virtual Private Database (VPD) in Oracle

Since the release of *Oracle8i*, the Virtual Private Database (VPD) has been included as one of major access control components in Oracle database systems. The VPD, defined as "the aggregation of server-enforced, fine-grained access control" [22], provides a way to limit data access at the row level. Surely, it is possible to support row level access control using the view mechanism. However, the view mechanism has several limitations as fine-grained access control mechanism. First, it is not scalable. For instance, consider a table *Employees*, and suppose each employee can only access her own information. In order to enforce such policy using views, administrators have to create a view for each employee and grant each employee a permission to access the personal view. Clearly, this task is not efficient when there are a large number of employees. Also, this approach is cumbersome to support when policies frequently change. Another drawback of using views for access control is that view security becomes useless if users have direct access to base tables.

The VPD overcomes such limitations of views by dynamically modifying user queries. The basic idea of the VPD is as follows. A table (or a view) that needs protection is associated with a policy function which returns various predicates depending on the system context (e.g., current user, current time, etc.). Then when a query is issued against the table, the system dynamically modifies the query by adding the predicate returned by the policy function. Thus, the VPD provides an efficient mechanism to control data access at the row level based on both the system context and the data content.

It is possible to use VPD at the column level; i.e., associate policy functions with a column, not an entire table. In the case of column level VPD, one can also change the behavior of the VPD so that instead of filtering out inaccessible tuples, just the prohibited values are masked with nulls. The VPD also allows one to associate multiple policy functions with the same table. In such case, all policies are enforced with *AND* syntax.

5.3 Truman Model vs. Non-Truman Model

As previously mentioned, query-modification has been widely accepted as an effective technique for implementing access control in relational database systems. With this technique, when a subject issues a query, a modified query is executed against the database. As the modification of queries is transparent to users, this approach is equivalent to providing each user with a view of the complete database restricted by access control considerations and executing her queries only on this view. For this reason, this approach is called the "Truman model" for access control in relational database systems [23].

Although the Truman model can effectively support fine-grained access control, it has a major drawback; there may be significant inconsistencies between what the user expects to see and what the system returns [23]. For instance, consider a relation *Grades* which contains the grades of multiple students, and suppose each student is allowed to access only the grade of herself. Then when a student issues a query `SELECT * FROM Grades`, the Truman model returns only the grade of the student. However, suppose that a student wants to know the average grade and issues a query `SELECT AVG(grade) FROM Grades`. The Truman model effectively filters out the tuples inaccessible to the student. However, the system ends up returning the grade of the student, which is obviously incorrect. However, as such filtering is transparent to the student, the student may falsely believe that her grade is the same as the average grade.

Due to the inadequacy of the Truman model described above, Rizvi et al. [23] argue that query modification is inherently inapplicable as a mechanism for access control in database systems. As an alternative, they propose that every user be associated with a set of authorized views. Then when a user issues a query, the system assesses whether the query can be answered based on the authorized views associated with the user. If the answer is positive, then the query is processed without any modification. If the answer is negative, the query is rejected with a proper notification given to the user. For instance, if the user issues a query that asks for the average grade of all students, and the user's authorized views do not include every student's grade, then the system does not process the query and informs the user that the query cannot be answered.

5.4 Challenges in Fine-Grained Access Control

In general, a query processing algorithm A that enforces fine-grained access control policy takes as input a database D, a policy P, and a query Q, and outputs a result $R = A(D, P, Q)$. We argue that a "correct" query processing algorithm A should have the following three properties.

Soundness. $A(D, P, Q)$ should be "consistent" with $S(D, Q)$, where S is the standard relational query answering procedure. What do we mean by consistent will be formally defined later.

The intuition is as follows. When the policy P allows complete access to D, then $A(D, P, Q)$ should be the same as $S(D, Q)$. When P restricts access to D, $A(D, P, Q)$ may return less information than $S(D, Q)$ does, but it shouldn't return wrong information.

Security. $A(D, P, Q)$ should be using only information allowed by P; in other words, $A(D, P, Q)$ should not depend on any information not allowed by P. To formalize this, we require that for any D', P' such that if the portion of D allowed by P is the same as the portion of D' allowed by P', $A(D, P, Q) = A(D', P', Q)$. Precise definition of "the portion of D allowed by P" will depend on how P is specified, and will be given later.

This security property is inspired by the notion of "indistinguishability" security requirement for encryption schemes [**?**]. It says that if the algorithm A is used for query processing, then no matter what queries one issue, one cannot tell whether the database is in state D, P or in D', P'.

Maximality. While satisfying the above two conditions, A should return as much information as possible. To appreciate the importance of this property, observe that a query processing algorithm that always returns no information would satisfy the sound and secure property; however this is clearly .

We argue that our list of the three properties is intuitive and natural. In particular, it is declarative in that it does not define any procedure for answering queries. We illustrate the importance of having a definition of "correct query processing with fine-grained access control policy" by showing that two approaches previously discussed violate the above properties. For illustration, we consider the following example.

Example 3. We have a relation *Employee*, with four attributes: *id*, *name*, *age*, and *salary*, where *id* is the primary key.

id	**Name**	**Age**	**Income**
1	Alice	22	30000
2	Bob	45	65000
3	Carl	34	40000
4	Diane	28	55000

A policy for a user Alice consists of the following views:

– V_1: Alice can see all information about herself.
– V_2: Alice can see *id*, *name*, and *income* information for anyone whose *income* is less than \$60000.
– V_3: Alice can see *id*, *name*, and *age* for anyone whose *age* is less than 30.

Suppose Alice issues a query:

```
SELECT name, age, income FROM Employee
```
(Q_1)

Using the algorithm in [26], only the tuple about Alice is returned, because neither V_2 nor V_3 includes all three attributes in the query. However, if Alice issues two queries:

```
SELECT id, name, income FROM Employee
```
(Q_2)
```
SELECT id, name, age FROM Employee
```
(Q_3)

and does a natural join, then Alice would also be able to see Diane's information. In other words, Diane's information is indeed allowed by the views; however, the query rewriting algorithm in [26] does not use this information. Thus, the maximal property is violated.

In [17], an access control policy specifies which cells are allowed and which cells are not, and the approach for enforcing such a policy is to effectively replace each cell that is not allowed with the special value null. The soundness property is however violated using the standard SQL approach for handling null values; the standard SQL evaluates any operation involving a null as its operand to false. For instance, suppose that Alice issues a query:

```
SELECT name FROM Employee WHERE age < 25
```
(Q_4).

Observe that Q_4 is equivalent to the following query

```
(SELECT name FROM Employee) EXCEPT
   (SELECT name FROM Employee WHERE age ≥ 25)
```
(Q_5)

However, using the algorithm in [17], Q_3 returns {Alice} while Q_4 returns {Alice, Bob, Carl}.

Although our correctness criteria seems trivial, to the best of our knowledge, no fine-grained access control algorithm exists that is correct with respect to our definition. Also, devising a "correct" algorithm is not trivial. It seems that this problem of fine-grained access control still remains open.

6 Concluding Remarks

In this paper we have discussed some important requirements towards the development of privacy-preserving DBMS and we have identified initial approaches to address some of these requirements. In particular, we have presented two approaches dealing with purpose meta-data and their use in access control. These approaches represent initial solutions, that need to be extended in various directions. Relevant extensions are represented by efficient storage techniques and the introduction of obligations. We have also discussed current approaches to fine-grained access control and we have outlined the major drawbacks of these approaches. We have also identified three important properties that fine-grained access control models should satisfy. To date, however, no such model exist and its development is an important challenge.

Acknowledgement

The work reported in this paper has been partially supported by NSF under the project "Collaborative Research: A Comprehensive Policy-Driven Framework for Online Privacy Protection: Integrating IT, Human, Legal and Economic Perspectives".

References

1. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *The 28th International Conference on Very Large Databases (VLDB)*, 2002.
2. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. An XPath-based preference language for P3P. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, pages 629–639. ACM Press, May 2003.
3. A. I. Anton, E. Bertino, N. Li, and T. Yu. A roadmap for comprehensive online privacy policy. Technical Report TR 2004-47, Purdue University, 2004.
4. E. Bertino, E. Ferari, and A. Squicciarini. Trust negotation: Concepts, systems and languages. *IEEE Computing in Science and Engineering*, 6(4):27–34, Jul 2004.
5. J. Byun, E. Bertino, and N. Li. Purpose based access control for privacy protection in relational database systems. Technical Report 2004-52, Purdue University, 2004.
6. J. Byun, E. Bertino, and N. Li. Purpose based access control of complex data for privacy protection. In *Symposium on Access Control Model And Technologies (SACMAT)*, 2005. To appear.
7. C. Clifton. Using sample size to limit exposure to data mining. *Journal of Computer Security*, 8(4):281–308, 2000.
8. C. Clifton and J. Vaidya. Privacy-preserving data mining: Why, how, and when. *IEEE Security and Privacy*, 2(6):19–27, Nov 2004.
9. L. Cranor. P3P user agent guidlines, May 2003. P3P User Agent Task Force Report 23.
10. L. F. Cranor. Personal communication.
11. L. F. Cranor and J. R. Reidenberg. Can user agents acurately represent privacy notices?, Aug. 2002. Discussion draft 1.0.
12. M. M. et al. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, Apr. 2002. W3C Recommendation.
13. G. Hogben. A technical analysis of problems with P3P v1.0 and possible solutions, Nov. 2002. Position paper for W3C Workshop on the Future of P3P. Available at http://www.w3.org/2002/p3p-ws/pp/jrc.html.
14. G. Hogben. Suggestions for long term changes to P3P, June 2003. Position paper for W3C Workshop on the Long Term Future of P3P. Available at http://www.w3.org/2003/p3p-ws/pp/jrc.pdf.
15. G. Hogben, T. Jackson, and M. Wilikens. A fully compliant research implementation of the P3P standard for privacy protection: Experiences and recommendations. In *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, volume 2502 of *LNCS*, pages 104–125. Springer, Oct. 2002.
16. M. Langheinrich. A P3P Preference Exchange Language 1.0 (APPEL1.0). W3C Working Draft, Apr. 2002.
17. K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases, Aug. 2004. In 30th International Conference on Very Large Data Bases (VLDB), Toronto, Canada.

18. N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.
19. N. Li, T. Yu, and A. I. Antón. A semantics-based approach to privacy languages. Technical Report TR 2003-28, CERIAS, Nov. 2003.
20. N. McDonald, M. Stonbraker, and E. Wong. Preliminary specification of ingres. Technical Report 435-436, University of California, Berkeley, May 1974.
21. A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *The Fifth International Conference on Data Engineering (ICDE)*, pages 339–347, Feb. 1989.
22. Oracle Coperation. *Oracle Database: Security Guide*, December 2003. Available at www.oracle.com.
23. S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 551–562, Paris, France, 2004. ACM Press.
24. M. Schunter, E. V. Herreweghen, and M. Waidner. Expressive privacy promises — how to improve the platform for privacy preferences (P3P). Position paper for W3C Workshop on the Future of P3P. Available at http://www.w3.org/2002/p3p-ws/pp/ibm-zuerich.pdf.
25. D. M. Schutzer. Citigroup P3P position paper. Position paper for W3C Workshop on the Future of P3P. Available at http://www.w3.org/2002/p3p-ws/pp/ibm-zuerich.pdf.
26. M. Stonebraker and E. Wong. Access control in a relational database management system by query modification. In *Proceedings of the 1974 Annual Conference (ACM/CSC-ER)*, pages 180–186. ACM Press, 1974.
27. L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. In *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.
28. L. Sweeney. K-anonymity: A model for protecting privacy. In *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.
29. W3C. Platform for privacy preferences (P3P) project. http://www.w3.org/P3P/.
30. R. Wenning. Minutes of the P3P 2.0 workshop, July 2003. Available at http://www.w3.org/2003/p3p-ws/minutes.html.

Intrusion Detection: Introduction to Intrusion Detection and Security Information Management

Hervé Debar and Jouni Viinikka

France Télécom Division R&D,
42 rue des Coutures,
F-14066 Caen Cedex 4
{herve.debar, jouni.viinikka}@francetelecom.com

Abstract. This paper covers intrusion detection and security information management technologies. It presents a primer on intrusion detection, focusing on data sources and analysis techniques. Data sources presented therein are classified according to the capture mechanism and we include an evaluation of the accuracy of these data sources. Analysis techniques are classified into misuse detection, using the explicit body of knowledge about security attacks to generate alerts, and anomaly detection, where the safe or normal operation of the monitored information system is described and alerts generated for anything that does not belong to that model. It then describes security information management and alert correlation technologies that are in use today. We particularly describe statistical modeling of alert flows and explicit correlation between alert information and vulnerability assessment information.

1 Introduction

Information systems security has been a research area for a long time. Initial viruses and worms propagated slowly through the exchange of magnetic containers. With the development of TCP/IP, security problems have become more frequent and taken very different forms, and have lead to the development of new security techniques. Very early in the development of the Internet, vulnerabilities affecting operating systems have allowed attackers to move from system to system. Detecting attackers has been a necessity for military environments. Insufficient access control measures have led to the development of intrusion-detection systems (IDS).

These IDS have been developed to detect abnormal behaviour of information systems and networks, indicating a breach of the security policy. Two families of techniques have been developed, *misuse-detection* and *anomaly-detection*, to analyze a *data stream* representing the activity of the monitored information system. Misuse-based analysis detects known violations of the security policy, explicitly specified by the security officer. Anomaly-based analysis detects deviations from the normal behaviour of the monitored information system.

The objective of intrusion-detection systems today is to inform operators on the security health of information system. This mostly improves accountability but does not protect the information system from attacks. The development of dependable analysis techniques, particularly reduction of false alarms and identification of attack context, should in the future enable the migration to efficient intrusion protection systems, merging access control at the network layer with access control at the application layer. Distributing IDS components onto single workstations should create an efficient multi-layered approach to information security in the near future.

Beyond intrusion detection, security information management (SIM) platforms have emerged to manage alerts created by intrusion detection / prevention systems and other security tools, and provide a global view of the security state of the information system. These platforms are a must-have for large organization concerned with the security of their information systems, and are offering facilities for alert correlation, display and threat management.

In this paper, we will first cover intrusion detection by examining sensors, data sources and analysis techniques. We will then present security information management and alert correlation techniques.

2 Intrusion Detection Sensors

Figure 1 presents a schematic model of an intrusion detection / intrusion prevention system (IDS/IPS) according to the Intrusion Detection Working Group[1] (IDWG) of the Internet Engineering Task Force (IETF). This model contains all the important components of an intrusion detection system. The square boxes represent software or hardware components, while the ellipses represent human roles.

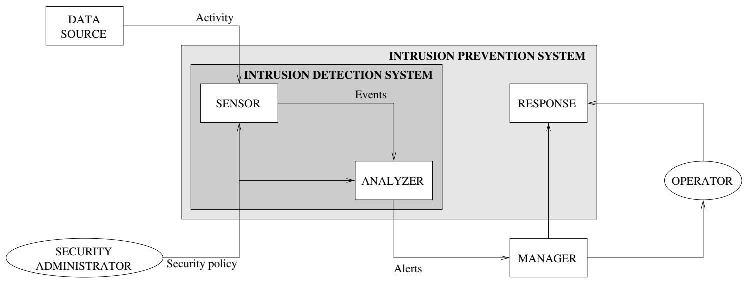


Fig. 1. Schematic model of an intrusion detection / intrusion prevention system according to the Intrusion Detection Working Group

An intrusion detection system observes the activity of the monitored information system through a data source. These data sources are captured and synthesized as events by the *SENSOR* component of the intrusion detection system. While nothing prevents an intrusion detection system to incorporate several

[1] `http://www.ietf.org/html.charters/idwg-charter.html`

sensors simultaneously and feed heterogeneous events to the *ANALYZER* component, current commercial intrusion detection systems acquire and analyze a single data source. Data sources are detailed in section 3.

Events are processed by an *ANALYZER*. This analyzer decides whether the event is malicious and should trigger the generation of an alert, or whether the event participates in the normal activity of the monitored information system. His decision depends both on the analysis technique and algorithms (see section 4) and on the configuration decided by the security administrator. The security administrator decides what conditions trigger alerts, what information is sent to the manager and what the appropriate response is. When alerts are triggered, the analyzer sends them to the manager. Alerts are described as Intrusion Detection Message Format[1] XML messages, transported over the IDXP[2] protocol.

When the *MANAGER* receives an alert, it displays its content to an operator. It can also decide to respond to the alert by applying countermeasures on one or several response components.

When an operator receives an alert, he is responsible for processing the alert according to the guidelines set forth by the security policy.

The main difference between IDS and IPS lies in the response capability. An IPS is often positioned inline, separating two networks like an application-level firewall. Now, for each packet or session that triggers an alert, the IPS can decide that the session needs to be terminated, the packet dropped or rejected, or possibly other measures. The response capability existed in IDS systems, in the sense that they were able to reconfigure other components such as firewalls to achieve a block, or to emit TCP RST packets to cut a connection. However, these responses depended on external devices or on luck, and they were often inefficient, mostly because the response was too slow and the attack had already propagated.

3 Data Sources

The first differentiator of intrusion detection technologies is the data source. Each data source requires specific processing to obtain event information, and allows the detection of different attacks.

3.1 Network Data Sources

IDS using network data are called Network Intrusion Detection Systems (NIDS). These NIDS observe the network, either on a hub, the SPAN port of a switch, or taps, collect the packets, and reconstruct an image of the activity of the users. NIDS are probably the most deployed systems today, and IPS are mostly NIDS. IPS have also brought in this area dedicated hardware devices capable of analyzing several gigabits of traffic inline, what previous software-based IDS were not able to do. There has been a large body of research related to optimizing NIDS such as the open source snort[3], but the current trend is clearly towards hardware-based or hardware-assisted solutions.

The network is a very attractive information source because it introduces little disturbance on the monitored information system[4]. SPAN ports on switches may imply bandwidth limitations on the observed traffic, but the stability problems that plagued early SPAN implementations are mostly gone. Another good solution is the tap, with a fail-open mode that guarantees service continuity even in the case of hardware failure. Finally, network operators are generally willing to look at the kind of traffic that crosses their wire, and the natural aggregation of traffic allows the monitoring of large IT infrastructures with a few well positioned boxes.

However, obtaining meaningful events from network observation is not very easy. One needs to decompose the traffic according to the various protocol layers. Also, experience has shown [5] that NIDS may suffer evasion, i.e. the attacker may be able to inject traffic in the network that will either render the attack invisible to the intrusion detection system or make it generate false alarms. Examples of these issues are:

Address masking. Network Address Translation (NAT) does not provide the IP address of the endpoint to the NIDS.

Encryption. Encrypted protocols defeat NIDS because most of their detection process requires access to headers (sometimes encrypted) and to payload (always encrypted). Popular protocols such as SSL and SSH render NIDS unusable.

Fragmentation. IP fragmentation is a rare phenomenon normally. It can be used to fragment the payload in such a way that the NIDS will not see the attack in one piece.

Reliability of the source address. Many attacks can be realized while using fictitious IP addresses (*IP spoofing*[5]) when no answer from the target is necessary. An attacker can also use *stepping stones*[6] to mask the origin of the attack. Finally, it can also use *reflectors*[7] to trick unwilling agents into carrying out the attack on its behalf.

Transience of address information. In many organizations, and ISPs in particular, IP addresses are handed out dynamically via DHCP. Identification of a particular customer through the IP address requires log analysis and can therefore be costly, or even impossible if logs have been rolled over.

The interested reader will find more detailed information in [5,8]. While these issues are not new and are fairly well known at the network layer, evasion techniques keep appearing at the application layer. Application protocols encode the information in the packet payload. Hence, exploitation of the payload requires that the IDS recognizes the possible encodings. The attacker can hide its actions through the use of specific encodings that are not understood by the NIDS.

For example, the HTTP protocol allows the replacement of any character by its ASCII hexadecimal code, prefixed with the % character [9]. This replacement is mandatory for special characters, obviously % but also space, and others. Ptacek and Newsham have shown that NIDS in 1998 did not understand this encoding and could not detect encoded attacks [5]. This particular problem has been solved for a long time, but others regularly appear.

Also, NIDS can include more intelligence than simple decoding, and incorporate recognition of protocol states, referred to as *protocol awareness.* A protocol aware NIDS recomposes the target protocol finite state machine and applies the detection algorithm to only the appropriate state. A NIDS that does not understand protocol states (often referred to as *network grep*) applies its detection algorithm to the payload of the packet regardless of the protocol state or history, and is much more likely to create false positives. Of course, keeping protocol states is costly, and these NIDS include safeguard mechanisms to avoid memory saturation, falling back to stateless detection if needed. '

3.2 System Data Sources

The system audit trails cover all the data sources that are made available by the operating system. The intrusion detection systems using host data sources are named Host Intrusion Detection Systems (HIDS). Syslog on UNIX systems and the NT event log under Microsoft windows operating systems provide to applications a service for identifying, time-stamping and storing information. Using this kind of facility is very easy for application developers, and they are used by several HIDS as a tool to collect, correlate and present system-related information.

Several operating systems also offer a so-called "C2 audit", to conform to the US government requirement for computer purchases. Such an audit aims at providing a trace of all privileged operations realized on a given computer, usually through the recording of system calls. It offers a strong user identification capability and an extremely fine-grained action description. Unfortunately, this C2 audit system is rarely properly documented and has a strong performance impact, so it has been abandoned by most HIDS.

Recent HIDS have developed specific interception software, similar to antivirus technology. This interception software allows the HIDS to recover only the information that can be analyzed, at lesser cost. Of course, these interception mechanisms are operating-system dependant, which results in a smaller number of offerings in the product space.

The main advantage of host audit data is the precision of the information. On this basis, an HIDS is able to reduce the number of false positives, while providing detailed information about the circumstances of the attack. In particular, actors (both target and perpetrator) are correctly and precisely identified. As such, the counter-measures can be appropriately tailored to the situation. Contextual information, related to the success of the attacker's activity, allows the operator to evaluate the risk and determine the appropriate level of counter-measures.

The main drawback of system audit data is that the HIDS has to reside on the same host, or a large volume of data has to be transported for remote analysis. Performance is such degraded through consumption of either bandwidth or processing power for security. Moreover, the behaviour of HIDS under stress heavily relies on the capabilities of the underlying operating system, and there is a real risk that denial-of-service attacks will either incapacitate the HIDS (if the original application has priority) or be facilitated (if the HIDS has priority).

Moreover, an application needs to be installed on the host. This has a strong impact on server deployment when servers have to be qualified before being placed in a production environment. Also, the signature updates may be problematic if software updates are also included in the signature updates.

3.3 Application Data Sources

Application logs cover all the traces maintained by the applications. The intrusion detection systems using application logs are considered as HIDS, even though an application log could provide information about a distributed environment, spanning multiple machines.

A typical example of application logs is the HTTP server log files storing requests presented to the server (usually in the `access.log` file) and error messages (usually stored in the `error.log` file. Each line stores the request presented to the server, and statistics about the response. The format of these log lines is reasonably easy to parse, making the data source an attractive proposition for developers.

Retrieval of the information generally consists of watching the file and parsing additional information into the data structures of the HIDS. Since these logs may ignore local information, constant information such as host names may be added on the fly to obtain an autonomous message.

Application logs are often more precise and dense than both system audits and network traffic, because they contain information that is atomic from the point of view of the application, while multiple packets or several thousand system calls may be necessary to realize the function.

Also, they provide more accurate information with the inclusion of return codes and error messages. These return and error messages are extremely important for the intrusion detection system, because they provide effective diagnostic of the issue and its impact on the monitored information system.

As already mentioned with network traffic, applications may use specific encodings. Depending on the log, decoding may also be required to normalize the information.

The biggest issue with application logs is that they are often targeting debugging and abnormal termination cases. As such, these files may not contain enough data for the HIDS. In certain cases, it is necessary to collect the entire transaction log, because even error-free transactions may contain attack-related activity that needs to be analyzed.

4 Analysis Techniques

Misuse detection takes advantage of the body of knowledge related to security vulnerabilities and penetration of information systems and networks. The IDS contains information about these vulnerabilities and looks for attempts to take advantage of them. When such an attempt is detected, an alert is sent to the management console. In other words, any action that is not explicitly identified as an inappropriate usage of the information system is considered acceptable.

Note that misuse detection does cover more than known attacks and vulnerabilities. If a security policy explicitly bans certain activities, these activities can be linked to alerts in an IDS. The best example would be banning IRC activity from a network. Any connection using the IRC port would trigger an alert. SNMP is also banned from certain environments and its inappropriate usage by network management tools can easily be detected. Also, recurrent attack mechanisms have been analyzed to abstract generic attack methods, covering not a single vulnerability but a class of them. These abstract models allow detection of broad attack patterns covering even some unknown vulnerabilities, or at least ensure that the detection mechanism does not rely on specificities of some attack tools.

Signature Description. In a misuse-based approach, one needs to define the trigger that, when found in the event stream, will generate an alert. This trigger is usually referred to as an attack *signature*, although the terms *scenario* and *rule* have been used to describe these triggers as well. The term *signature* will be used throughout this paper.

Initially, trigger description in IDES [10] took the form of facts entered into an expert system. User actions abstracted from the event stream were also represented as facts, while the detection process was described as production rules. This procedure was extremely costly, because of the processing needed to abstract several low level audit events into a single user action. Snapp and Smaha found that instead of abstracting system or network events to the expert system, it was easier to express vulnerabilities as sequences of events found in the event stream, named signatures [11].

A signature is the expression of some sequence of events characterizing the exploitation of a vulnerability. The detection process is thus simplified, and the cost is transferred to the definition and test of the signature for all the possible event stream formats that the IDS intends to support. This is sometimes a costly trade-off, if the event stream does not contain all the data that is being looked for, or if multiple encodings have to be taken into account.

In practice, a signature is expressed by a sequence of bytes being matched in the event stream [12], or in more complex cases by regular expressions [13]. These expressions are easier to write than the initial sequences of bytes proposed by [11], but it is still somewhat difficult to implement. Difficult because even for the same event stream format and the same exploited vulnerability, attacks can show under very different forms: attackers can mask their attempt under specific encodings or change sequencing by introducing irrelevant events in the data stream. Applying signatures to the data stream requires the sensor to remove protocol-specific encodings or operating-system related dependencies. This phase can be complex and costly performance-wise.

Misuse Detection and False Positives. The misuse detection approach should be able to generate very few false positives, if any. This however postulates that the attack is effectively detectable from the data stream, and that at least one signature properly characterizes the exploit.

False positives in misuse detection mainly come from an erroneous characterization of the vulnerability. This erroneous characterization often occurs when the IDS attempts to detect the execution of an application without differentiating between normal usage and the actual malicious attempt. For example, detection of CGI attacks is often based on the detection of the script name in the HTTP request, and identically-named scripts induce false positives.

Moreover, it is often difficult to differentiate the interactions between an attacker and a vulnerable information system from the interactions between normal users and the same information system. It is thus important to analyze alerts with the knowledge of the configuration of the monitored system, to ensure proper evaluation of the severity of these alerts.

Misuse Detection and False Negatives. Clearly, false negatives in misuse detection occur on new attacks, when there is no signature associated to the vulnerability. Collecting vulnerability information of sufficient quality to write adequate signatures is a time-consuming task, and validation of this information is often limited, due to the sheer number of attack combinations possible. Most often than not, IDS vendors obtain sample event stream information containing the attack and ensure that their tools can detect the occurrence of the sample data in the event stream. This is a long and tedious task.

Let's take a few numbers to illustrate this fact. An IDS today contains between 500 and 2000 signatures. Public vulnerability databases contain anywhere between 6000 and 20000 different vulnerability reports. Hence, there is roughly a one to 10 factor between what an intrusion-detection system knows about vulnerabilities, and what is publicly available. This ratio seems to be fairly stable; one counts between 100 and 150 new vulnerability announcements per month, associated with 10 to 20 new signatures announced by IDS vendors over the same period.

This difference is the product of two factors:

– Not all vulnerabilities are of interest to IDS users, because they affect only rare, specific environments or tools, or they do not provide the attacker with access to the vulnerable system, only limited denial of service. In addition, some of these vulnerabilities are old and affect very old software revisions that are not available anymore.
– A vulnerability may only leave some tracks in specific event streams. If the IDS does not recover this particular event stream, the vulnerability exploit cannot be detected.

Finally, there is the possibility of generic signatures that trigger on multiple vulnerabilities. This happens because attack code is reused from exploit script to exploit script, or because variations resulting in multiple vulnerabilities affect the same operating system or application and result in a single signature. Vendors are focusing on these generic signatures, hoping to cover not only vulnerabilities but also attack principles. Many products include generic buffer overflow detection, hoping to catch new exploits if they fit the attack technique.

Note that misuse-detection techniques are usually less-well suited for the detection of internal malicious activity. Registered users have access to the information system, and are likely to possess enough privileges on this information system to carry out most malicious activities without resorting to the exploitation of known vulnerabilities.

Misuse Detection and Counter-Measures. Misuse detection allows a contextual analysis of the attack and its effects on the monitored information system. This facilitates the understanding of the problem and the decision-making process for corrective or preventive action. Current research in alert correlation includes correlation between vulnerability assessment tools and intrusion-detection tools. Automated lookup of alert references in vulnerability reports will provide the operator with a mean to rank alert severity not only with respect to the attacker's potential gain, but also with the target's potential risk.

When target machine and target service are identified, it is reasonably easy to detect if the attack has some probability of succeeding, and if its effects are incompatible with the site security policy. The operator is able to evaluate the trade-off between the reliability and business objectives of the service, and the security policy objectives. *This is fundamental in counter-measures*: it could be legitimate to let the information system provide services even if compromised.

Evolution of Misuse Detection. Misuse detection prototypes have been initially implemented using first-order logic and expert systems. Current commercial products follow the so-called "signature-based" approach. There are also Petri-nets-related implementations and state transition analysis implementations.

Signature-based intrusion-detection systems usually rely on string or regular expression matching to detect specific pieces of information occurring in the data source. The matching mechanism is constrained further by specifying additional characteristics of the event stream, such as specific communication ports or protocol states. Each of these characteristics describes a particular facet of the vulnerability.

The expression of the signature depends of the level of detail available in the data source during exploitation of the vulnerability. For example, if a web server stops functioning during an attack before log entries can be written to disk, an intrusion-detection system based on log file analysis will not be able to detect the attack. Product vendors today tend to provide extremely wide signatures that will trigger on anything from normal usage to simple scanning, encouraging the notion that intrusion-detection systems cry wolf without cause.

4.1 Anomaly Detection

The general objective of anomaly detection is to define the correct behaviour of the monitored information system. An alert is generated when an event cannot be explained by the model of correct behaviour. This method assumes that an intrusion will induce a deviation from the normal usage of the information system.

Description of the Correct Behaviour Model. The model of correct behaviour can be constructed either from past samples of observed behaviour, of from explicit policy declarations. When an event occurs, the intrusion detection system compares the current activity with the model. An alert corresponds to the deviation of one or several measures between the current activity and the model.

As such, *anything that does not correspond to an explicitly-defined acceptable activity is considered anomalous.* Of course, the efficiency of such a system strongly depends on the capability of the model to represent the activity of the information system. For example, only using measures of CPU activity to model the normal behaviour of an information system would not allow straightforward detection of denial-of-service attacks filling disks or memory. It also assumes that the measures discriminate normal activity and malicious activity, as postulated by Denning [14,15]. Unfortunately, this postulate has not been validated theoretically. Experimental systems show that it is possible to detect some malicious activity by anomaly-based techniques, but do not qualify the coverage of this detection process.

The first models were based on learning techniques. A set of variables is defined that represents the interesting factors of the information system. Acceptable ranges for these variables are defined through observation of past data. A range here can be an association of average and standard deviation, or more complex statistical measures [16]. The model is trained during an observation period, and should converge towards stable values at the end of the observation period.

This area is still a research subject. New models and detection methods are regularly proposed, that improve constantly on existing technologies.

Advantages of Anomaly Detection Anomaly detection has, at least in principle, several advantages over misuse detection. First of all, it should be able to detect usage of unknown vulnerabilities. This is particularly important, as it does not rely on explicit security knowledge.

It also does not rely (or only in a limited way) on operating system specific knowledge, or application-specific knowledge. This is a great advantage when monitoring heterogeneous systems. After measures have been collected for the model, the intrusion detection system performs the modelling and detection process autonomously.

Finally, it can also detect abuse of privileges and insider attacks. Insiders usually have access to the monitored information system and do not need to use well known vulnerabilities to compromise the system and get access to the information they need. Misuse detection seldom detects insider attacks, whereas anomaly detection could show deviations from normal usage patterns.

Anomaly Detection and False Positives. Anomaly detection techniques often have a high false positive rate. This phenomenon arises from the fact that deviations from the model are often observed for any incident occurring on the monitored information system. Deviations also occur with configuration changes. Hence, the workload for processing alerts is large, and the operators have a frequent feeling that alerts are irrelevant to security.

This feeling is aggravated by the lack of explanation coming with the alerts. The root cause of the phenomenon is unknown, and the information provided seldom helps in resolving the issue.

Anomaly Detection and False Negatives. False negatives in anomaly detection have two main causes, corruption of the behaviour model and absence of measurement.

Corruption of the behaviour model occurs when the model learns an intrusive behaviour and incorporates it in its coverage. The intrusion detection system becomes incapable of detecting occurrences of the attack that has been accepted as part of the normal behaviour of the information system. Learning intrusive behaviour as normal occurs in particular in intrusion-detection systems where the model is constructed using past samples. Such systems need to be retrained periodically, and unfiltered training data could include malicious behaviour. Current research is therefore going away from learning technologies, and developing specification-based techniques to construct the model of normal behaviour.

Also, attacks sometimes do not impact the measures used by the model of normal behaviour. Let's take the very simple example of an intrusion-detection system that would monitor CPU usage and not disk usage. An attack that would fill the disk would not be detected by such a system. Of course, intrusion-detection systems make use of much more complex measures, including dynamic ones. As such, the exact coverage of the monitoring is difficult to establish.

Anomaly Detection and Counter-Measures. Alerts coming from an anomaly-based intrusion-detection system are often difficult to analyze. Counter-measures are difficult to deploy because neither target nor attack source are clearly identified, as well as the attack principle. Without an explicitly identified attack principle, counter-measures become extremely hazardous.

A new approach based on honeypot-like technology has recently been developed to improve identification of attack sources. When suspicious requests are identified, the response provided by the intrusion-detection system contains uniquely-identifiable information. When these specific tags come back, the intrusion-detection system can clearly identify the anomaly and its source.

5 Security Information Management

Once alerts are generated, they need to be handled by operators. Due to the volume and diversity of alert sources, security information management (SIM) platforms have emerged in recent years as the solution for concentrating heterogeneous logs and providing the security officer with a homogenous view of the security state of its information system.

The requirement for a central event and alert processing platform comes from the fact that many devices only provide a partial view of the security state of the information system, coherent with their role. Offering the desired global view can only be done by concentrating and consolidating as many information sources as possible. Note, however, that this does not mean that there will be only one SIM platform per organization, as SIM platforms should be able to communicate with one another.

5.1 SIM Functions

A SIM platform should cover the following four functions today.

Event Acquisition. Event acquisition deals with gathering and transporting events to a central point for further processing. This function covers the reliability of event transport, allowing both push and pull collection models, over a variety of protocols, to ensure that firewalls and other access control devices are properly traversed. This function has to deal with fairly heavy data flows, and it should be able to send hundreds to thousands of events per second to the central platform.

Another task carried out during event acquisition is related to filtering, aggregation and normalization. Given the enormous volume of event information that needs to be inserted in the database, the acquisition process must be able to select which events get inserted into the database. Also, for regular event streams, it is sometimes preferable to aggregate several identical events as one, adding the count of such aggregated events to the one tat is finally inserted in the database.

Finally, the normalization part of the acquisition process deals with ensuring a uniform representation of events in the database. There are differences in the naming conventions adopted by security tools such as intrusion detection systems or anti-virus systems. Two products tend to name the same attack with different signatures. The normalization process aims at ensuring that two events representing the same attack get the same name. Also, this process includes the capability to add reference information to the signatures, to ensure that internal references and processes are properly taken into account.

Contextual Information Management. Alert information usually includes some identification of the victim or source of the attack, identifying users and machines affected. This identification is often partial, including only network addresses or host names. However, most organisations maintain inventory information or vulnerability information assessment. This information should be attached to the host or user independentely of the host or user representation provided in the alert.

Therefore, the role of the contextual information management function is to ensure that all the contextual data is properly attached to hosts and users, and managing changes in this data so that it is kept up to date and accurate. This can be a tricky task in dynamic environments, for example with DHCP [17] or when remote users connect via VPN connections.

Alert Correlation. Alert correlation has as main objective to decide which alerts should be presented first to the security officer. It is in essence a triage and priority management system, which must ensure that the most critical alerts will be seen first. This triage system is supported by the association of a priority or security level with each event. Priority levels and schemes vary, but this is the role of the acquisition process to ensure that they are normalized to the IDMEF [1] set of values. To ensure that this triage process is successful, correlation must:

– fuse alerts that represent identical threat information together so that this threat is handled only once. This process is made difficult by the fact that clocks are often not exactly synchronized, and that some hypotheses must be made as to whether the fused events have the same root cause, e.g. have been raised by the observation of the same packet.
– relate alerts that participate in the same threat. Real attacks translate in multiple attacker actions, translating into multiple observations and multiple alerts being generated by the various intrusion detection and monitoring systems.
– aggregate high-volume alerts that cannot be interpreted individually, to ensure that patterns of aggregate alerts conform to the usual behaviour of the information system.
– incorporate contextual information into the evaluation of the severity of the event, to ensure that it has the proper awareness level. The most common representation of this process is to compare alert information and vulnerability assessment information, to inform the security officer of attacks associated with a security risk.

Alert correlation is an important research topic, particularly related to the processing of large volumes of alerts and to the intelligence of the correlation process.

Reporting and Exchanging. Finally, one needs to realize that a SIM platform does not live in isolation, but must offer several interfaces for accessing and pushing information. Typically, the following interfaces need to be provided :

– Operator real-time interface. This interface provides real-time alert information to the operator, typically through a scrolling window. This is the most common interface available in SIM consoles today, but not the most useful one, as operators need to be constantly on watch.
– Forensics analysis console. This interface provides navigation capabilities over the database of alerts, so that the security analyst can understand the incident, gather all related alerts, and propose solutions for better detection and resolution of the threat in later instances.
– Real-time incident reporting. In many cases, the threat requires countermeasures that have an impact on the normal function and the configuration of the monitored information system. However, if the configuration of the information system is not handled by the SIM console, it needs to send threat information to the system management console for proper handling.

The reporting and exchanging modules can also be used to create the peer relationships between SIM consoles or hierarchical relationships according to the needs of the organization.

5.2 Alert Representation

To support these functions, we have organized our data model as a set of concentric circles. Our data model is inspired from the Snort relational database schema, the IDMEF message format [1], and the M2D2 model [18]. We participated in deploying these tools and developing these models, so they naturally were used as a starting point for our development. However, we believe that event and contextual information are not equivalent and this is not obvious in the three models cited before. Hence, we choose to provide a different representation shown in figure 2.

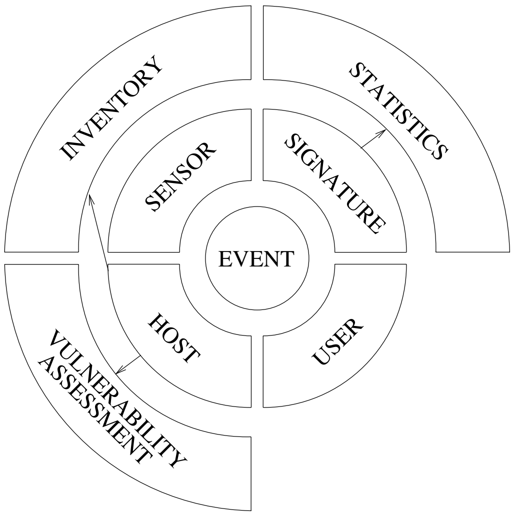


Fig. 2. Representation of alerts in the database

Event Information. The inner circle represents core event information, sensor, signature and timestamp. Note that the two first bits (sensor and signature) are in fact quite complex, comprising several tables and attributes in our database schema. These more complex bits are stored in the second circle, and each event links to the second circle for sensor and signature reference. This mechanism naturally takes into account differences in volume, as there are only a few hundred different sensors and a few thousand signatures for several million events. It also naturally renders the fact that sensor and signature information evolves on a much longer timeframe than event information.

Contextual Information. The second circle represents contextual information. This core event information links to host and user information in the second circle. Signature information links to sensor configuration, to indicate whether a sensor is able to detect an attack or not, and under which condition. Contextual information is mostly generated by the knowledge management processes (see 5.3) and by statically entered configuration information. While this contextual information evolves slowly over time, there is a need to track changes, as they have an impact on the signification of the events. Signatures are tracked using revision numbers that reflect improvements in their design. As such, events attached to the same signature message but with a higher revision number are considered more reliable than earlier events. The same process is used to track the evolution of sensor properties, each property change being tagged with the appropriate timestamp.

Transient Information. The third circle represents transient information that is generated by correlation processes (see Section 6). For example, statistical processes need to store numerical values associated with the statistical model; we use this area for the EWMA control values that monitor signature activity [19]. This circle also links to the first two. For example, the signature trending tool associates signature information in the second circle and event flows stored in the first circle.

The arrows of Figure 2 represent examples of links between contextual information and event information. A network event links to host information with source and destination of the network connection. A system event links to local host information indicating on which host the event occurred.

5.3 Contextual Information

Contextual information is related to the description of hosts and users. The objective of the contextual information knowledge management module is to ensure that the information linking alerts on one hand and hosts and users on the other hand are kept synchronized.

Structure of Host Representation. Logs represent host by three different keys, a host *name*, a host *IP address* and a host *MAC address*. The name is either fully qualified or a simple machine name, depending on the information source. This type of information is often provided by host-based information sources, or by devices configured to do on-the-fly reverse DNS mapping. An IP address is often provided by network-based IDS sensors and other network equipments. Finally, MAC addresses are provided by low-level networking devices such as wireless access points and switches, when specific network or wireless attacks are detected. All three keys are frequently found in event logs.

Different information sources will describe the host using different keys. To ensure that the same device is recognized by different sources, these three keys are associated in the same structure. Each key is associated with a Boolean value indicating whether this key was used by an information source or was derived from a data enrichment process. Upon insertion of an event, the process will first retrieve the appropriate host key with the Boolean set to $TRUE$, checking whether the same device has already been accessed. It will then re-query the same host key with the Boolean set to $FALSE$, checking whether enrichment of the host has taken place for an already existing host. If this is the case, it means that the host was previously inserted using another key coming from another source. For example, an IP address could be used for the first insertion, then a reverse DNS resolution could provide the host name that could then be encountered in host events. If such a host entry is found, the Boolean associated with the key will be set to $TRUE$ to facilitate future insertions, and the host entry found will be used to insert the event. If both searches fail, it will create a new host entry.

The knowledge management process attempt among other things to complete the key information associated with a host in the database. If a host is identified by an IP address, then a reverse DNS lookup is attempted to obtain the host name. If a host is identified by a host name, a DNS lookup is also attempted. Both operations are costly and would result in undue delays upon insertion of a new host, hence the choice of off-loading the acquisition process and pushing such task to a background process.

While defining the keys vas fairly straightforward, it happens that there are a number of issues with the keys that we have designed.

Host Information Collection Point. The first remark is that we collect host information in a different location that event information. Most if not all of our sensors are passive devices, to limit the risk of attack against them. Therefore, they do not have the capability of adding host information to events. As a result, the gathered host information is from the vantage point of the application server and not the sensor. The advantage is that all events are tagged from the same viewpoint, thus normalizing the events. The disadvantage is that the application server needs to be able to access all host information, and that static local information (e.g. host names stored in `/etc/hosts`) will not be accessible. Even though one could fear that the visibility from the application server and from the sensor is different, we have not observed wrong host information as a result of this process.

Network Address Translation. Network address translation [20] is frequently used by private companies and internet service providers to mask the internal structure of their environment and to lessen address space requirements. As such, different machines may be seen as a single one by our application. When NAT is in place, the name resolved through DNS is the name of the NAT device and does not reflect the exact name of the target or source of the event. As a result, our application in this configuration identifies a domain but not the exact target or source.

This problem rarely occurs for hosts under our control. This means that we usually can still precisely identify hosts that are within our realm.

Dynamic Host Configuration. The dynamic host configuration protocol (DHCP [17]) allows the same machine to have multiple IP addresses over time.

Moreover, host name information is sometimes generic as well, reusing for example the two last bytes of the IP address. When this is the case, our application is not able to uniquely identify a machine.

DHCP is a frequent occurrence in the environments monitored by our application. Our tool attempts to determine whether the host is within a DHCP environment. When this is recognized, the host is identified by its host name, which is stored in our DNS servers during DHCP handshake and is uniquely generated in the corporation. When such a host is identified by an IP address, the knowledge management tool reconfigures the event-host associations a posteriori, using DNS queries. This is a domain-specific solution and may not be applicable to other environments. In particular, ISPs tend to use generic names, as mentioned earlier. Another solution would be to use the MAC address, but this key is very rarely available.

Mobility. Mobility is a frequent occurrence in our corporate environment. Laptops and the use of DHCP facilitate remote connections, as well as the generalization of VPN connectivity. However, this poses a problem when such a host is the subject of a security problem, particularly a viral or worm infection. We need to distinguish the case of laptops connected internally into a site that is not their home site, and laptops using VPN connections to access the information system.

In the case of locally-connected laptops, it is often the case that the resolution between host name and host IP address is wrong. Let's take the case of a virus infection. This virus infection is logged into the NT Event Log of the laptop, which in turns connects to a central server to deliver the infection alert. This infection alert is based on the host name, which is a unique key in our corporate network. As such, the event will be correctly assigned to the host. Unfortunately, geographic information is based on the IP address. If an IP address already exists for the host name, this information is not systematically refreshed as this is a costly process. Therefore, getting the current geographic coordinates of the infected laptop requires an additional DNS lookup to retrieve the current IP address and its associated geographical location. This process is quite time consuming and correct information may not be immediately available; if this process takes too much time the connection is terminated.

In the case of VPN-connected laptops (which is also used for wireless connections), the IP address of the laptop resolves to the IP address of the VPN concentrator. Therefore, it is impossible to retrieve the physical location of the infected machine and the connection is terminated.

5.4 Vulnerability Assessment

Collecting vulnerability assessment information is generally done through the usage of a vulnerability scanner, either remotely by querying the audited host, or locally by installing an inventory module. Local auditing us usually more accurate, but requires software installation on the tested host, which is not always feasible. Therefore, vulnerability assessment information is often connected through the network.

While vulnerability assessment reports are extremely useful for the security officer, they suffer from the following issues:

Server side only Remote vulnerability assessment report query active services. Therefore, they do not provide information about client side vulnerabilities or firewalled ports. Only local software installation can provide this information.

Audit cost and timeliness More than actual bandwidth consumption these days, the audit cost is the time it takes to test and evaluate large numbers of hosts. Therefore, audits may only be carried out at spaced intervals.

Audit risk and accuracy Certain tests can have undesirable side effects on the tested host, such as leaving it vulnerable to certain attacks, or bringing it down. The more accurate an audit report is, the riskier it generally is as well.

To take into account some of these issues, passive network observation has been introduced to collect information related to vulnerabilities on both clients and servers. By collecting product names and versions from the network, either with a dedicated tool such as ettercap or with a network intrusion detection system equipped with a specific rule set, the passive network observation sensor can obtain a fairly complete inventory of the information system. Using external vulnerability databases such as OSVDB, it is then possible to deduce vulnerability assessment information for the information system.

6 Alert Correlation

The need to automate alert processing and to reduce the amount of alerts displayed to the operator by the system is a widely recognized issue and the research community has proposed as one solution to correlate related alerts to facilitate the diagnostics by the operator [21].

Alert correlation has three principal objectives with regard to information displayed to the operator:

Volume reduction: Group or suppress alerts, according to common properties. E.g. several individual alerts from a scan should be grouped as one meta alert.

Content improvement: Add to the information carried by individual alert. E.g. the use of topology and vulnerability information of monitored system to verify or evaluate the severity of the attack.

Activity tracking: Follow multi-alert intrusions evolving as time passes. E.g. if attacker first scans a host, then gains remote-to-local access, and finally obtains root access, individual alerts from these steps should be grouped together.

We perform volume reduction eliminating redundant information by aggregating alerts that are not strictly symptoms of compromise and appear in high volumes. Only changes in the behaviour of the aggregate flow are reported to the user. Correlation techniques capable of detecting unknown, novel relationships in data are said to be *implicit* and techniques involving some sort of definition of searched relationships are called *explicit*. As the aggregation criteria are manually selected, this is an explicit correlation method. Overall, we aim to save operator resources by freeing the majority of time units that manually processing the background noise would require and thus to enable him to focus on more relevant alerts. Even though this manual processing is likely to be periodic skimming through the accumulated noise, if there are several sources with omnipresent activity, the total time used can be significant. Next we discuss why despite the large amounts of alerts background noise monitoring can be useful.

6.1 Statistical Correlation

According to our experience (see Sect. 6.1) a relatively large portion of alerts generated by a sensor can be considered as background noise of the operational system. However, the division to true and false positives is not always so black and white. The origins of problem can be coarsely divided to three. 1) Regardless of audit source, the audit data usually does not contain all required technical information, such as the topology and the vulnerability status for the monitored system for correct diagnosis. 2) The non-technical contextual factors, such as operator's task and the mission of the monitored system, have an effect on which types of alerts are of high priority and relevant. 3) Depending on the context of the event, it can be malicious or not, and part of this information can not be acquired by automated tools or inferred from the isolated events. For the first case, think of a Snort sensor that does not know if the attack destination is running a vulnerable version of certain OS or server and consequently can not diagnose whether it should issue an alert with very precise prerequisites for success. An example of the second is a comparison of on-line business and military base. At the former the operator is likely to assign high priority on the availability of the company web server, and he might easily discard port scans as minor nuisance. At the latter the operator may have only minor interest towards the availability of the base web server hosting some PR material, but reconnaissance done by scanning can be considered as activity warranting user notification. Instead of high priority attacks, the third case involves action considered only as potentially harmful activity, such as ICMP and SNMP messages that indicate information gathering or network problems, malicious as well as innocuous as part of normal operation of the network. Here the context of the event makes the difference, one event alone is not interesting, but having a thousand or ten events instead of the normal average of a hundred in a time interval can be an interesting change and this difference can not be encoded into signature used by pattern matching sensor.

This kind of activity is in the grey area, and the resulting alerts somewhere between false and true positive. Typically the operator can not afford to monitor it as such because of the sheer amount of events. The current work on correlation is largely focusing on how to pick out the attacks having an impact on monitored system and show all related events in one attack report to the operator. Depending on the approach, the rest of the alerts are assigned such a low priority that they do not reach the alert console [22], or they can be filtered out before entering the correlation process [23,24]. However, if the signature reacting to grey area events is turned on, the operator has some interest towards them. Therefore it is not always the best solution to only dismiss these less important alerts albeit their large number. Monitoring aggregated flows can provide information about the monitored system's state not available in individual alerts, and with a smaller load on operator. Our work focuses on providing this type of contextual information to the user.

EWMA control charts were originally developed for statistical process control (SPC) by Roberts [25], who used the term geometric moving averages instead of EWMA, and since then the chart and especially the exponentially weighted moving average have been used in various contexts, such as economic applications and intrusion detection [26,27,28]. Our needs differ from those of Roberts' quite much, and also to a smaller degree from those of the related work in intrusion detection. Below our variation of the technique is described, building largely on [28], and the rationale for changes and choices is provided.

The monitored measure is the alert intensity of a flow x, the number of alerts per time interval. One alert flow consists typically of alerts generated by one signature, but also other flows, such as alerts generated by a whole class of signatures, were used. Intensity x is used to form the EWMA statistic. This statistic is called the *trend* at time i. It is quite impossible to define a nominal average as the test baseline x_0, since these flows evolve significantly with time. Like Mahadik et al. [28], to accommodate the dynamic, non-stationary nature of the flows, the test baseline is allowed to adapt to changes in alert flow.

Learning Data. The tool was developed for an IDS consisting of Snort sensors logging alerts into a relational database. The sensors are deployed in a production network, one closer to Internet and two others in more protected zones. This adds to the difficulty of measuring and testing, since we do not know the true nature of traffic that was monitored. On the other hand, we expect the real world data to contain such background noise and operational issues that would not be easily incorporated to simulated traffic.

The data set available to us in this phase contained over 500 K alerts accumulated during 42 days. Of the 315 activated signatures, only five were responsible for 68 % of alerts as indicated in Table 1 and we chose them for further scrutiny. To give an idea of the alert flow behaviour, examples of alert generation intensities for four of these signatures are depicted in Fig. 3. The relatively benign nature of these alerts and their high volume was one of the original inspirations for this work. These alerts are good examples of the problem three and demonstrate the reason why we opt not just filter even the more deterministic components out. For example, the alert flow in Fig. 3(c) triggered by SNMP traffic over UDP had only few (source, destination) address pairs, and the constant component could be easily filtered out. However, this would deprive the operator being notified of behaviour such as the large peak and shift in constant

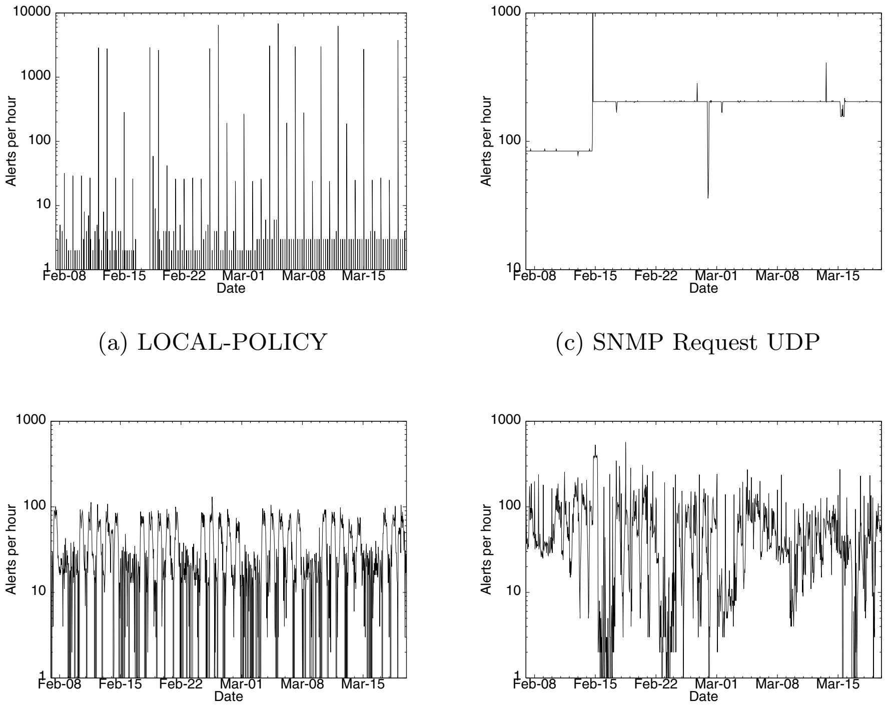
Table 1. Five most prolific signatures in the first data set

signature name	**number of alerts**	**proportion**
SNMP Request UDP	176 009	30 %
ICMP PING WhatsupGold Windows	72 427	13 %
ICMP Destination Unreachable (Communication Administratively Prohibited)	57 420	10 %
LOCAL-POLICY External connexion from HTTP server	51 674	9 %
ICMP PING Speedera	32 961	6 %
sum	**390 491**	**68 %**

component around February 15th as well or the notches in the end of February and during March 15th. Not necessarily intrusions, but at least artefacts worth further investigation. On the other hand, we do not want to distract the operator with the alerts created during the hours that represent the stable situation with constant intensity. For the others, Fig. 3(a) shows alerts from a user defined signature reacting to external connections from an HTTP server. The alerts occur in bursts as large as several thousands during one hour and the intensity profile resembles impulse train. As custom made, the operator has likely some interest in this activity. In Figs. 3(b) and 3(d) we have alerts triggered by two different ICMP Echo messages, former being remarkably more regular than latter. In the system in question, deactivation of ICMP related signatures was not seen as a solution by the operator as they are useful for troubleshooting problems. Consequently, we had several high volume alert flows for which the suppression was not the first option.

Effect of Flow Volume. Judging from the busy interval reduction, the method is useful for alert flows that had created more than 10 K alerts, the effectiveness increasing with the flow volume. The busy interval reduction for flows below 10 K alerts is already more modest, and below 1 K alerts the reduction is relatively negligible. Tables 2 and 3 depict respectively the reduction as percentage from non-zero intervals and alerts flagged anomalous, due to space constraints only for flows of over 10 K alerts. Reduction is shown with smoothing factors 0.80 and 0.92 for each of the three models, continuous, hourly, and weekday. In Table 2 also the total number of active intervals, and in Table 3 the total number of alerts are shown for each flow.

Table 4 summarizes alert reduction results with continuous model and smoothing factor 0.92. All 85 flows are grouped to four classes according to both their output volume (over 100, 1 K, 10 K or 100 K alerts) and the achieved reduction in busy intervals and alerts (below 5 %, 10 %, 50 % or 100 % of original), respectively. These results show also the poorer performance for flows below the 10 K limit. The busy intervals show more consistent relation between the volume and reduction. On the right hand side of Table 4 in the class over 100 K alerts, `ICMP Dest Unreachable (Comm Admin Proh)` stands out with reduction signifi-



(a) LOCAL-POLICY

(c) SNMP Request UDP

(b) Speedera

(d) Dest. Unreachable

Fig. 3. Hourly alert intensity for some of the most prolific signatures in learning data. Horizontal axis is the time, and the vertical shows the number of alerts per hour

cantly smaller than others in the same class. We found two explanations for this behaviour. First, there was one large alert impulse of approximately 17 K alerts flagged in the test data. This makes up roughly 10 % of flagged alerts. Second, the flow nature is more random compared to others, this is visible in Fig. 3(d) for learning data and applies also for the larger data set. This randomness causes more alert flagging, but still the reduction in busy intervals is comparable to other flows in this volume class.

Reasons for Poor Summarization. There seems to be two main reasons for poorer performance. 1) Many flows had few huge alert peaks that increase the alert flagging significantly. 2) The intensity profile has the form of impulse train that has negative impact both on reduction of alerts and busy intervals. As the first cause does not increase remarkably the number of reported anomalous intervals i.e. the number of times the user is disturbed, this is smaller problem. However, the second cause renders our approach rather impractical for monitoring such a flow, as the operator is notified on most intervals showing activity.

Table 2. The proportion of flagged intervals from intervals showing activity for the flow with different models and smoothing factors

		cont.		**daily**		**weekd.**	
flow	**int.**	.80	.92	.80	.92	.80	.92
Known DDOS Stacheldraht infection	563	1.6	1.8	8.9	8.5	2.0	2.5
SNMP request UDP	2311	4.3	2.9	5.8	4.6	4.2	3.0
ICMP PING WhatsupGold Windows	2069	5.1	3.3	5.8	2.6	5.1	3.2
DDOS Stacheldraht agent→handler (skillz)	512	1.2	1.6	12	16	1.8	2.1
ICMP Dst Unreachable (Comm Adm Proh)	2578	5.4	3.5	6.7	5.8	5.4	3.4
ICMP PING speedera	2456	3.3	1.7	4.2	2.9	3.3	0.9
WEB-IIS view source via translate header	2548	5.2	3.8	6.4	5.7	5.1	4.0
WEB-PHP content-disposition	2287	6.8	4.3	7.7	5.2	6.7	4.0
SQL Sapphire Worm (incoming)	1721	2.2	1.2	4.9	3.5	2.4	1.6
(spp_rpc_decode) Frag RPC Records	421	13	7.8	20	20	12	9.0
(spp_rpc_decode) Incompl RPC segment	276	21	13	27	27	22	13
BAD TRAFFIC bad frag bits	432	34	23	37	33	35	22
LOCAL-WEB-IIS Nimda.A attempt	537	24	16	30	25	24	16
LOCAL-WEB-IIS CodeRed II attempt	1229	6.3	4.6	14	14	6.9	5.3
DNS zone transfer	855	9.7	6.7	13	10	9.8	6.5
ICMP L3retriever Ping	107	29	26	71	70	28	23
WEB-MISC http directory traversal	708	12	9.3	15	13	12	9.5
(spp_stream4)STLTH ACT(SYN FIN scan)	29	65	58	82	79	62	62

Table 3. The percentage of flagged alerts with different models and smoothing factors

		cont.		**daily**		**weekd.**	
flow	**alerts**	.80	.92	.80	.92	.80	.92
Known DDOS Stacheldraht infection	308548	1.2	1.2	4.4	8.4	1.4	1.5
SNMP request UDP	303201	4.4	3.0	4.9	4.4	4.2	3.2
ICMP PING WhatsupGold Windows	297437	5.4	4.0	4.5	2.9	5.2	3.1
DDOS Stacheldraht agent→handler (skillz)	280685	0.8	1.0	7.3	7.0	1.2	1.2
ICMP Dst Unreachable (Comm Adm Proh)	183020	32	28	39	37	32	28
ICMP PING speedera	95850	5.5	3.1	2.5	2.3	5.3	1.4
WEB-IIS view source via translate header	58600	25	21	12	11	24	22
WEB-PHP content-disposition	48423	18	14	15	13	18	14
SQL Sapphire Worm (incoming)	38905	3.0	1.9	11	9.1	3.1	2.5
(spp_rpc_decode) Frag RPC Records	38804	63	62	94	93	63	62
(spp_rpc_decode) Incompl RPC segment	28715	64	62	93	93	64	62
BAD TRAFFIC bad frag bits	27203	51	42	57	54	53	42
LOCAL-WEB-IIS Nimda.A attempt	25038	65	61	69	64	64	62
LOCAL-WEB-IIS CodeRed II attempt	20418	11	7.5	17	22	11	7.1
DNS zone transfer	15575	32	35	55	55	32	36
ICMP L3retriever Ping	12908	11	12	90	90	11	12
WEB-MISC http directory traversal	10620	41	38	46	45	41	38
(spp_stream4)STLTH ACT(SYN FIN scan)	10182	96	90	93	93	96	96

Table 4. All 85 flows grouped by the number of alerts created and the percentage level below which busy intervals or alerts were flagged

busy interval reduction

alerts	5 %	10 %	50 %	100 %
> 100 K	5	0	0	0
> 10 K	5	3	4	1
> 1 K	0	4	19	7
> 100	0	1	12	24
sum	10	8	35	32

alert reduction

alerts	5 %	10 %	50 %	100 %
> 100 K	4	0	1	0
> 10 K	2	1	6	4
> 1 K	0	1	15	14
> 100	0	0	8	29
sum	6	2	30	47

The flow `(spp_stream4)` on the last row of Tables 2 and 3 is a typical example, as its alert profile consisted only from impulses. In such situation a large majority of active intervals are flagged as anomalous. A closer look on alert impulses revealed that they were usually generated in such a short time interval that increasing the sampling frequency would not help much. Instead, other means should be considered to process them.

Represented Alert Types. Amongst the most prolific signatures, we can identify three main types of activity, *hostile*, *information gathering* and alerts that can be seen to reflect the *dynamics* of networks.

Hostile activity is represented by DDoS tool traffic and worms with five signatures. The two DDoS signatures are actually the same, different names were used by the operator for alert management reasons. If busy interval reduction below 5 % with continuous model and $(1 - \lambda) = 0.92$ is used to define EWMA monitoring applicable for a flow, then we have three fourths in feasible range for the hostile activity.

In the system in question, possible information gathering is the most common culprit for numerous alerts. This category can be further divided to information gathering on applications (web related signatures) and network architecture (ICMP, SNMP and DNS traffic). In both categories, there are both suitable and unsuitable flows for this type monitoring.

The ICMP Destination Unreachable (Communication Administratively Prohibited) message is an example of the activity that describes the dynamics of the network. It reflects the network state in terms of connectivity, and the origins and causes of these events are generally out of operators control.

Signatures firing on *protocol anomalies* can be considered as an orthogonal classification, since they can present any of the three types above. (`(spp rpc decode)`, `(spp stream4)` and `BAD TRAFFIC`) were all poorly handled by the method. Another common factor is the smaller degree of presence in the data set in terms of non-zero intervals. As the `(spp_stream4)` means possible reconnaissance, and being present only on 29 intervals, it is less likely to be just background noise.

The nature of these alerts and their volumes in general support the claim that large proportion of generated alerts can be considered as noise. Even in the

Table 5. The omnipresence of signatures and their types. Presence measured in active intervals

signature	**type**	**< 5 %**	**active**	**present**
ICMP Dst Unreachable (Comm Adm Proh)	network	ok	2578	95 %
WEB-IIS view source via translate header	info_web	ok	2548	93 %
ICMP PING speedera	info_net	ok	2456	90 %
SNMP request UDP	info_net	ok	2311	85 %
WEB-PHP content-disposition	info_web	ok	2287	84 %
ICMP PING WhatsupGold Windows	info_net	ok	2069	76 %
SQL Sapphire Worm (incoming)	hostile	ok	1721	63 %
LOCAL-WEB-IIS CodeRed II attempt	hostile	ok	1229	45 %
DNS zone transfer	info_net	no	855	31 %
WEB-MISC http directory traversal	info_web	no	708	26 %
Known DDOS Stacheldraht infection	hostile	ok	563	20 %
LOCAL-WEB-IIS Nimda.A attempt	hostile	no	537	19 %
DDOS Stacheldraht agent-¿handler (skillz)	hostile	ok	512	18 %
BAD TRAFFIC bad frag bits	proto	no	432	15 %
(spp_rpc_decode) Frag RPC Records	proto	no	421	15 %
(spp_rpc_decode) Incompl RPC segment	proto	no	276	10 %
ICMP L3retriever Ping	info_net	no	107	3 %
(spp_stream4)STLTH ACT(SYN FIN scan)	proto	no	29	1 %

case of hostile activity the originating events warrant aggregation. This applies in our case, but the situation may vary with different operating environments.

Table 5 shows the signature flows ordered by their omnipresence giving the number of active intervals and the percentage this makes out of the whole testing interval. A rough division according to the 5 % watershed is made and type of signature according to above discussion is assigned. We can see that for all signatures showing activity on more than 45 % of the intervals the number of alerts issued to operator can be significantly reduced in this system.

It would seem that the omnipresence of alerts would be better criteria than the alert type for determining whether EWMA monitoring would be useful or not.

Impact of Time Slot Choice. According to these metrics the usefulness of daily and weekday models was limited to a few exceptions, generally the continuous model was performing as well as the others. We just happened to have one of the exceptions that really profited from hourly approach in our early experimentations, and made the erroneous hypothesis of their commonness. The metrics are however limited for this kind of comparisons. It is especially difficult to say if the hourly approach just marks more intervals as anomalous or is it actually capturing interesting artefacts differently. On many occasions the smaller reduction was at least partly due to abrupt intensity shifts. As several different statistics making up the hourly model signal an anomaly whereas the continuously updated statistic does this only once. The two DDoS flows had intensity profiles resembling a step function, which caused the hourly model to flag significantly more alerts than the continuous.

Another factor encumbering the comparisons is the difference in efficient lengths of model memories. As the time slot statistics of hourly and weekday models are updated with only the corresponding intensity measures the values averaged have longer span in real time. For example the hourly model's statistics are affected by 8 or 24 *days* old measurements.

Class Flows. Grouping signature classes together increased the flagging percentage. Table 6 shows obtained reductions with continuous model and $(1-\lambda) = 0.92$ for class aggregates with more than 1000 alerts. In fact, almost every class contains one or more voluminous signatures that were problematic statistically already by themselves, and this affects the behaviour of class aggregate. The increased flagging could also indicate that anomalies in signature based flows with smaller volume are detected to some degree. The levels of busy intervals are reduced relatively well and again generally the flagging increases as alert volume decreases. The aggregation by class might be used to gain even more abstraction and higher level summaries in alert saturated situations. However, there are likely to be better criteria for aggregation than the alert classes.

Table 6. The reduction in alerts and busy intervals when aggregating according to signature classes. Results for continuous model with $1 - \lambda = 0.92$

	raw		**anomalous**	
flow	**int.**	**alerts**	**int.**	**alerts**
misc-activity	2678	618273	1.9 %	8.9 %
class_none	1429	380568	4.8 %	18.3 %
attempted-recon	2635	360613	3.7 %	7.0 %
known-issue	563	308548	1.7 %	1.1 %
web-application-activity	2569	88554	3.3 %	16.3 %
bad-unknown	2559	65883	3.7 %	20.9 %
known-trojan	1511	46014	5.4 %	34.9 %
misc-attack	1727	39070	1.3 %	2.1 %
web-application-attack	1017	9587	9.1 %	40.5 %
attempted-user	272	3694	19.4 %	40.6 %
attempted-dos	361	2782	24.3 %	67.8 %
attempted-admin	444	1760	20.2 %	33.1 %

Flow Stability. To give an idea of the stability of flow profiles, Table 7 compares the alert and busy interval reduction obtained for four signatures used in the learning phase against the reduction in testing data. In general the flagging is slightly higher in the training data set. The most notable exception is signature `ICMP Destination Unreachable (Communication Adm Prohibited)`, where a significant number of alerts is marked anomalous in the test set. The large alert impulse in this flow, mentioned earlier, accounts for approximately

Table 7. A comparison of results obtained during learning and testing phases. $(1-\lambda) = 0.92$

	alerts		intervals	
flow	**learn.**	**test**	**learn.**	**test**
SNMP request UDP	2.7	3.5	2.2	3.5
ICMP PING WhatsupGold Windows	4.6	3.6	2.9	3.6
ICMP Dst Unreachable (Comm Adm Proh)	12	36	3.2	3.7
ICMP PING speedera	2.8	3.2	1.3	2.0

14 % units of this increase in test data. Even if those alerts were removed, the increase would be large. Still, the reduction in busy intervals is quite similar, suggesting higher peaks in the test set. The fifth signature enforcing a local policy, also viewed in the learning phase, did not exist anymore in the testing data set. This signature created alert impulses (see LOCAL-POLICY in Fig. 3(a)) and the alert reduction was marginal in learning data.

It seems like with the used parameters the reduction performance stays almost constant. This would suggest that after setting parameters meeting the operators needs, our approach is able to adapt to lesser changes in alert flow behaviour without further adjustment. At least during this test period, none of the originally nicely-enough-behaving flows changed to more problematic impulse-like nor vice versa. Also signatures having a constant alert flow or more random process type behaviour, both feasible for the method, kept to their original profile.

To wrap up the results, it seems possible to use this approach to summarize and monitor the levels of high volume background noise seen by an IDS. Up to 95 % of the one hour time slots showing activity from such an alert flow can be unburdened from the distraction. For the remaining intervals, instead of a barrage of alerts, only one alert would be outputted in the end of the interval. As both data sets came from the same system, the generality of these observations is rather limited, and more comprehensive testing would be required for further validation.

If the user is worried that aggregation at signature level loses too much data, it is possible to use additional criteria, such as source and destination addresses and/or ports to have more focused alert streams. The reduction in aggregation is likely to create more flagged intervals, and this is a tradeoff that the user needs to consider according to his needs and the operating environment. Determining if the summarization masked important events in the test set was not possible, as we do not possess records of actual detected intrusions and problems in the monitored system against which we could compare our results.

6.2 Correlation with Vulnerability Information

Correlation between alerts and vulnerability information aims at assessing the risk that the monitored information system incurs from the attacker's actions. The actions of the attacker are deemed potentially successful and high risk if the vulnerability exists on the information system.

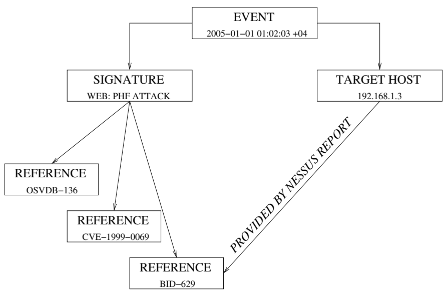


Fig. 4. Explicit correlation between event and vulnerability information, using vulnerability assessment reports.

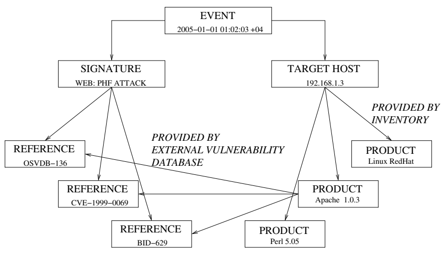


Fig. 5. Explicit correlation between event and vulnerability information, using inventory information.

The easiest way to realize this correlation mechanism is shown in Figure 4. An event contains information about the attack mechanism and the target host of the attack. The attack mechanism is associated with references, often to public external databases, that constitute an open dictionary for all intrusion detection and SIM vendors. The vulnerability assessment information provides the link between hosts and vulnerability references. When a loop can be found in the graph, the alert represents an attack for which the information system was vulnerable, hence a more serious risk.

The difficulty with this kind of correlation is that we may not have a vulnerability assessment report for the host. When this is the case, we may use inventory information, either created externally or through passive network observation as mentioned in Section 5.4. As shown in Figure 5, the association between hosts and vulnerability references goes through product information.

Note that this procedure also offers another advantage, which is the possibility to correlate alerts with non-existent vulnerabilities. Vulnerability assessment tools rarely indicate explicitly when a server is not vulnerable to a given attack. This lack of information can be attributed to the absence of the vulnerability, but there could be other factors that make the test fail or not complete, while the vulnerability would still exist. However, vulnerability databases often include non-vulnerable information related to the product versions. If the product versions are comparable, then it is also possible to lower the severity of an alert when the risk does not exist in the information system.

This correlation mechanism is largely in use in SIM consoles. We are currently studying the efficiency of this mechanism, to precisely evaluate what it can and cannot provide.

7 Conclusion

In this paper, we have presented intrusion detection and security information management as two important and active research domains for information systems security. While intrusion detection offers mature technologies for deployment, security information management remains an interesting research subject from which we can expect new advances.

Related to intrusion detection, we expect that research will focus on application-level attacks and on much more accurate sensors and detection algorithms than are currently available. The sheer number of uninteresting intrusion detection alerts generated by these tools will require continuous tuning and development, until systems become more secure.

Security information management will continue to foster research in alert correlation, leading to more complex scenarios that actually provide reliable threat information to the security officer. Once this stage is reached, we will see a large body of research taking place on automated countermeasures, i.e. ensuring that attacks are dealt with efficiently and accurately, with as little human intervention as possible.

References

1. Debar, H., Curry, D., Fenstein, B.: Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition. Internet Draft (work in progress) (2005) http://search.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-14.txt.
2. Feinstein, B., Matthews, G., White, J.: The intrusion detection exchange protocol (idxp). Internet Draft (work in progress), expires April 22nd, 2003 (2002)
3. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Proceedings of LISA'99, Seattle, Washington, USA (1999)
4. Northcutt, S., Novak, J.: Network Intrusion Detection. 3 edn. QUE (2003) ISBN 0735712654.
5. Ptacek, T.H., Newsham, T.N.: Insertion, Evasion, and Denial of Service : Eluding Network Intrusion Detection. Secure Networks, Inc (1998)
6. Zhang, Y., Paxson, V.: Detecting stepping stones. In: Proceedings of the 9th USENIX Security Symposium, Denver, CO (2000)
7. Paxson, V.: An analysis of using reflectors for distributed denial-of-service attacks. Computer Communication Review **31** (2001)
8. Handley, M., Kreibich, C., Paxson, V.: Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In: Proceedings of the 10th USENIX Security Symposium, Washington, DC (2001)
9. Fieldings, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (1999)
10. Denning, D.E., Edwards, D.L., Jagannathan, R., Lunt, T.F., Neumann, P.G.: A prototype IDES — A Real-Time Intrusion Detection Expert System. Technical report, Computer Science Laboratory, SRI International (1987)
11. Snapp, S.R., Smaha, S.E.: Signature Analysis Model Definition and Formalism. In: Proc. Fourth Workshop on Computer Security Incident Handling, Denver, CO (1992)
12. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM **20** (1977) 762–772
13. Thomson, K.: Regular expression search algorithm. Communications of the ACM **11** (1968) 419 – 422

14. Denning, D.E., Neumann, P.G.: Requirements and model for IDES - a real-time intrusion detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (1985)
15. Denning, D.: An Intrusion-Detection Model. IEEE Transactions on Software Engineering **13** (1987) 222–232
16. Javitz, H.S., Valdez, A., Lunt, T.F., Tamaru, A., Tyson, M., Lowrance, J.: Next generation intrusion detection expert system (NIDES) - 1. statistical algorithms rationale - 2. rationale for proposed resolver. Technical Report A016–Rationales, SRI International, 333 Ravenswood Avenue, Menlo Park, CA (1993)
17. Droms, R.: Dynamic host configuration protocol. RFC 2131 (1997)
18. Morin, B., Mé, L., Debar, H., Ducassé, M.: M2D2 : A Formal Data Model for IDS Alert Correlation. In: Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID). (2002)
19. Viinikka, J., Debar, H.: Monitoring ids background noise using ewma control charts and alert information. In: Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004). Lecture Notes in Computer Science, Springer-Verlag (2004)
20. Egevang, K., Francis, P.: The ip network address translator (nat). RFC 1631 (1994)
21. Debar, H., Morin, B.: Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In: Proceedings of RAID 2002. (2002)
22. Porras, P.A., Fong, M.W., Valdes, A.: A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. In: Proceedings of RAID 2002. (2002) 95–114
23. Julisch, K.: Mining Alarm Clusters to Improve Alarm Handling Efficiency. In: Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC). (2001)
24. Morin, B., Debar, H.: Correlation of Intrusion Symptoms: an Application of Chronicles. In: Proceedings of RAID 2003. (2003) 94–112
25. Roberts, S.W.: Control Chart Tests Based On Geometric Moving Averages. Technometrics **1** (1959) 230–250
26. Ye, N., Vilbert, S., Chen, Q.: Computer Intrusion Detection Through EWMA for Autocorrelated and Uncorrelated Data. IEEE Transactions on Reliability **52** (2003) 75–82
27. Ye, N., Borror, C., Chang, Y.: EWMA Techniques for Computer Intrusion Detection Through Anomalous Changes In Event Intensity. Quality and Reliability Engineering International **18** (2002) 443–451
28. Mahadik, V.A., Wu, X., Reeves, D.S.: Detection of Denial of QoS Attacks Based on χ^2 Statistic and EWMA Control Chart. Online document, http://arqos.csc.ncsu.edu/papers.htm (2002) Submitted for Usenix 2002.

Security and Trust Requirements Engineering*

Paolo Giorgini, Fabio Massacci, and Nicola Zannone

Department of Information and Communication Technology,
University of Trento - Italy
{giorgini, massacci, zannone}@dit.unitn.it

Abstract. Integrating security concerns throughout the whole software development process is one of today's challenges in software and requirements engineering research. A challenge that so far has proved difficult to meet.

The major difficulty is that providing security does not only require to solve technical problems but also to reason on the organization as a whole. This makes the usage of traditional software engineering methologies difficult or unsatisfactory: most proposals focus on protection aspects of security and explicitly deal with low level protection mechanisms and only an handful of them show the ability of capturing the high-level organizational security requirements, without getting suddenly bogged down into security protocols or cryptography algorithms.

In this paper we critically review the state of the art in security requirements engineering and discuss the motivations that led us to propose the Secure Tropos methodology, a formal framework for modelling and analyzing security, that enhances the agent-oriented software development methodology i*/Tropos. We illustrate the Secure Tropos approach, a comprehensive case study, and discuss some later refinements of the Secure Tropos methodology to address some of its shortcomings. Finally, we introduce the ST-Tool, a CASE tool that supports our methodology.

1 Introduction

The last decades have seen an increasing awareness that security plays a key role in system development. Unfortunately, security modelling and policy work has been largely independent of system requirements and system models. The usual approach towards the inclusion of security within a system is to identify security requirements after system design. This is a critical problem [4], mainly because security mechanisms have to be fitted into a pre-existing design which may not be able to accommodate them [53]. Moreover, the implementation of the software system may assume security mechanisms that are simply not necessary. Alternatively, the implementation may introduce protection mechanisms that just hinder operation in a trusted domain that was not perceived as a trusted domain by the software engineer. Late analysis of security requirements can also generate conflicts between security needs and functional requirements of the system. Even with the growing interest in secure engineering, current methodologies for software (notably, information system) development do not address security-related problems [56], and fail to integrate successfully security concerns throughout the whole

* This article provides a survey of the research material which is described in [25,26,27,28].

development process. There has also been lack of interaction between researchers working on requirements modelling and security policy. Security is compromised most often not by breaking mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being utilized. Security mechanisms cannot be blindly inserted into a security-critical system. Instead, the overall system development process must take security concerns into account.

One of the current research challenge is to integrate security requirements analysis with the standard requirements process. A security requirements is a manifestation of a high-level organizational policy into the detailed requirements of a specific system. The integration of security engineering into a model-driven software development approach has advantages. Security requirements can be formulated and integrated into system designs at a high level of abstraction. In this way, it becomes possible to develop security aware applications that are designed with the goal of preventing violations of a security policy. At one side of the spectrum, the call for SE profession has been on good coding practices to avoid errors that could compromise the software's security (e.g., [60]). At the other extreme, the emphasis has been on securing the organization and its procedures (e.g., [4]).

Across the whole spectrum among these two extremes, modelling and analysis of security requirements has become a key challenge for Software Engineering [14,16], and it is the subject of this paper. In the next section (§2) we start a critical review of the existing proposals for security requirements engineering. Then, we discussed the key intuitions that lead us to propose an enhanced methodology and provide a description of Tropos concepts and describe the basic ones that we use for modelling security (§3). We introduce a scenario used as running example throughout the paper (§4). Then, we present a conceptual refinement of the framework (§5). Next, we formalize the security notions introduced in previous sections and define axioms and properties (§6). Finally, we introduce a CASE tool supporting our methodology (§7).

2 Security Requirements Engineering: A Survey

Strictly speaking of Software Engineering, modelling requirements is one of the key challenges that secure systems must meet (See Devanbu and Stubblebine's paper at ICSE [16]) and a number of researchers have been heeding the call. Proposals for Security Requirements Engineering can be classified under one of two classes: object-level and meta-level modelling.

The **object-level modelling** uses an off-the-shelves requirement framework, such as UML, KAOS, i*/Tropos, etc. and model in that framework a number of security requirements. The analysis features of the framework are then used to draw conclusions about the security modelling or to derive some guidance for the implementation.

The advantage of the object-level approach is that reasoning about security is virtually cost-free from the view point of the user: no new language to learn, all (good and bad) features of the modelling framework are immediately usable. If the framework is equipped with a formal semantics and formal reasoning procedures they are also inherited. In the formal framework the "security-notions" are indistinguishable from other objects, i.e., other requirements. This is also the major disadvantage: the link between security and functional requirements is lost and must be introduced by ad-hoc predicates or relationships by the designer. This makes particularly difficult the modelling of general relationships or rules (such as all processing of personal data should be authorized by the person whose data is being processed).

In the early requirement arenas we can list a number of works in the object-level field. For instance, in [62] security is frequently considered as a vague goal to be satisfied, while a precise description and enumeration of specific security properties and behavior is still missing. The work by Liu et al. [41] uses i*/Tropos for dealing with security and privacy requirements by introducing softgoal[1], as "Security" or "Privacy", to model these notions, and use dependencies analysis to check if the system is secure. In [5,6], general taxonomies for security and privacy are established. These can serve as a general knowledge repository for a knowledge-based goal refinement process. Another early RE example is [55], which presents a requirements process model, based upon reuse, together with a reusable template to organize security policies in a organization and a catalog filled with reusable personal data security requirements. Finally, He et al. [32] present a goal-driven framework for modelling privacy requirements in the role engineering process. The goal of this framework is to bridge the gap between competing stakeholders' security and privacy requirements, i.e., companies' privacy practices may be in conflict with user preferences. Privacy requirements are modelled as contexts and constraints of permissions and roles.

The **meta-level modelling** takes a off-the-shelves requirement framework as well as object-level modelling approach, but enhance it with linguistic constructs that capture security requirements. The analysis feature or implementation guidance of the framework must then be revised to allow for the new features.

The meta-level models trade off readiness for expressivity and compactness. The addition of suitable constructs makes usually the model more compact and more intuitive to use. This main advantage is coupled by the possibility of designing analysis features that are tailored to the security domain. This is also the key disadvantage: unless the addition of new features is carefully planned, the new framework needs the definition of analysis, semantics and reasoning procedures. To minimize this problem most sensible approaches try to design the framework in such a way that if one doesn't use the new features then one can still inherit all the old framework capabilities.

The need for conceptual models of security features have brought up a number of proposals especially in UML community. In approaches explicitly intended for security, we find the CORAS methodology for modelling risk and vulnerability [19]. Jürjens proposes UMLsec [35], an extension of the Unified Modelling Language (UML), for modelling security related features, such as confidentiality and access control. He proposes a concept for specifying requirements on confidentiality and integrity in analysis models based on UML. Lodderstedt et al. [42] present a UML-based modelling language (SecureUML). Their approach is focused on modelling access control policies and integrating them into a model-driven software development process. SecureUML is a modelling language designed to integrate information relevant to access control into application models defined with UML. The language builds on the access control model of RBAC [18,33,47,51] with additional support for specifying authorization constraints.

[1] Mostly non-functional requirements was satisfaction is fuzzy.

To address security concerns during software design, Doan et al. [17] incorporate Mandatory Access Control (MAC) into UML. Ray et al. [49] propose to model RBAC as a pattern by using UML diagram template. Further, they represent constraints on RBAC model through the Object Constraint Language. One of the major limitations of all these proposals is that they treat security in system-oriented terms, and do not support the modelling and analysis of security requirements at an organizational level. In other words, they are targeted to model a computer system and the policies and access control mechanisms it supports. In contrast, to understand the problem of security engineering we need to model the organization and social relationships between all actors involved in the system.

For early requirements, a preliminary modification of Tropos methodology has been proposed in [45]. In particular, this extension use security constraints and secure capabilities as basic concepts. However, [23] shows that the key missing concept is the separation of the notions of offering a service and ownership of the very same service. Further, it does not allow for the modelling of trust relationships.

Other approaches propose to model the behavior of attackers. Crook et al. [14] introduce the notion of anti-requirements to represent the requirements of malicious attackers. Anti-requirements are expressed in terms of the problem domain phenomena and are quantified existentially: an anti-requirement is satisfied when the security threats imposed by the attacker are realized in any one instance of the problem. Lin et al. [40] incorporate anti-requirements into abuse frames. The purpose of abuse frames is to represents security threats and to facilitate the analysis of the conditions in the system in which a security violation occurs. They allow the examination of a system's vulnerabilities to different kinds of security threats in a bounded context. Abuse frames share the same notation as the normal problem frames, but each domain is now associated with a different meaning. McDermott and Fox adapt use cases [44] to capture and analyze security requirements, and they call the adaption an abuse case model. An abuse case is an interaction between a system and one or more actors, where the results of the interaction are harmful to the system, or one of the stakeholders of the system. Guttorm and Opdahl [52] propose to model security by defining misuse cases, the inverse of UML use cases, which describe functions that the system should not allow. This new construct makes it possible to represent actions that the system should prevent together with those actions which it should support. Moving towards early requirements, an extension of the KAOS framework is presented in [59] where the notion of obstacle is introduced. KAOS uses the notion of goal as a set of desired behaviors. Likewise, an obstacle defines a set of undesirable behaviors. Therefore, the negation of such obstacles is used to determine preconditions for the goal to be achieved. Although obstacle are sufficient for modelling accidental, non-intentional obstacles to security goals, they appear too limited for modelling and resolving malicious, intentional obstacles. To this end, van Lamsweerde et al. [58] introduce the notion of anti-requirements and anti-goals that are, respectively, the requirements of malicious attackers and the intentional obstacles to security goals.

2.1 Towards a "Terra Incognita": Why a New Methodology Is Needed

Most proposals in the literature focus on protection aspects of security and explicitly deal with a series of security services (integrity, availability etc.) and related protection mechanisms (such as passwords, or cryptographic mechanisms). If we look at the requirement refinement process of many proposals, we find out that at certain stage a leap is made: we have a system with no security features consisting of high-level functionalities, and the next refinement shows encryption, access control and authentication. The modelling process should instead makes it clear why encryption, access control and authentication are necessary. What is missing is capturing the high-level security requirements, without getting suddenly bogged down into security solutions or cryptographic algorithms.

Early requirements requires to reason about trust relationships, ownership and delegation of authority besides the traditional notion of functional dependencies. The first step in this direction is described the papers [25,26] which extended the i*/Tropos modelling framework [10] to introduces concepts such as ownership, trust, and delegation within a requirements modelling framework and shows how security and trust requirements can be derived and analyzed.

After a large case study on the compliance of an ISO-17799-like security policy [43] with Italian privacy legislation, it was concluded that the concepts offered by Secure Tropos are the right ones but are too coarse-grained to capture important security facets.

The first observation is that for pragmatic reasons, it is often the case that services and permissions are delegated to actors who are not trusted. Nevertheless, the overall system is still considered secure if there is a way to hold such delegations accountable by monitoring their (wrong) doings.

The second observation is that trust in actors (or lack thereof) comes in different flavors: we may trust an actor to actually deliver the services we require (taking into account skills and/or commitment), or to honor granted permissions. In trust management and authorization settings (e.g. [7,15,38]) one only finds delegations of permission (through authorization). Requirements of availability are equally important, however, and can only be captured by modelling delegation of execution (where one actor delegates to another the responsibility to execute a service).

Finally, in a recent study, the majority of Information Security Administrators said that their biggest worry is employee negligence and abuse [48]. Internal attacks can be more harmful than external attacks since they are being performed by trusted users that can bypass access control mechanisms. So, we need models that compare the structure of the organization (roles and relations among them) with the concrete instance of the organization (agents playing some roles in the organization and relations among them). The original Tropos proposal involves two different levels of analysis: social and individual. In the organization level we analyze roles and positions of the organization, whereas in individual level the focus is on single agents. Of course there is no explicit separation between the two levels, and so Tropos is not able to maintain the consistency between the social level (roles and positions) and the individual level (agent).

3 Secure Tropos: A Goal Oriented SRE Methodology

Secure Tropos [25,26] enhances the agent-oriented software development methodology i*/Tropos [10]. The Tropos methodology is intended to support all analysis and design activities in the software development process, from the application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, that is incrementally refined and extended, providing a common interface to the various software development activities, as well as a basis for documentation and evolution of the software.

Tropos uses the concepts of actor, goal, plan, resource and social dependency for defining the obligations of actors (dependees) to other actors (dependers). A goal represents the strategic interests of an actor. A plan specifies a particular course of action that produces a desired effect, and can be executed in order to satisfy a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a plan, or deliver a resource. Tropos is well suited to to describe both an organization and an IT system. As we already discussed, in [23] we have argued that it lacks the ability to capture at the same time the functional and security features of the organization, and hence the new proposal.

In the following, we introduce Secure Tropos as an extension of the requirements analysis phase of the Tropos Methodology. Basic concepts, relationships, and models will be presented along the methodological approach and the modelling activities.

3.1 Requirement Analysis Phase

Requirement analysis represents the initial phase in many software engineering methodologies. Similarly to other software engineering approaches, in Tropos the final goal of requirement analysis is to provide a set of functional and non-functional requirements for the system-to-be. The requirements analysis in Tropos is split in two main phases: *Early Requirements* and *Late Requirements* analysis. Both share the same conceptual and methodological approach. Thus most of the ideas introduced for early requirements analysis are used for late requirements as well.

More precisely, during the first phase, the requirements engineer identifies the domain stakeholders and models them as social actors, who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. By clearly defining these dependencies, it is then possible to state the *why*, beside the *what* and *how*, of the system functionalities and, as a last result, to verify how the final implementation matches the real needs.

In the *Late Requirements* analysis, the conceptual model is extended including a new actor, which represents the system, and a number of dependencies with other actors part of the environment. These dependencies define all the functional and non-functional requirements of the system-to-be.

3.2 The Key Concepts

Models in Tropos are acquired as instances of a *conceptual metamodel* resting on the following concepts/relationships:

– **Actor**, which models an entity that has strategic goals and intentionality within the system or the organizational setting. An actor represents a physical or a software *agent* as well as a *role* or *position*. While we assume the classical AI definition of software agent, that is, a software having properties such as autonomy, social ability, reactivity, proactivity, as given, for instance in [46], in Tropos we define a *role* as an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor, and a *position* represents a set of roles, typically played by one agent. An agent can occupy a position, while a position is said to cover a role. A discussion on this issue can be found in [61].

– **Goal**, which represents actors' strategic interests. We distinguish hard goals from softgoals, the second having no clear-cut definition and/or criteria for deciding whether they are satisfied or not. According to [12], this different nature of achievement is underlined by saying that goals are *satisfied* while softgoals are *satisficed*. Softgoals are typically used to model non-functional requirements.
– **Plan**, which represents, at an abstract level, a way of doing something. The execution of plan can be a means for satisfying a goal or for satisficing a softgoal.
– **Resource**, which represents a physical or an informational entity. The main difference with an agent is that a resource has not intentionality.
– **Dependency** between two actors, which indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource. The former actor is called the *depender*, while the latter is called the *dependee*. The object around which the dependency centers is called *dependum*. In general, by depending on another actor for a dependum, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Four new relationships have been introduced in Secure Tropos:

– **Ownership**, which indicates that the actor is the legitimate owner of some goal, some plan, or some resource. The owner has full authority concerning to achieve his goal, execute his plan, or use his resource, and he can also delegate this authority to other actors.
– **Provisioning**, which indicates that the actor has the capability to achieve some goal, execute some plan, or deliver a resource.
– **Trust**, between two actors, which indicates the believe of one actor that the other does not misuse some goal, some plan, or some resource. The former actor is called the *truster*, while the latter is called the *trustee*. The object around which the dependency centers is called *trustum*. In general, by trusting another actor for a trustum, an actor is sure that the trustum is properly used. At the same time, the truster becomes vulnerable. If the trustee misuses the trustum, the truster cannot guarantee to achieve some goal, execute some plan, or deliver a resource securely.
– **Delegation**, between two actors, which indicates that one actor delegates to the other the permission to achieve some goal, execute some plan, or use a resource. The former actor is called the *delegater*, while the latter is called the *delegatee*. The object around which the dependency centers is called *delegatum*. In general, delegation marks a formal passage in the domain that is currently modelled by the requirements engineers. This would be matched by the issuance of a delegation

certificate such as digital credential or a letter if we are delegating permission or by a call to an external procedure if we are delegating execution.

3.3 Modelling Activities

Various activities contribute to the acquisition of a first early requirement model, to its refinement and to its evolution into subsequent models. They are:

- **Actor modelling**, which consists of identifying and analyzing both the actors of the environment and the system's actors and agents. In particular, in the early requirement phase actor modelling focuses on modelling the application domain stakeholders and their intentions as social actors which want to achieve goals. During late requirement, actor modelling focuses on the definition of the system-to-be actor.
- **Dependency modelling**, which consists of identifying actors which depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. In particular, in the early requirement phase, it focuses on modelling goal dependencies between social actors of the organizational setting. New dependencies are elicited and added to the model upon goal analysis performed during the goal modelling activity discussed below. During late requirements analysis, dependency modelling focuses on analyzing the dependencies of the system-to-be actor. In the architectural design phase, data and control flows between sub-actors of the system-to-be actors are modelled in terms of dependencies, providing the basis for the capability modelling that will start later in architectural design together with the mapping of system actors to agents.

A graphical representation of the model obtained following these modelling activities is given through *actor diagrams*, called *dependency model*, which describe the actors (depicted as circles), their goals (depicted as ovals and cloud shapes) and the network of dependency relationships among actors (two arrowed lines connected by a graphical symbol varying according to the dependum: a goal, a plan or a resource).

- **Goal and plan modelling** rests on the analysis of an actor goals, conducted from the point of view of the actor, by using three basic reasoning techniques: *means-end analysis*, *contribution analysis*, and *AND/OR decomposition*. In particular, means-end analysis aims at identifying plans, resources and softgoals that provide means for achieving a goal. Contribution analysis identifies goals that can contribute positively or negatively in the fulfillment of the goal to be analyzed. In a sense, it can be considered as an extension of means-end analysis, with goals as means. AND/OR decomposition combines AND and OR decompositions of a root goal into subgoals, modelling a finer goal structure. Goal modelling is applied to early and late requirement models in order to refine them and to elicit new dependencies. During architectural design, it contributes to motivate the first decomposition of the system-to-be actors into a set of sub-actors.

A graphical representation of goal and plan modelling is given through *goal diagrams*, which appears as a balloon within which goals of a specific actor are analyzed and dependencies with other actors are established. Goals are decomposed into subgoals and positive/negative contributions of subgoals to goals are specified. Goal decomposition can be closed through a means-end analysis aimed at identifying plans, resources and softgoals that provide means for achieving the goal.

The revised methodology introduces new steps that replaces the old ones:

- **Trust modelling** which consists of identifying actors which trust other actors for goal, plans, and resources, and actors which own goal, plans, and resources. In particular, in the early requirement phase, it focuses on modelling trust relations between social actors of the organizational setting. New trust relations are elicited and added to the model upon the refinement activities discussed above. During late requirements analysis, trust modelling focuses on analyzing the trust relations of the system-to-be actor.
- **Delegation modelling** which consists of identifying actors which delegate to other actors the permission and task of execution on goals, plans, and resources. In particular, in the early requirement phase, it focuses on modelling delegations between social actors of the organizational setting. New delegations are elicited and added to the model upon the refinement activities discussed above. During late requirements analysis, delegation modelling focuses on analyzing the delegations involving the system-to-be actor.

A graphical representation of the models obtained following these last two modelling activities is given through two different kinds of actor diagrams: *trust model*, and *trust management implementation*. Essentially, the first represents the trust network among the actors involved in the system and the latter represents which permissions are effectively delegated by actors and which actors receive such permissions. These models use the same notation for actors, goals, plans and resource used during dependency modelling. The old dependency model is replaced by the delegation of execution model.

3.4 Process

The overall methodological process is an iterative process in which the above presented modelling activities are used to produce different kinds of actor and goal diagrams. Table 1 summarizes the process activities and the diagrams elaborated in each activity. The diagrams produced in one activity are used as input for the other activities.

Table 1. Activities and diagrams produced during the analysis process

Activity	**Diagrams produced**
1. Actor modelling	Actor diagram: actors and their goals are elicited
2old. Dependency modelling	Actor diagram: dependencies between actors are discovered
2a. Trust modelling	Trust diagram: trust relationships between actors are discovered
2b. Delegation modelling	Trust management implementation diagram: delegations between actors are modelled
3. Goal modelling	Goal diagram: actor goals are analyzed
4. Plan modelling	Goal diagram: plans associated to goals are analyzed

The process starts with the actor modelling activity (1) in which the relevant actors (stackholders and existing software (sub)systems) are elicited and modelled with their goals. The actor diagram produced after this activity is used as input for the dependency modelling activity (2old), where the dependencies between the actors are discovered and established. The resulting actor diagram can be either used to further revise the initial actor diagram or as input for the next activity (3). Goal modelling focus on the goals associated to each actor of the actor diagram and it analyzes them using various forms of analysis as described earlier. During the analysis new dependencies can be discovered so to revise and enrich the model produced in (2). Goal diagram is also used as input for plan modelling activity (4), where each single goal is analyzed in terms of plans that can be used for its fulfillment. Plans are analyzed in details and new dependencies between actor can emerge so to require a new dependency analysis (2old). In the new model we start with the trust model (2a), which in turn can require a further goal analysis (3). Dependency can then be devised by as in step (2old) by modelling delegation and trust (if any). This may require further goal analysis (3) as in the standard Tropos project. The final trust model is used to develop the trust management implementation for permission (2b), that finally can be used to revise the delegation of execution model (2b) and the trust model (2a). The process ends when no further analysis are needed.

4 Using SRE for Compliance with Data Privacy Legislation

To instantiate some of the above mentioned concepts we show some fragments of a complex case study: the compliance to the Italian legislation on Privacy and Data Protection by the University of Trento, leading to the definition and analysis of an ISO-17799-like security management scheme (we refer to [43] for more detials). The final EU and Italian legislation systematized the norms on privacy and data protection by specifying

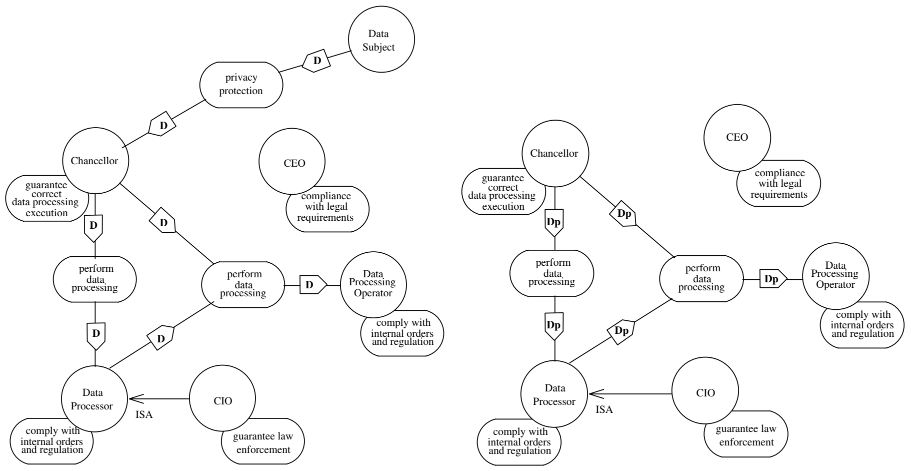
– the definitions of personal data, sensitive data, and data processing,
– the definitions of all entities involved in data processing, their roles and responsibilities (controller, processor, operator, subject),
– the obligations relating to public and private data controllers with specific reference to the legitimate purpose of data processing and the adoption of minimal precautionary security measures to minimize the risks on data.

4.1 Modelling Actors

The first activity in the early requirements phase is actors' modelling. In our example we can list some of them:

Data Controller determines the purposes and means of the processing of personal data. In the University, the data controller is identified with Chancellor (as the post-holder is also the legal representative of the University).

Data Processor monitors personal data processing on behalf of the controller. In the University, these are:
  – Faculty Deans;
  – Head of Department;


Data Subject
D
privacy protection
D
Chancellor
CEO
guarantee correct data processing execution
compliance with legal requirements
D
D
perform data processing
perform data processing
D
Data Processing Operator
comply with internal orders and regulation
D
D
Data Processor
CIO
ISA
comply with internal orders and regulation
guarantee law enforcement
Dp

(a) Actor Diagram

(b) Trust Management Implementation diagram

Fig. 1. Actor Diagrams

- Central Directorate Managers, and in particular with:
  - Chief Executive Officer (CEO);
  - Chief Information Officer (CIO).

Data Processing Operator is appointed by the data controller or processor to perform the operations related to the data processing or to manage and maintain the information systems and services. At University of Trento, these are:
- Personal Data Processing Operator;
- Database Security Operator;
- Network Security Operator.

Data Subject is the natural or legal person to whom the personal data are related. In the Secure Tropos terminology, this is the legitimate owner of the data.

4.2 Modelling Dependencies and Delegation

The analysis proceeds introducing the functional dependencies and the delegation of permission between actors and the consequent integrated security and functional requirements. Figure 1(a) and Figure 1(b) show the functional dependency model and the trust management implementation. We use delegation of permission (**Dp**) to model the actual transfer of rights in some form (e.g. a digital certificate, a signed paper, etc.), and **D** for functional dependency.

In the functional dependency model, *Chancellor* is associated with a single relevant goal: *guarantee correct data processing execution*, while *CEO* has an associated goal *compliance with legal requirements*. Along similar lines, *Data Processor* and *Data Processing Operator* want to *comply with internal orders and regulation*, while *CIO*, wants to *guarantee law enforcement*. Finally, the diagram includes some functional dependencies: *Data Subject* depends on *Chancellor* for *privacy protection* goal; *Chancellor* depends on *Data Processor* and *Data Processing Operator* to *perform data processing*; and, in turn, *Data Processor* depends on *Data Processing Operator* for it.

In the actor diagram, *Chancellor* is associated with a single relevant goal: *guarantee correct data processing execution*, while *CEO* has an associated goal *compliance with legal requirements*. Along similar lines, *Data Processor* and *Data Processing Operator* want to *comply with internal orders and regulation*, while *CIO*, wants to *guarantee law enforcement*. Finally, the diagram includes some delegations of execution: *Data Subject* delegates to *Chancellor* the goal *privacy protection*; *Chancellor* delegates to *Data Processor* and *Data Processing Operator* the goal *perform data processing*; and, in turn, *Data Processor* delegates it to *Data Processing Operator*.

In the trust management implementation diagram, *Chancellor* delegates permissions to *perform data processing* to *Data Processor* and *Data Processing Operator*. In turn, *Data Processor* delegates permissions to *perform data processing* to *Data Processing Operator*.

At this stage, the analysis already reveals a number of pitfalls in the actual document template provided by the ministry's agency. The most notable one is the absolute absence of functional dependencies between the Chancellor and the CEO, who is actually the one who runs the administration. Such functional dependency is present in the Universities statutes, but not here (an apparently unrelated document).

Another missing part in the trust management implementation is the delegation of permission from the data subject. This can be also automatically spotted with the techniques developed in [26]. Somehow paradoxically (for a document template enacted in fulfillment of a Data Protection Act) the process of acquisition of data (and the relative authorization) is neither mentioned nor forseen. In practice this gap is solved by the University by a blanket authorization: in all the paper or electronic data collection steps a signature is required to authorize the processing of data in compliance with the privacy legislation.

4.3 Goal Refinement

In this paper, we present a goal analysis for *Data Processor* and refer to [43] for an accurate analysis of the other actors involved in the system.

Figure 2 shows the goal analysis for Data Processor, relative to the goal *comply with internal orders and regulation*. This goal is decomposed into *provide for appointing data processing operators*, *security control* and *adopt security measures* for which *Data Processor* depends on *CIO* and *CEO*. The goal *provide for appointing data processing operators* is decomposed into three goals: *identify data processing operators* for which *Data Processor* depends on *Data Processing Operator*, *give instructions to data processing operators* for which *Data Processing Operator* depends on *Data Processor*, and *enable access profile* for which *Data Processing Operator* depends on *Data Processor* and, in turn, *Data Processor* depends on *CIO*. The goal *enable access profile* is decomposed into *assign access profile*, *assign ID and password* which *Data Processor* depends on *CIO*, and *communicate name of security operators* for which *CIO* depends on *Data Processor*. The goal *security control* is decomposed into *monitor security mea-*

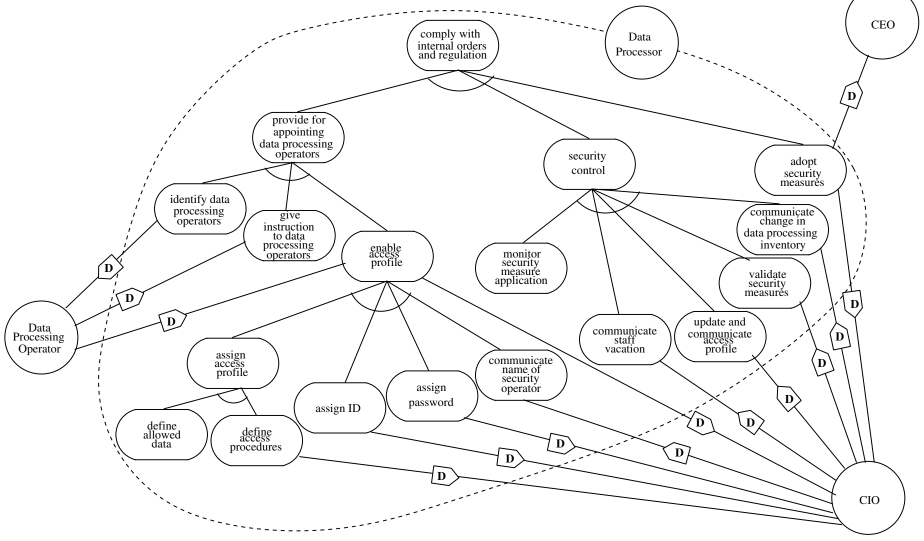

comply with internal orders and regulation
Data Processor
CEO
provide for appointing data processing operators
security control
adopt security measures
identify data processing operators
give instruction to data processing operators
enable access profile
monitor security measure application
communicate change in data processing inventory
validate security measures
Data Processing Operator
communicate staff vacation
update and communicate access profile
assign access profile
communicate name of security operator
assign ID
assign password
define allowed data
define access procedures
CIO
D

Fig. 2. Functional Dependency Model for Data Processor

sure application and other goals, such as *communicate staff vacation* and *update and communicate access profile*, for which *CIO* depends on *Data Processor*, *update and communicate access profile*.

Figure 3 shows the trust management implementation for *Data Processor*. The diagram displays that *Data Processor* delegates *mail with instructions* to *Data Processing Operator*. Further, *Data Processor* delegates the *list of name of security operators*, *list of employees in vacation*, *access profile* and *data processing inventory* to *CIO*. Finally, *Data Processor*receives from *CEO* the *list of security measures*.

5 The Plot Thickens: Refining Delegation and Trust

In this section, we introduce a conceptual refinement of the delegation and trust relationships, that will allow us to capture and model important security facets [27,28].

In order to explain the conceptual refinement we will use examples based on the case study presented in previous section. For the sake of readability we introduce here dramatis personae [2] together with the rules they play:

Alice is an administrative officer, for example of the teaching evaluation office;
Bob, Bert, and Bill are students;
Sam is (the manager of) the student IT system;
Paul and Peter are professors.

[2] This impersonation is actually closer to reality than one may think: the law requires the assignment of responsibility of each IT sub-system to a person.

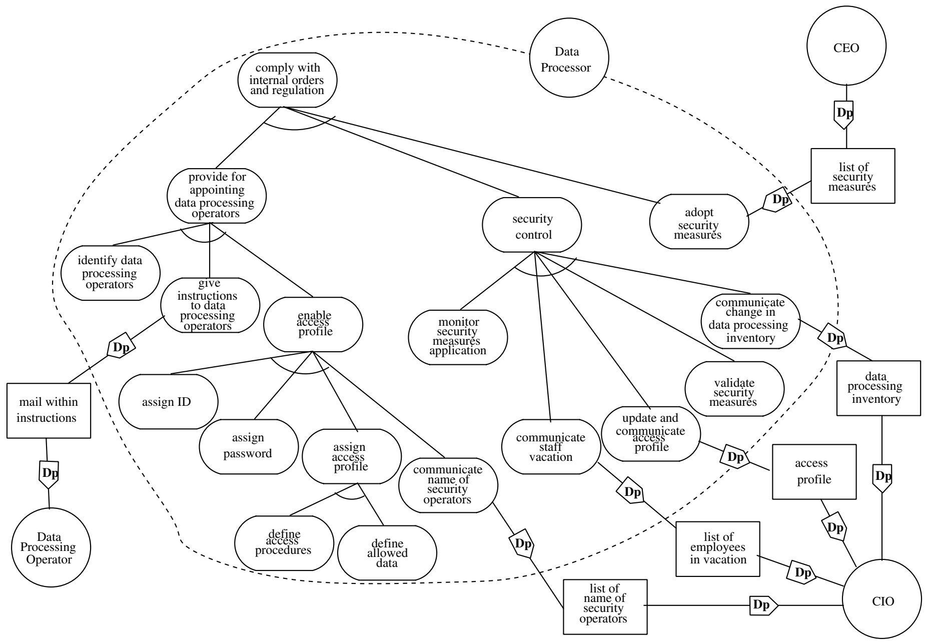

Data Processor
CEO
comply with internal orders and regulation
Dp
list of security measures
provide for appointing data processing operators
security control
adopt security measures
identify data processing operators
give instructions to data processing operators
enable access profile
monitor security measures application
communicate change in data processing inventory
data processing inventory
mail within instructions
assign ID
validate security measures
assign password
assign access profile
communicate name of security operators
communicate staff vacation
update and communicate access profile
access profile
Data Processing Operator
define access procedures
define allowed data
list of employees in vacation
list of name of security operators
CIO

Fig. 3. Trust Management Implementation for Data Processor

5.1 Execution vs Permission

Example 1. Alice is interested in gathering data on students' performance, for which she depends on Sam. Bob owns his sensitive personal information, such as his student careers. Bob delegates permission to provide information about his career to Sam on condition that his privacy is protected (i.e., his identity is not revealed).

In this scenario, there is a difference of relationship between Alice–Sam and Bob–Sam. This difference is due to a difference in the type of delegation.

Example 2. Bob *delegates permission* to Sam to provide only the relevant information and nothing else. On the other hand, Alice, who wants student data, *delegates* the *execution* of her goal to Sam. According to Alice, Sam should at least fulfill the goal she requires. She is not interested in what Sam does with Bob's trust, apart from getting her information. The major worry of Alice is availability whereas Bob cares about authorization. In other words, Alice's major concerns would be that tasks are delegated to people that can actually do them, whereas Bob would be concerned that subtasks are given to trusted people who will not misuse the permissions they have acquired.

If we want to check functional and security requirements consistency, it is essential to distinguish between these two notions of delegation. We use **at-most delegation** when the delegater wants the delegatee at most achieves the goal, execute the plan, or furnishes the resource. This is *delegation of permission*, where the delegatee thinks

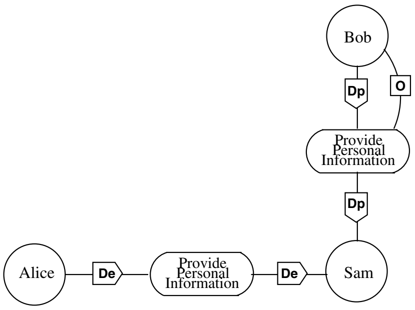


Fig. 4. At-least and At-most Delegation

"I have the permission to achieve the goal (but I do not need to)", whereas **at-least delegation** means that the delegater wants the delegatee to achieve at least the goal. This is the *delegation of execution*. The delegatee thinks, "Now, I have to get the goal fulfilled (let's start working)". In the pictorial representation of Fig. 4 we represent these relationship as edges respectively labeled by **Dp** and **De**.

Further, we want to separate the concepts of trust and delegation, as we might need to model systems where some actors must delegate permission or execution to other actors they don't trust. Also in this case it is convenient to have a suitable distinction for trust in managing permission and trust in managing execution. The meaning of **at-most trust** is that an actor (truster) trusts that another actor (trustee) at most fulfills the goal but will not overstep it. The meaning of **at-least trust** is that an actor (truster) trusts that another actor (trustee) at least fulfills the goal.

Example 3. At-most trust is good for permissions: Bob trusts Sam to remain within certain bounds. He may delegate Sam more permissions than actually needed because Sam will not abuse them. At-least trust fits execution. Alice believe Sam can accomplish her plans and possibly more.

The new Secure Tropos concepts "explain" the classical Tropos dependency between two actors in terms of trust and delegation (Fig. 5).

Indeed, the semantics associated to the Tropos dependency states that there is an actor, the dependee, that wants to achieve a specific goal (perform a task or have a resource) and there is another actor, the depender, that is able to satisfy the goal (perform the task or deliver the resource). The two actors get an agreement and a goal (task or resource) dependency is established between the two. The implicit assumption is that after the agreement the depender will be responsible for the goal and will do the best to achieve it.

The distinction between *execution* and *permission* allows us to define a dependency in terms of trust and delegation. In particular, when the dependum is a goal or a plan we have delegation and trust of execution, whereas when the dependum is a resource we have delegation and trust of permission. In symbols:

$$\mathsf{depends}(A, B, S) \iff \mathsf{delegate}(exec, A, B, S) \land \mathsf{trust}(exec, A, B, S) \tag{1}$$

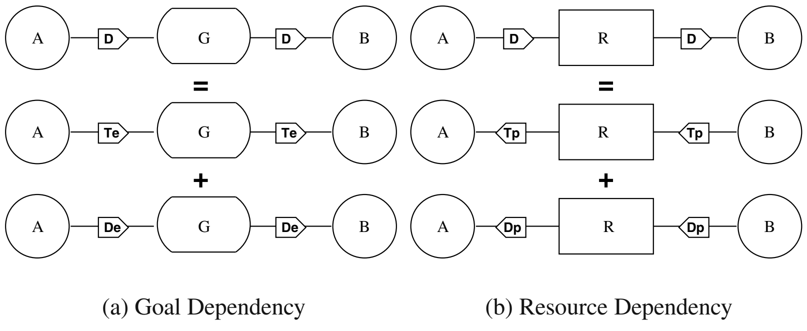


Fig. 5. Tropos dependency in terms of Secure Tropos

where S is a goal or a plan, and

$$\mathsf{depends}(A, B, S) \iff \mathsf{delegate}(perm, ID, B, A)\, S \wedge \mathsf{trust}(perm, B, A, S) \qquad (2)$$

where S is a resource. A graphical representation of these formulas is given, respectively, in Fig. 5(a) and in Fig. 5(b). These diagrams use the label **D** for Tropos dependency and labels **Te** and **Tp**, respectively for trust of execution and trust of permission. Notice also from Fig. 5 that the same dependency is mapped into differently oriented relations at the lower level.

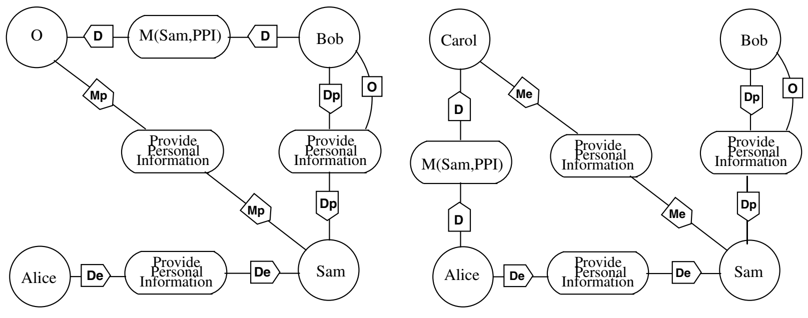
5.2 Introducing Distrust

Another refinement is the introduction of negative authorizations which are needed for some scenarios. Tropos already accommodates the notion of positive or negative contribution of goals to the fulfillment of other goals. We use negative authorizations to help the designer in shaping the perimeter of positive trust to avoid incautious delegation certificates that may give more powers than desired.

Suppose that an actor should not be entitled to achieve a goal, perform a plan, or delivery a resource. In situations where authorization administration is decentralized, an actor possessing the right to achieve a goal, execute a plan, or delivery a resource, can delegate the authorization to do that to the wrong actor.

We propose an explicit distrust relationship as an approach for handling this type of situations. This is also sound from a cognitive point of view if we follow the definition of trust given by [11]: trust is a mental state based on a set of beliefs. We can say that if, on your own knowledge, you feel to trust me, then you trust me. Similarly, if you feel like distrusting me, then you distrust me. Obviously, there are various reasons for distrusting agents such as unskillfulness, unreliability and abuse, but these situations are not treated here.

As we have done for trust, we also distinguish between distrust of execution and distrust of permission. The graphical diagrams presented in this paper use the labels **Se** and **Sp**, respectively, for distrust of execution and distrust of permission. In the case there is no explicit trust relationship between agents, the label “**?**” is used.


O
D
M(Sam,PPI)
D
Bob
Mp
Dp
O
Provide Personal Information
Provide Personal Information
Mp
Dp
Alice
De
Provide Personal Information
De
Sam
Carol
Me
Bob
D
Dp
O
M(Sam,PPI)
Provide Personal Information
Provide Personal Information
D
Me
Dp
Alice
De
Provide Personal Information
De
Sam

(a) At-most Monitor (b) At-least Monitor

Fig. 6. Monitoring

5.3 Monitoring

When work needs to be delegated even when there is no trust, then monitoring can offer a surrogate for trust. Accordingly to Gans's et al. [21], the existence of distrust can be tolerated with an additional overhead of monitoring the untrustworthy delegatee. Here we refine Gans's et al. intuition integrating it in our framework.

The goal of an actor playing the role of *monitor* is to check for the violation of trust[3]. The act of monitoring can be done by the delegater himself[4], or he can delegate it to some other actors to get it done. Depending on the type of delegation, we have two different kinds of monitors: **at-most monitor** and **at-least monitor**. Consider the situation presented in Fig. 4.

Example 4. Suppose that there is no trust between Bob and Sam for the goal "maintain privacy", but the student must delegate permission nonetheless. In this case, he depends (**D**) on the ombudsman (*O*) for monitoring if Sam transgresses her permissions. This is shown in Fig. 6(a)) with an at-most monitor (monitor for permission – **Mp**) relationship between the ombudsman and Sam.

Example 5. If Alice is not confident that Sam will provide updated information, she may delegate to her secretary Carol the task of confirming with, or nagging Sam to insert new data as soon as it becomes available. This is shown in Fig. 6(b)) with an at-least monitor (monitor for execution – **Me**) relationship between Carol and Sam.

Another important distinction that emerges when we use a monitor is related to what we have to monitor. If we are monitoring a plan (i.e., a specific sequence of actions), the Monitor has to check if Sam executes the actions of the plan. What happens if Sam delegates the task or some of its subtasks to other actors?

[3] Indeed, monitoring could also be used for the evaluation of the fulfillment of a goal assigned to a trusted actor.

[4] Intuitively, this is like saying that fellow is unreliable, I'll give him the job but keep an eye on him myself".

Example 6. To achieve the goal delegated to him in Example 5, Sam will issue a letter to the head of each student secretariat office so that student marks are entered into the system within 30 days from the date that exams have taken place.

A solution to this problem is to extend the monitoring to all sublevels of delegation until the level where the actual execution takes place. So, there will be a monitor relationship between the Monitor and all the actor involved in the execution of at least a part of the task.

Example 7. To reach the objective of 30 days requires that professors return to the office assigned marks. This is a further step of delegation of execution. Then, the actor responsible at the office, beside actually monitoring his employees, may also assign the task of reminding professors that they must return on time their mark sheets.

Notice that monitoring as such is not a primitive construct. It can be captured by other constructs within our modelling framework. Specifically, every goal, plan and resource will either be delegated during the design process to a trusted actor, or it will be delegated to an untrusted one, in which case the delegatee will be monitored by a trusted actor.

On the formal model this corresponds to a design pattern formalized in terms of additional axioms that allow us to conclude that an actor is confident that a goal will be executed, a plan will be performed or a resource will be furnished, or a permission will not be abused even if existing trust relations suggest otherwise.

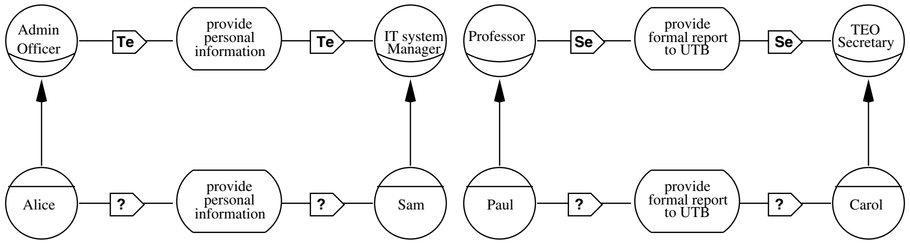
Once we see monitoring as a simple design solution (essentially a security pattern) we can treat monitoring goals just as any other goal. So they can be further subject to refinement, delegation of execution and delegation of permission. Trust relationships linked to monitoring can then be captured with standard constructs. For example, monitoring often requires having permission to access monitored data or personnel. This itself may create problems of permission and authorization that can be model in the framework.

5.4 Social vs Individual Trust

When we model and analyze functional trust and security relationships, it is possible that such requirements are given only at individual level or at social level and that there is a mismatch between the levels. Let us see why this is needed with examples drawn from the same domain.

Example 8. According the University policy, administrative officers should trust managers of IT systems to get information they need to perform their duties (Fig. 7(a)). Sam is the new manager of the student IT system and Alice has never met him before. Still, Alice should trust Sam for getting student personal information in order to guarantee the availability of the goal.

Example 9. Professors should not rely on the teaching evaluation officer secretary for providing a formal report to the University Teaching Board (Fig. 7(b)). Here, Paul and Carol don't know each other. Then, Paul should distrust Carol for providing a formal report to University Teaching Board.


Admin Officer
Te
provide personal information
Te
IT system Manager
Alice
?
provide personal information
?
Sam
Professor
Se
provide formal report to UTB
Se
TEO Secretary
Paul
?
provide formal report to UTB
?
Carol

(a) Trust of execution (b) Distrust of execution

Fig. 7. Missing (dis)trust relations at individual level

We don't consider the case in which the relations are missing at social level because this level represents the structure of the organization which should be described explicitly in the requirements. The presence of a large number of trust relations at individual level that is not matched by a social level may be an indicator of a missing link at social level (or of a problem in the organization for distrust relations). On the contrary, Hannoun et al. [31] propose to detect the inadequacy of an organization regarding the relations existing among the agents involved in the system.

In [26] we have only considered when trust is explicit, and we have not distinguished the case where there is explicit distrust and the case where no trust relation is given. Contrarily, in this paper we take in consideration all these three possibilities. The presence of positive and negative authorization at the same time could generate some conflicts on trust relationships. We define a *trust conflict* the situation where there are both a positive and a negative trust relation between two actors for the same trustum. Next, formal definitions are given.

Definition 1. *A conflict on trust of execution occurs when*

$$\exists x, y \in \texttt{Agent}\ \exists s \in \texttt{Goal} \cup \texttt{Task} \cup \texttt{Resource} \mid \mathsf{trust}(exec, x, y, s) \wedge \mathsf{distrust}(exec, x, y, s)$$

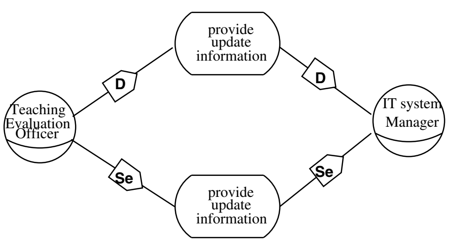
Definition 2. *A conflict on trust of permission occurs when*

$$\exists x, y \in \texttt{Agent}\ \exists s \in \texttt{Goal} \cup \texttt{Task} \cup \texttt{Resource} \mid \mathsf{trust}(perm, x, y, s) \wedge \mathsf{distrust}(perm, x, y, s)$$

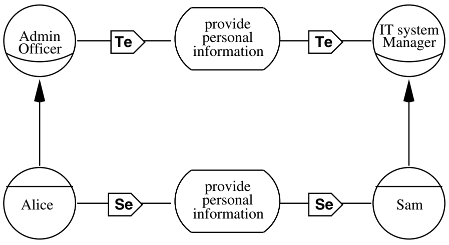
A trust conflict may exist, for example, since system designers wrongly put both a (implicit) trust relation and the corresponding distrust relation.

Example 10. The teaching evaluation officer depends on the manager of the student IT system for providing update information, but the latter is distrusted for such goal (Figure 8(a)).

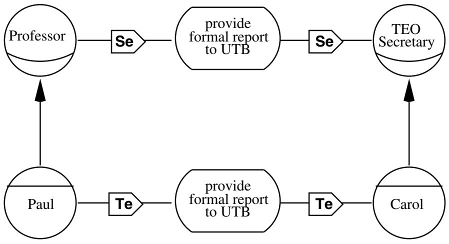
When we model and analyze security requirements, it is also possible that such requirements are specified at both individual and social levels, they could be in contrast with each other.


Teaching Evaluation Officer
provide update information
D
D
IT system Manager
Se
Se
provide update information

(a) Conflict due to implicit trust



(b) Social Trust vs Individual Distrust



(c) Social Distrust vs Individual Trust

Fig. 8. Conflicts on (dis)trust relations

Example 11. Consider again Example 8. What happens if Alice had some problems with Sam in the past and he doesn't trust her? This scenario is presented in Fig.8(b).

Example 12. Consider again Example 9. What happen if Paul trusts Carol for providing a formal report to University Teaching Board? This scenario is presented in Fig. 8(c).

Monitoring, which we have introduced early in thic paper, is a good solution to this extent. So we don't need to add anything to the system just to cope with trust conflicts.

Example 13. Referring to Example 11, we believe that Alice should monitor (or delegate this task to another actor) whether Sam does what he has to do since the organization imposes her to trust him, but it is not her own choice.

6 Automated Reasoning in SRE

We use Datalog [1] as the underlying semantic framework, also to be close to the semantics of other frameworks for trust or security (e.g. [15,39,50]).

A Datalog program is a set of rules of the form L:- $L_1 \wedge ... \wedge L_n$ where L, called head, is a positive literal and $L_1, ..., L_n$ are literals and they are called body. Intuitively, if $L_1, ..., L_n$ are true in the model then L must be true in the model. We use the notation

Table 2. Predicates

General predicates

goal($\texttt{Goal}: g$)
plan($\texttt{Plan}: t$)
resource($\texttt{Resource}: r$)

agent($\texttt{Agent}: a$)
position($\texttt{Position}: a$)
role($\texttt{Role}: a$)

play($\texttt{Agent}: a, \texttt{Role}: b$)
is_a($\texttt{Role}: a, \texttt{Role}: b$)

depends($\texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
delegate($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
delegateChain($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
trust($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
trustChain($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
distrust($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
distrustChain($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
monitoring($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Actor}: b, \texttt{Service}: s$)
confident($\texttt{Type}: t, \texttt{Actor}: a, \texttt{Service}: s$)

Specific for execution

requests($\texttt{Actor}: a, \texttt{Service}: s$)
provides($\texttt{Actor}: a, \texttt{Service}: s$)
should_do($\texttt{Actor}: a, \texttt{Service}: s$)
can_satisfy($\texttt{Actor}: a, \texttt{Service}: s$)

Specific for Permission

owns($\texttt{Actor}: a, \texttt{Service}: s$)
has_per($\texttt{Actor}: a, \texttt{Service}: s$)

Goal refinement

subgoal($\texttt{Service}: s_1, \texttt{Service}: s_2$)
OR_subgoal($\texttt{Service}: s_1, \texttt{Service}: s_2$)
AND_subgoal($\texttt{Service}: s_1, \texttt{Service}: s_2$)
AND_decomp($\texttt{Service}: s_1, \texttt{Service}: s_2, \texttt{Service}: s_3$)

$\{L\}$:-$L_1, \ldots, L_n$ to indicate that if $L_1, \ldots, L_n$ are true then L *may* be true. Essentially, L will be added to the model only if some constraints demand its inclusion. This construction can be captured with a simple encoding in logic programs. In Datalog, negation is treated as negation as failure: if there is no evidence that an atom is true, it is considered to be false. Hence if an atom is not true in some model, then its negation should be considered to be true in that model.

We start by presenting the predicates for our framework. We distinguish between two main types of predicates: extensional and intensional. Extensional predicates are predicates set directly with the help of ground facts and are the ones corresponding the edge and circles drawn by the requirements engineer on the CASE tool. Intensional predicates are implicitly determined with the help of rules. Table 2 presents the predicates used to formalize the requirements. For compactness' sake we use the first argument of the predicates to indicate the type of actions. Thus, delegate, delegateChain, distrust, distrustChain, and monitoring have a type $t \in \{exec, perm\}$; trust, trustChain have a type $t \in \{exec, perm, mon\}$; and confident has a type $t \in \{satisfy, exec, owner, mon\}$. Once again, we specify predicates for generic "services" because differentiating them into goals, plans and resources is immediate[5].

The unary predicates goal, plan and resource are used respectively for identifying goals, tasks and resource. Note that type `Goal`, `Task` and `Resource` are sub-types of `Service`. We shall use letters S, G, T and R possibly with indices as metavariables ranging over the terms, respectively, of type `Service`, `Goal`, `Task` and `Resource`. The intuition is that agent(a) holds if instance a is an agent, position(a) holds if instance a is a position, and role(a) holds if instance a is a role. Note that type `Agent`, `Position` and `Role` are sub-types of `Actor`. We shall use letters X, Y and Z as metavariables ranging over the terms of type `Actor`, A, B and C as metavariables ranging over the terms of type `Agent`, and T, Q and V as metavariables ranging over the terms of type `Role`. Metalevel variables are used as a syntactic sugar to avoid to write the predicates that type variables. For example, when the metavariable G occurs in a rule, the predicate goal(G) should be put in the body of the rule. The predicate play(a, b) holds if agent a is an instance of role b. The intuition is that is_a(a, b) holds if role a is a specialization of role b. The predicate depends(a, b, s) holds if actor a depends on actor b for service s. Notice also that when a relation uses variables of type `Actor` the relation can apply to both social and individual levels, but separately.

6.1 Formal Model for Execution

The predicates that we introduced correspond to the relations that the requirements engineer can actually draw during his analysis. The predicate requests(a, s) holds if actor a wants service s fulfilled, while provides(a, s) holds if actor a has the capability to fulfill service s. The predicate delegate$(exec, a, b, s)$ holds if actor a delegates[6] the execution of service s to actor b. Actor a is called *delegater*; actor b is called *delegatee*. The predicate trust$(exec, a, b, s)$ holds is actor a trusts that actor b at least fulfills service s. Actor a is called *truster*; actor b is called *trustee*. The predicate trust(mon, a, b, s) holds if actor a trusts that actor b monitors whether service s will be satisfied. The predicate monitoring$(exec, a, b, s)$ holds if actor a monitors if actor b at least can satisfy service s.

Other predicates are used to define properties that will be used during formal analysis. The predicates delegateChain$(exec, a, b, s)$ and trustChain$(exec, a, b, s)$ hold if there is a delegation and a trust chain respectively, between actor a and actor b. The predicate should_do(a, s) identifies actors who should directly fulfill the service. The basic idea of the predicate can_satisfy is that "for every goal I have assigned responsibilities so that it can be fulfilled". In other words, if an actor has the objective of fulfilling a service, he can satisfy it. Thus it locates the common leaves of the delegation trees of execution and permission. Thus, the predicate can_satisfy(a, s) holds if actor a can satisfy service s. The predicate confident$(satisfy, a, s)$ holds if actor a is confident that

[5] For resources we must replace the subgoal relation with the part-of relation.

[6] For the sake of simplicity we do not deal with the question of depth here. See Li et al. [38] for an account of delegation with depth. What has emerged from several case studies is that depth is less important than qualifications such as "only to members of the same office".

Table 3. Axioms for execution

Delegation	
E1	$\mathsf{delegateChain}(exec, X, Y, S) \leftarrow \mathsf{delegate}(exec, X, Y, S)$
E2	$\mathsf{delegateChain}(exec, X, Z, S) \leftarrow \mathsf{delegate}(exec, X, Y, S) \wedge \mathsf{delegateChain}(exec, Y, Z, S)$
Trust	
E3	$\mathsf{distrustChain}(exec, X, Y, S) \leftarrow \mathsf{distrust}(exec, X, Y, S)$
E4	$\mathsf{distrustChain}(exec, X, Z, S) \leftarrow \left\{ \begin{array}{c} \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{distrust}(exec, Y, Z, S) \wedge \\ \mathsf{not}\ \mathsf{distrustChain}(exec, X, Y, S) \end{array} \right.$
E5	$\mathsf{trustChain}(exec, X, Y, S) \leftarrow \mathsf{trust}(exec, X, Y, S) \wedge \mathsf{not}\ \mathsf{distrustChain}(exec, X, Y, S)$
E6	$\mathsf{trustChain}(exec, X, Z, S) \leftarrow \left\{ \begin{array}{c} \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{trustChain}(exec, Y, Z, S) \wedge \\ \mathsf{not}\ \mathsf{distrustChain}(exec, X, Z, S) \end{array} \right.$
E7	$\mathsf{trustChain}(exec, X, Z, S) \leftarrow \mathsf{trustChain}(mon, X, Y, S) \wedge \mathsf{monitoring}(exec, Y, Z, S)$
E8	$\mathsf{trustChain}(exec, X, Y, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(exec, X, Y, S)$
M1	$\mathsf{trustChain}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Y, S)$
M2	$\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trust}(mon, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
M3	$\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
M4	$\mathsf{trustChain}(mon, X, Y, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(mon, X, Z, S)$
Monitoring	
M5	$\mathsf{monitoring}(exec, Y, Z, S_1) \leftarrow \left\{ \begin{array}{c} \mathsf{delegateChain}(exec, X, Y, S_1) \wedge \\ \mathsf{monitoring}(exec, Z, X, S) \wedge \mathsf{subgoal}(S_1, S) \end{array} \right.$
M6	$\mathsf{confident}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Z, S) \wedge \mathsf{monitoring}(exec, Z, Y, S)$
Should do	
E9	$\mathsf{should\_do}(X, S) \leftarrow \mathsf{delegateChain}(exec, Y, X, S) \wedge \mathsf{provides}(X, S)$
E10	$\mathsf{should\_do}(X, S) \leftarrow \mathsf{requests}(X, S) \wedge \mathsf{provides}(X, S)$
Can satisfy	
E11	$\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{should\_do}(X, S)$
E12	$\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{delegate}(exec, X, B, S) \wedge \mathsf{can\_satisfy}(B, S)$
E13	$\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{can\_satisfy}(X, S_1)$
E14	$\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{can\_satisfy}(X, S_1) \wedge \mathsf{can\_satisfy}(X, S_2)$
Confident to can satisfy	
E15	$\mathsf{confident}(satisfy, X, S) \leftarrow \mathsf{should\_do}(X, S)$
E16	$\mathsf{confident}(satisfy, X, S) \leftarrow \left\{ \begin{array}{c} \mathsf{delegateChain}(exec, X, Y, S) \wedge \\ \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{confident}(satisfy, Y, S) \end{array} \right.$
E17	$\mathsf{confident}(satisfy, X, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{confident}(satisfy, X, S_1)$
E18	$\mathsf{confident}(satisfy, X, S) \leftarrow \left\{ \begin{array}{c} \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{confident}(satisfy, X, S_1) \\ \wedge \mathsf{confident}(satisfy, X, S_2) \end{array} \right.$

service s can be satisfied. Finally, we have the predicates for goal refinement. Their semantics and axiomatization are straight-forward.

The axiomatization is more complex for modelling execution as shown in Table 3. E1 and E2 build a delegation chain of execution. E3-8 define the intensional versions, trustChain and distrustChain of the extensional predicates trust and distrust that are used to build (dis)trust chains by propagating (dis)trust of execution (permission) relations. E5 and E6 (M1 and M2) build a trust chain for execution (monitoring); E5 builds chains over monitoring steps. E8 and M4 have chains propagate to subgoals. According to E8 execution-trust flows top-down with respect to goal refinements. The axiom for monitoring M4 states that trustChain flows top-down with respect to goal refinements. M5 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. M6 introduces the intensional predicate confident(mon, a, b, s): actor a is confident that there exists someone that monitors actor b for service s.

The remaining axioms describe how global properties of the model are defined. E9-10 state that an actor has to execute the service if he provides a service and if either some actor delegates the service to him, or he himself aims for the service. E11-12 state an actor, who requests for a service, can satisfy the service if either he provides it or he has delegated it to someone who can satisfy it. Goal refinements are taken care of by using the axioms E13-14. If an actor can satisfy at least one of the or-subgoals of a service, then he can satisfy the main service. Also, if he can satisfy all and-subgoals, then he can satisfy the main service.

The notion of confidence is captured by axioms E15-E18. An actor is confident that a service will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. Goal refinements are taken care of by using axioms E17-18: if an actor is confident that at least one of the or-subgoals of a service will be fulfilled, then he can be confident that the service will be fulfilled. The axiom for and-decomposition is dual.

6.2 Formal Model for Permission

In Table 2 we also have predicates for modelling permission. The first set of predicates corresponds to the relations drawn by the requirements engineer. The predicate owns(a, s) holds if actor a owns service s. The owner of a service has full authority concerning access and usage of his services, and he can also delegate this authority to other actors. The intuition is that delegate$(perm, a, b, s)$ holds if actor a at most delegates the permission to fulfill service s to actor b. The predicate trust$(perm, a, b, s)$ holds is actor a trusts that actor b at most has the permission to fulfill service s. The predicate monitoring$(perm, a, b, s)$ is the dual of the execution counterpart.

Also in this case other predicates are used to define interesting properties for the formal analysis by the requirement engineer. The predicates delegateChain$(perm, a, b, s)$ and trustChain$(perm, a, b, s)$ hold if there is a delegation, resp. a trust chain of permission among actor a and actor b. The basic idea of has_per sums up the possible ways in which an actor can grab the permission on a service: either directly or by delegation. From the point of view of the owner, confidence means that the owner is confident that the permission that he has delegated will not be misused. Alternatively, the owner is confident that he has delegated permission only to trusted or monitored agents. This means that even if there is one untrusted or unmonitored delegation, then the owner could be uneasy about the likely misuse of his permissions. So, an owner is confident, if there is no likely misuse of his permission. It can be seen that there is an intrinsic double negation in the statement. So we try to model it using a predicate diffident(a, s). At any point of delegation of permission, the delegating agent is diffident, if the delegation is being done to an agent who is neither trusted not monitored or if the delegatee could be diffident himself. In this way, confident$(owner, a, s)$ holds if the owner a is confident to give the permission on service s only to trusted actors.

Table 4 presents the axioms for modelling permission. P1 and P2 build a delegation chain of permission. P3-6 define the intensional versions, trustChain and distrustChain

Table 4. Axioms for permission

Delegation	
P1	$\mathsf{delegateChain}(perm, X, Y, S) \leftarrow \mathsf{delegate}(perm, X, Y, S)$
P2	$\mathsf{delegateChain}(perm, X, Z, S) \leftarrow \mathsf{delegate}(perm, X, Y, S) \wedge \mathsf{delegateChain}(perm, Y, Z, S)$
Trust	
P3	$\mathsf{distrustChain}(perm, X, Y, S) \leftarrow \mathsf{distrust}(perm, X, Y, S)$
P4	$\mathsf{distrustChain}(perm, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{distrust}(perm, Y, Z, S) \wedge \\ \mathsf{not}\ \mathsf{distrustChain}(perm, X, Y, S) \end{cases}$
P5	$\mathsf{trustChain}(perm, X, Y, S) \leftarrow \mathsf{trust}(perm, X, Y, S) \wedge \mathsf{not}\ \mathsf{distrustChain}(perm, A, B, S)$
P6	$\mathsf{trustChain}(perm, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{trustChain}(perm, Y, Z, S) \wedge \\ \mathsf{not}\ \mathsf{distrustChain}(perm, X, Z, S) \end{cases}$
P7	$\mathsf{trustChain}(perm, X, Z, S) \leftarrow \mathsf{trustChain}(mon, X, Y, S) \wedge \mathsf{monitoring}(perm, Y, Z, S)$
P8	$\mathsf{trustChain}(perm, X, Y, S) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(perm, X, Y, S_1)$
M7	$\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
Monitoring	
M8	$\mathsf{monitoring}(perm, Z, Y, S_1) \leftarrow \begin{cases} \mathsf{delegateChain}(perm, X, Y, S_1) \wedge \\ \mathsf{monitoring}(perm, Z, X, S) \wedge \mathsf{subgoal}(S_1, S) \end{cases}$
M9	$\mathsf{confident}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Z, S) \wedge \mathsf{monitoring}(perm, Z, Y, S)$
Has permission	
P9	$\mathsf{has\_per}(X, S) \leftarrow \mathsf{owns}(X, S)$
P10	$\mathsf{has\_per}(X, S) \leftarrow \mathsf{delegateChain}(perm, Y, X, S) \wedge \mathsf{has\_per}(Y, S)$
P11	$\mathsf{has\_per}(X, S_1) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{has\_per}(X, S)$
Owner is confident to give the service to trusted actors	
P12	$\mathsf{confident}(owner, X, S) \leftarrow \mathsf{owns}(X, S) \wedge \mathsf{not}\ \mathsf{diffident}(X, S)$
P13	$\mathsf{diffident}(X, S) \leftarrow \mathsf{delegateChain}(perm, X, Y, S) \wedge \mathsf{diffident}(Y, S)$
P14	$\mathsf{diffident}(X, S) \leftarrow \mathsf{delegateChain}(perm, X, Y, S) \wedge \mathsf{not}\ \mathsf{trustChain}(perm, X, Y, S)$
P15	$\mathsf{diffident}(X, S) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{diffident}(X, S_1)$

of the extensional predicates trust and distrust that are used to build (dis)trust chains by propagating (dis)trust of permission relations.

P5 and P6 build a trust chain for permission; P7 builds chains over monitoring steps. P8 has the chain propagate through subgoals. If an actor trusts that another will not overstep the set of actions required to fulfill a subgoal of a service, then the first can trust the last not to overstep the set of actions required to fulfill the service. The permission trust, with respect to goal refinements, flows bottom-up. M7 is used to build a trust chain for monitor. M8 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. M9 is the permission counterpart of M6. The owner of a service has full authority concerning access and disposition of it. Thus, P9 states that if an actor owns a service, he has permission on it. P10 states that the delegatee has permission on the service. P11 propagates permission through subgoals. The notion of confidence and diffidence that we have sketched above is captured by the axioms P12-P16.

6.3 Combining Execution and Permission

More sophisticated properties require reasoning with both execution and permission. To this end, we introduce some notions that put together these two notions. In Table 5

Table 5. Axioms Involving both permission and execution

Can see the service fulfilled (can execute)	
Ax1	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{should\_do}(X,S) \wedge \mathsf{has\_per}(X,S)$
Ax2	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{delegateChain}(exec,X,Y,S) \wedge \mathsf{can\_execute}(Y,S)$
Ax3	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{OR\_subgoal}(S_1,S) \wedge \mathsf{can\_execute}(X,S_1)$
Ax4	$\mathsf{can\_execute}(X,S) \leftarrow \begin{cases} \mathsf{AND\_decomp}(S,S_1,S_2) \wedge \mathsf{can\_execute}(X,S_1) \\ \wedge\ \mathsf{can\_execute}(X,S_2) \end{cases}$
Confident to see the service fulfilled (confident to execute)	
Ax5	$\mathsf{confident}(exec,X,S) \leftarrow \mathsf{should\_do}(X,S) \wedge \mathsf{has\_per}(X,S)$
Ax6	$\mathsf{confident}(exec,X,S) \leftarrow \begin{cases} \mathsf{delegateChain}(exec,X,Y,S) \wedge \\ \mathsf{trustChain}(exec,X,Y,S) \wedge \mathsf{confident}(exec,Y,S) \end{cases}$
Ax7	$\mathsf{confident}(exec,X,S) \leftarrow \mathsf{OR\_subgoal}(S_1,S) \wedge \mathsf{confident}(exec,X,S_1)$
Ax8	$\mathsf{confident}(exec,X,S) \leftarrow \begin{cases} \mathsf{AND\_decomp}(S,S_1,S_2) \wedge \mathsf{confident}(exec,X,S_1) \\ \wedge\ \mathsf{confident}(exec,X,S_2) \end{cases}$
Need to know	
Ax9	$\mathsf{need\_to\_have\_perm}(X,S) \leftarrow \mathsf{should\_do}(X,S)$
Ax10	$\mathsf{need\_to\_have\_perm}(X,S) \leftarrow \begin{cases} \mathsf{delegate}(perm,X,Y,S) \wedge \mathsf{need\_to\_have\_perm}(Y,S) \\ \wedge\ \mathbf{not}\ \mathsf{other\_delegater}(X,Y,S) \end{cases}$
Ax11	$\mathsf{other\_delegater}(X,Y,S) \leftarrow \begin{cases} \mathsf{delegate}(perm,Z,Y,S) \wedge \\ \mathsf{need\_to\_have\_perm}(Y,S) \wedge X \neq Z \end{cases}$

we present the notions from both the point of view of the requester and the point of view of the owner. The predicate $\mathsf{can\_execute}(a,s)$ holds if actor a can see service s fulfilled. The predicate $\mathsf{confident}(exec,a,s)$ holds if actor a is confident to see service s fulfilled. Actor a, who aims for service s, is confident that s will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. This is done using the axioms Ax5-6. Goal refinements are taken care of by using the axioms Ax7-8. If a is confident that at least one of the or-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled. Also, if a is confident that all and-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled.

Owners may wish to delegate permissions to providers only if the latter actually do need the permission. The last part of Table 5 defines the predicates that are necessary to analyze *need-to-know* properties. As a result of absence of diffidence, the owner can be confident that his permission will not be misused. But has this permission reached the agents who actually *need* it? The owner might also want to ensure that there has been not unwanted delegation of permission. This can be achieved by identifying the agents who actually *need-to-know* (or rather *need-to-have*) the permission. This set of axioms captures also the possibility of having alternate paths of permission delegations. In this case the formal analysis will not yield one model but multiple models in which only one path of delegation is labeled by the need-to-have property and the others are not.

Example 14 (Figure 9). Alice and Carol (7 and 8) have both received the consent (permission) by Bob (1) for using his personal data, and both delegate it to the faculty secretariat (3), which must have the permission to provide the data to Paul (6), the university tutor who should provide personal counseling to Bob. In this case only one of either Alice or Carol needs to have the permission.

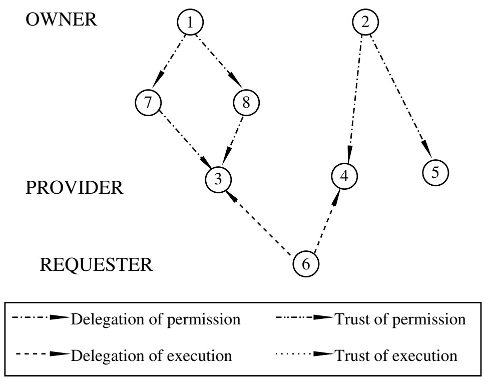


Fig. 9. Need-to-Know and Multiple Permissions Paths

6.4 Other Features

In Table 6 there are the axioms to map Tropos dependency into Secure Tropos framework and vice versa. Notice that ST1-2 and ST5 have also to be repeated for the case where the dependum is a plan.

Table 6. Axioms for mapping Tropos into Secure Tropos and vive versa

From Tropos to Secure Tropos
ST1 $\mathsf{trust}(exec, X, Y, G) \leftarrow \mathsf{depends}(X, Y, G)$ ST2 $\mathsf{delegate}(exec, X, Y, G) \leftarrow \mathsf{depends}(X, Y, G)$ ST3 $\mathsf{trust}(perm, Y, X, R) \leftarrow \mathsf{depends}(X, Y, R)$ ST4 $\mathsf{delegate}(perm, Y, X, R) \leftarrow \mathsf{depends}(X, Y, R)$
From Secure Tropos to Tropos
ST5 $\mathsf{depends}(X, Y, G) \leftarrow \begin{cases} \mathsf{trust}(exec, X, Y, G) \land \mathsf{delegate}(exec, X, Y, G) \land \\ \quad \textbf{not } \mathsf{distrust}(exec, X, Y, G) \end{cases}$ ST6 $\mathsf{depends}(X, Y, R) \leftarrow \begin{cases} \mathsf{trust}(perm, Y, X, R) \land \mathsf{delegate}(perm, Y, X, R) \land \\ \quad \textbf{not } \mathsf{distrust}(perm, Y, X, R) \end{cases}$

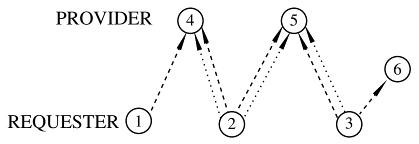
Table 7 presents the axioms for role hierarchy and for mapping relations from social level to individual level. The predicate specialize is the intensional version of is_a, whereas instance is intensional version of play. Axioms SI1-13 have to be repeated replacing the predicate instance with specialize and predicate agent with role for completing social level with respect to role hierarchy.

6.5 Analysis and Verification

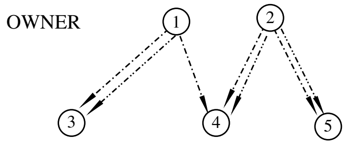
Design *properties* are not enforced with axioms for two reasons. At first the actual system drawn by the requirement engineer may not satisfy them, and therefore the missing link may be actually a bug. Second, there might be many ways in which a require-

Table 7. Axioms for role hierarchy and for mapping social level into individual level

Role Hierarchy	
RH1	$\mathsf{specialize}(T,Q) \leftarrow \mathsf{is\_a}(T,Q)$
RH2	$\mathsf{specialize}(T,Q) \leftarrow \mathsf{specialize}(T,V) \wedge \mathsf{is\_a}(V,Q)$
RH3	$\mathsf{instance}(A,T) \leftarrow \mathsf{play}(A,T)$
RH4	$\mathsf{instance}(A,T) \leftarrow \mathsf{instance}(A,Q) \wedge \mathsf{specialize}(Q,T)$
From social level to individual level	
SI1	$\mathsf{provides}(A,S) \leftarrow \mathsf{provides}(T,S) \wedge \mathsf{instance}(A,T)$
SI2	$\mathsf{requests}(A,S) \leftarrow \mathsf{requests}(T,S) \wedge \mathsf{instance}(A,T)$
SI3	$\mathsf{owns}(A,S) \leftarrow \mathsf{owns}(T,S) \wedge \mathsf{instance}(A,T)$
SI4	$\mathsf{trust}(exec,A,B,S) \leftarrow \mathsf{trust}(exec,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI5	$\mathsf{trust}(perm,A,B,S) \leftarrow \mathsf{trust}(perm,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI6	$\mathsf{distrust}(exec,A,B,S) \leftarrow \mathsf{distrust}(exec,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI7	$\mathsf{distrust}(perm,A,B,S) \leftarrow \mathsf{distrust}(perm,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI8	$\mathsf{delegate}(exec,A,B,S) \leftarrow \mathsf{delegate}(exec,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI9	$\mathsf{delegate}(perm,A,B,S) \leftarrow \mathsf{delegate}(perm,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI10	$\mathsf{monitoring}(exec,A,B,S) \leftarrow \mathsf{monitoring}(exec,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI11	$\mathsf{monitoring}(perm,A,B,S) \leftarrow \mathsf{monitoring}(perm,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI12	$\mathsf{trust}(mon,A,B,S) \leftarrow \mathsf{trust}(mon,T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$
SI13	$\mathsf{depends}(A,B,S) \leftarrow \mathsf{depends}(T,Q,S) \wedge \mathsf{instance}(A,T) \wedge \mathsf{instance}(B,Q)$



(a) From requesters' viewpoint



(b) From owners' viewpoint

Fig. 10. Design for delegation of execution and permission

ment engineer may wish to fulfill desired properties. We use the DLV system[7] to verify security properties with respect to a Secure Tropos model.

In Table 8 we use the $A \Rightarrow ?\ B$ to mean that one must check that each time A holds it is desirable that B also holds. In Datalog this can be represented as the constraint :- A, not B. If the set of features is not consistent, i.e., they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. This also guarantee us that our proposed axioms are consistent if we check for consistency of the model without trying to enforce any property.

Pro1 states that if there is a delegation chain either the delegater trusts the delegatee or there is the monitor and the delegater trust the monitor. Pro2 states that a requester wants to can satisfy his goals, and Pro3 states that a requester wants to be confident to satisfy the service.

[7] http://www.dbai.tuwien.ac.at/proj/dlv

Table 8. Desirable Properties of a Design

Execution	
Pro1	$\mathsf{delegateChain}(exec, X, Y, S) \Rightarrow ?\ \mathsf{trustChain}(exec, X, Y, S)$
Pro2	$\mathsf{requests}(X, S) \Rightarrow ?\mathsf{can\_satisfy}(X, S)$
Pro3	$\mathsf{requests}(X, S) \Rightarrow ?\mathsf{confident}(satisfy, X, S)$
Pro4	$\mathsf{should\_do}(X, S) \Rightarrow ?\mathsf{not}\ \mathsf{delegateChain}(exec, X, Y, S)$
Permission	
Pro5	$\mathsf{delegateChain}(perm, X, Y, S) \Rightarrow ?\ \mathsf{trustChain}(perm, X, Y, S)$
Pro6	$\mathsf{owns}(X, S) \Rightarrow ?\ \mathsf{confident}(owner, X, S)$
Pro7	$\mathsf{owns}(X, S) \Rightarrow ?\ \mathsf{not}\ \mathsf{delegateChain}(perm, Y, X, S) \wedge X \neq Y$
Execution & Permission	
Pro8	$\mathsf{requests}(X, S) \Rightarrow ?\mathsf{can\_execute}(X, S)$
Pro9	$\mathsf{requests}(X, S) \Rightarrow ?\mathsf{confident}(exec, X, S)$
Pro10	$\mathsf{owns}(X, S) \Rightarrow ?\mathsf{need\_to\_have\_perm}(X, S)$
Pro11	$\mathsf{owns}(X, S) \Rightarrow ?\mathsf{need\_to\_have\_perm}(X, S) \wedge \mathsf{confident}(owner, X, S)$

Example 15 (Figure 10(a)). Bob and Bert (1 and 2) need counseling. They can receive it (formal relation can satisfy) because they delegate the execution to Paul and Peter (4 and 5), while Bill (3) cannot receives all necessary advices because he requested some of them only to Alice (6) which is not able to provide counseling on faculty matters.

Bob is also confident to receive all counseling he needs since he delegates the execution to Paul and Peter (4 and 5) whom he trusts, while Bert is not confident since he delegates to Paul (4) that he does not trust.

Pro4 states that if an actor provides a service, then, if either some actor delegates the service to him, or if he himself requests the service, then he has to execute the service without further delegation. Pro5 states that if there is a delegation chain, either the delegater trusts the delegatee or there is the monitor. Pro6 states that the owner of the service has to be confident to give the service to trusted actors, and Pro7 states that a service cannot come back to the owner.

Example 16 (Figure 10(b)). Bob and Bert (1 and 2) need to provide their personal data for receiving accurate counseling. Bob is confident on his personal data since he delegates the permission on it to two Paul and Peter (4 and 5) who he trusts to use the data at most for counseling. On the other hand, Bert is not confident on her data since she delegates it to Paul (4) whom she does not trust to keep her information confidential.

This example is very close to the example that we have previously seen on misplaced delegation (Example 15). What changes is what can be obtained by poor Bert. In the former case he is afraid to receive a bad advice (delegation of execution), in the latter that her information can be used for other things than providing counseling.

The last part of Table 8 shows properties to verify at-most model and at-least model at the same time. Pro8 states that the requester has to can see the service fulfilled. Pro9 states that the requester has to be confident to see the service fulfilled.

Table 9 presents the properties used to identifying conflicts that occur when both a trust and a distrust relations exist among two actors for the same service. Pro1-2 are used to identify generic conflicts and correspond to Definition 1 and 2. These properties apply

Table 9. Properties for identifying conflicts

TC1 $\mathsf{trustChain}(exec, X, Y, S) \Rightarrow ?\mathsf{not}\ \mathsf{distrustChain}(exec, X, Y, S)$	
TC2 $\mathsf{trustChain}(perm, X, Y, S) \Rightarrow ?\mathsf{not}\ \mathsf{distrustChain}(perm, X, Y, S)$	
TC3 $\mathsf{trustChain}(exec, A, B, S) \Rightarrow ?$	$\begin{cases} \mathsf{not}\ \mathsf{distrustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC4 $\mathsf{trustChain}(perm, A, B, S) \Rightarrow ?$	$\begin{cases} \mathsf{not}\ \mathsf{distrustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC5 $\mathsf{distrustChain}(exec, A, B, S) \Rightarrow ?$	$\begin{cases} \mathsf{not}\ \mathsf{trustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC6 $\mathsf{distrustChain}(perm, A, B, S) \Rightarrow ?$	$\begin{cases} \mathsf{not}\ \mathsf{trustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$

Table 10. Axioms for solving conflicts

C1 $\{\mathsf{monitoring}(exec, M, B, S)\} \leftarrow$	$\begin{cases} \mathsf{distrustChain}(exec, A, B, S) \wedge \\ \mathsf{trustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \wedge \\ \mathsf{trustChain}(mon, A, M, S) \end{cases}$
C2 $\{\mathsf{monitoring}(perm, M, B, S)\} \leftarrow$	$\begin{cases} \mathsf{distrustChain}(perm, A, B, S) \wedge \\ \mathsf{trustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \wedge \\ \mathsf{trustChain}(mon, A, M, S) \end{cases}$

Table 11. Axioms in order to support monitoring

E4′ $\mathsf{distrust}(exec, A, B, S) \leftarrow$	$\begin{cases} \mathsf{distrust}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \\ \mathsf{instance}(B, Q) \wedge \textbf{not}\ \mathsf{confident}(mon, A, B, S) \end{cases}$
P4′ $\mathsf{distrust}(perm, A, B, S) \leftarrow$	$\begin{cases} \mathsf{distrust}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \\ \mathsf{instance}(B, Q) \wedge \textbf{not}\ \mathsf{confident}(mon, A, B, S) \end{cases}$

to both social level and individual level, independently and so A and B have to be typed as role for the social level and as agents for the individual level. Pro1-2 can be refined in order to identify conflicts of the form of Fig. 8(c) (Pro3-4) and Fig. 8(b) (Pro5-6).

Table 10 formalizes the proposal for solving conflicts when there is a trust relation at social level and a distrust relation at individual level. In order to accommodate C1-2 in our framework we have to modify axioms Ax6-7 in Table 7. The new version of these axioms is given in Table 11.

7 Computer Aided SRE

ST-Tool [24,29] is a CASE tool for design and verification of functional and security requirements, and has been designed to support the Secure Tropos methodology. It provides a user interface for drawing Secure Tropos models, support for translating automatically graphical models into formal specifications and a front-end with external tools for model checking.

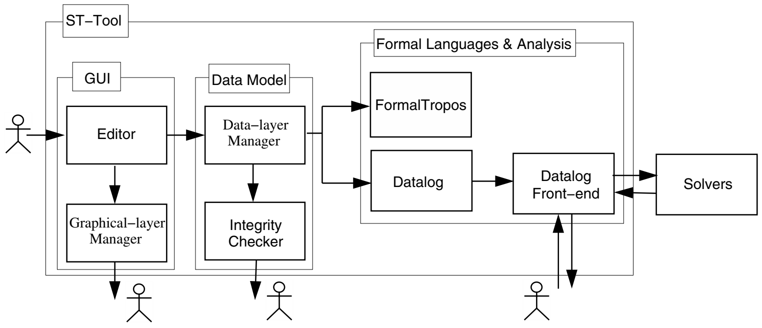
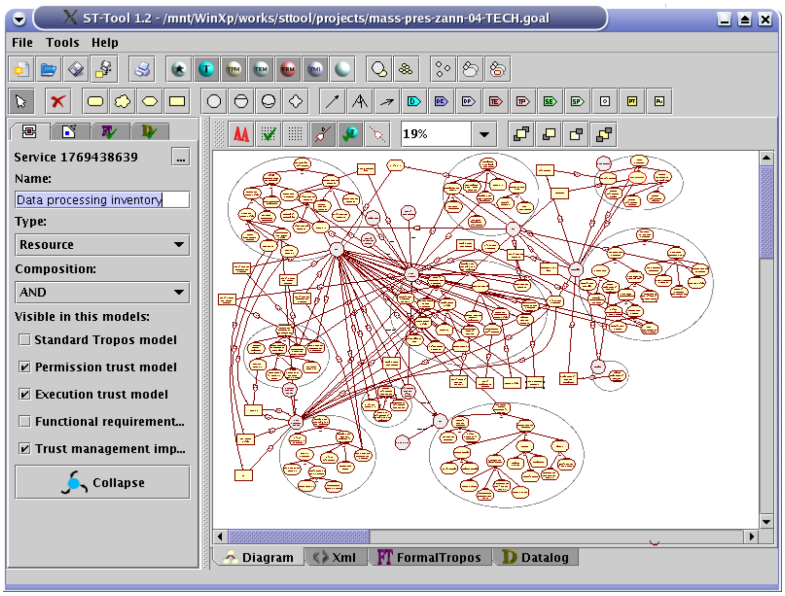


Fig. 11. The Architecture Overview


ST-Tool 1.2 - /mnt/WinXp/works/sttool/projects/mass-pres-zann-04-TECH.goal
File Tools Help
19%
Diagram
Xml
FormalTropos
Datalog
Collapse

Service 1769438639

Name:

Data processing inventory

Type:

Resource

Composition:

AND

Visible in this models:

- [ ] Standard Tropos model
- [x] Permission trust model
- [x] Execution trust model
- [ ] Functional requirement...
- [x] Trust management imp...

Fig. 12. ST-Tool screenshot

ST-Tool is mainly composed of two parts: the ST-Tool kernel and external solvers. ST-Tool kernel has an architecture comprised of three major parts, each of which is comprised of modules. Next, we will discuss these modules and their interconnections. In Fig. 11, the modules of ST-Tool are shown, their interrelations are also indicated.

The tool provides a graphical user interface (GUI), through which system designers can manage all the components and functionalities of the tool. A screenshot of the interface is shown in Fig. 12. To manage visual editing features and data management consistency at the same time, we have adopted a two-layer solution: a graphical layer and a data layer. In graphical layer, models are shown as graphs where actors and services are nodes, and relations are arcs. Each visual object refers to a data object. The collection of data objects is the data layer. The GUI's key component is the *Editor Mod-*

ule. This module allows the user to visually insert, edit or remove graphical objects in the graphical layer and object properties in the data layer. A second GUI component is the *Graphical-layer Manager (GM) Module* that manages graphical objects and their visualization. It supports goal refinement by associating a goal diagram with each actor and then allows to collapse actors and services in order to maintain readable diagrams. Further, GM permits to display one or more views of a diagram at the same time, namely dependency model (aka Tropos model), delegation models, and trust models.

The *Data-layer Manager (DM) Module* is responsible for building and maintaining data corresponding to graphical objects. For example, DM manages misalignments between social relations and their graphical representation. Actually, GM uses arcs to connect two nodes to each other, while many Secure Tropos relations are ternary. DM rebuilds these relations by linking two appropriate graphical objects (the two arcs) to the same data object (the relation). ST-Tool allows users to save models through the DM module that stores a neutral description of the entire model in `.xml` format files. A support for detecting errors and warnings during the design phase is provided by the *Integrity Checker Module*. This module analyzes models stored in the DM module and reports errors such as "orphan relations" (i.e. relations where an arc is missing) and "isolated nodes" (i.e. services not involved in any relations). Warnings are different from errors: they are failure of integrity constraints, like errors, but the designer may be perfectly happy with a design that does not satisfy them. Integrity Checker reports warnings, for example, when more than one service have the same name.[8]

After drawing so many nice diagrams, system designers may want to check whether the models derived so far satisfy some general desirable properties. To support formal analysis, ST-Tool allows automatic transformations from the `.xml` file stored by DM into Formal Tropos [20] and Datalog specifications. These transformations are performed, respectively, by two different modules: *Formal Tropos Module* and *Datalog Module*. The resulting specifications are displayed by selecting the corresponding panel.

The process for completing and checking models is controlled by the *Datalog Frontend (DF) Module*. Through this module, requirement engineers can choose the axioms to complete the model and the properties to be verified on it. Properties are grouped into Authorization, Availability, Integrity and Need-to-know categories, so that engineers only need to specify the categories they wants to verify to include the corresponding rule set. Once designers are confident with the model, the resulting Datalog specification is given in input to some external solvers that verify the consistency of the model corresponding to the specifications. Then, the solver output is parsed by the DF module in order to present in a more user-readable format. A scheme of the entire process for modelling and analyzing security requirements is given in Fig. 13.

We use different ASP solvers for the requirements analysis, namely ASSAT,[9] Cmodels,[10] Smodels,[11] and DLV.[12] ASSAT, Cmodels, and Smodels work with grounded logic programs generated by Lparse [54]. In particular, Cmodels and ASSAT use SAT solvers

[8] More than one service with the same name are needed to represent delegation and trust chains.
[9] http://assat.cs.ust.hk/
[10] http://www.cs.utexas.edu/users/tag/cmodels.html
[11] http://www.tcs.hut.fi/Software/smodels/
[12] http://www.dbai.tuwien.ac.at/proj/dlv/

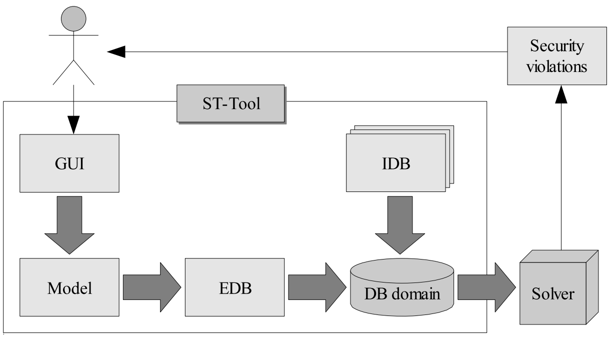


Fig. 13. ST-Tool: the analysis cycle

Table 12. Experimental Result

Solver	cmodels-1			cmodels-2			smodels			assat			dlv		
N. Ins.	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU
0	0	0m13.32s	0m0.25s	0	0m13.53s	0m0.13s	0	0m14.83s	0m0.13s	0	0m14.82s	0m0.13s	0	0m0.12s	0m0.01s
24	0	0m59.08s	0m0.61s	0	0m58.99s	0m0.57s	0	1m5.15s	0m0.56s	0	1m4.92s	0m0.59s	0	0m0.31s	0m0.00s
45	0	2m33.69s	0m2.06s	0	2m33.77s	0m1.73s	0	2m50.51s	0m1.68s	0	2m50.18s	0m1.75s	0	0m0.67s	0m0.02s
62	1	0m41.19s	0m1.80s	1	0m41.04s	0m1.74s	1	0m46.28s	0m1.66s	1	0m46.72s	0m1.60s	0	0m0.95s	0m0.01s
113	1	0m47.94s	0m1.72s	1	0m47.70s	0m1.76s	1	0m54.34s	0m1.63s	1	0m54.27s	0m1.71s	0	0m2.42s	0m0.02s
166	1	0m27.73s	0m1.58s	1	0m27.77s	0m1.55s	1	0m32.71s	0m1.75s	1	0m33.74s	0m1.86s	0	0m5.05s	0m0.08s

as research engine for determining the solution, while Smodels uses general-purpose answer set solvers. Finally, DLV is developed as a deductive database system.

In order to compare the different solvers, we have tested them on a pool of benchmarks based on a comprehensive case study on the compliance to the Italian security and privacy legislation of public administrations such as universities, local governments and health care authorities [43]. Benchmarks are defined from the structure of the organization (base case) by adding a growing number of agents (instances) playing the roles occurring in the model.

The benchmarks evaluation results of the experiments carried out are reported in Table 12. The experiments were executed on a bi-processor XEON, 3.2 GHz, 1 MB of Chache, 4GB of RAM, running Linux. For each problem we report the time used to complete the analysis (Wall) and by CPU. However, Wall and CPU reported in Table 12 do not take into account the time spent by Lparse that Cmodels, Smodels and Assat use for grounding. Further, with “0” we mark the experiments that complete successfully, while with “1” we mark those experiments that fail for some reason such as memory limits exceeded. The experiments show that DLV system is more efficient than the other solvers. Further, Cmodels, Smodels and ASSAT are not able to find a solution after a certain number of instances since Lparse exceeds memory limit.

8 Conclusions

Security Requirements Engineering is one of the challenging field for computer security research. Here we have sketched the overall methodological issues that underpins the design of a novel methodology for security design.

Looking back at our proposed classification, this work is well placed within the meta-level modelling field. To avoid some of the disadvantages of the approach we have focused on a modular addition so that dropping all newly proposed features makes us return to Tropos/i* original methodology.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic Databases. In *Proc. of VLDB'02*, pages 143–154. Morgan Kaufmann, 2002.
3. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. An Implementation of P3P Using Database Technology. In *Proc. of EDBT'04*, *LNCS 2992*, pages 845–847. Springer-Verlag, 2004.
4. R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.
5. A. I. Antòn and J. B. Earp. A requirements taxonomy for reducing Web site privacy vulnerabilities. *Requirements Eng.*, 9(3):169–185, 2004.
6. A. I. Antòn, J. B. Earp, and A. Reese. Analyzing Website privacy requirements using a privacy goal taxonomy. In *Proc. of RE'02*, pages 23–31. IEEE Press, 2002.
7. T. Aura. On the Structure of Delegation Networks. In *Proc. of 1998 CSFW*, pages 14–26. IEEE Press, 1998.
8. M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *Proc. of SAC'04*, 2004.
9. M. Backes, B. Pfitzmann, and M. Schunter. A Toolkit for Managing Enterprise Privacy Policies. In *Proc. of ESORICS'03*, *LNCS 2808*, pages 162–180. Springer-Verlag, 2003.
10. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.
11. C. Castelfranchi and R. Falcone. Principles of trust for MAS: Cognitive anatomy, social importance and quantification. In *Proc. of ICMAS'98*, pages 72–79. IEEE Press, 1998.
12. L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
13. L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation, Apr. 2002.
14. R. Crook, D. Ince, L. Lin, and B. Nuseibeh. Security Requirements Engineering: When Anti-requirements Hit the Fan. In *Proc. of RE'02*, pages 203–205. IEEE Press, 2002.
15. J. DeTreville. Binder, a logic-based security language. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 95–103. IEEE Press, 2002.
16. P. T. Devanbu and S. G. Stubblebine. Software engineering for security: a roadmap. In *Proc. of ICSE'00*, pages 227–239, 2000.
17. T. Doan, S. Demurjian, T. C. Ting, and A. Ketterl. MAC and UML for secure software design. In *Proc. of FMSE'04*, pages 75–85. ACM Press, 2004.
18. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *TISSEC*, 4(3):224–274, 2001.
19. R. Fredriksen, M. Kristiansenand, B. A. G. K. Stølen, T. A. Opperud, and T. Dimitrakos. The CORAS framework for a model-based risk management process. In *Proc. of SAFECOMP'02*, *LNCS 2434*, pages 94–105, 2002.
20. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements: Some experimental results. In *Proc. of RE'03*. IEEE Press, 2003.
21. G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the Impact of Trust and Distrust in Agent Networks. In *Proc. of AOIS'01*, pages 45–58, 2001.

22. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th Int. Conf. on Log. Prog.*, pages 1070–1080. MIT Press, 1988.
23. P. Giorgini, F. Massacci, and J. Mylopoulos. Requirement Engineering meets Security: A Case Study on Modelling Secure Electronic Transactions by VISA and Mastercard. In *Proc. of ER'03*, *LNCS 2813*, pages 263–276. Springer-Verlag, 2003.
24. P. Giorgini, F. Massacci, J. Mylopoulos, A. Siena, and N. Zannone. ST-Tool: A CASE Tool for Modeling and Analyzing Trust Requirements. In *Proc. of iTrust'05*, *LNCS 3477*, pages 415–419. Springer-Verlag, 2005.
25. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Filling the gap between Requirements Engineering and Public Key/Trust Management Infrastructures. In *Proc. of EuroPKI'04*, *LNCS 3093*, pages 98–111. Springer-Verlag, 2004.
26. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning. In *Proc. of iTrust'04*, *LNCS 2995*, pages 176–190. Springer-Verlag, 2004.
27. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modeling Security Requirements Through Ownership, Permission and Delegation. In *Proc. of RE'05*, 2005. To appear.
28. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modelling Social and Individual Trust in Requirements Engineering Methodologies. In *Proc. of iTrust'05*, *LNCS 3477*, pages 161–176. Springer-Verlag, 2005.
29. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. ST-Tool: A CASE Tool for Security Requirements Engineering. In *Proc. of RE'05*, 2005. To appear.
30. Z. Guessoum, M. Ziane, and N. Faci. Monitoring and Organizational-Level Adaptation of Multi-Agent Systems. In *Proc. of AAMAS'04*, pages 514–521. ACM Press, 2004.
31. M. Hannoun, J. S. Sichman, O. Boissier, and C. Sayettat. Dependence Relations between Roles in a Multi-Agent System: Towards the Detection of Inconsistencies in Organization. In *Proc. of MABS'98*, *LNCS 1534*, pages 169–182. Springer-Verlag, 1998.
32. Q. He and A. I. Antón. A Framework for Modeling Privacy Requirements in Role Engineering. In *Proc. of the 9th Int. Workshop on Requirements Eng. : Found. for Software Quality*, pages 137–146, 2003.
33. T. Jaeger and A. Prakash. Requirements of role-based access control for collaborative systems. In *Proc. of 1st ACM Workshop on Role-Based Access Control*, pages 53–64. ACM Press, 1995.
34. A. J. I. Jones and M. J. Sergot. A Formal Characterisation of Institutionalised Power. *J. of the Interest Group in Pure and Appl. Log.*, 4(3):429–445, 1996.
35. J. Jürjens. *Secure Systems Development with UML.* Springer-Verlag, 2004.
36. G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring Teams by Overhearing: A Multi-Agent Plan-Recognition Approach. *JAIR*, 17:83–135, 2002.
37. G. Karjoth, M. Schunter, and M. Waidner. Platform for Enterprise Privacy Practices: Privacy-enabled Management of Customer Data. In *Proc. of PET'02*. Springer-Verlag, 2002.
38. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *TISSEC*, 6(1):128–171, 2003.
39. N. Li, J. C. Mitchell, and W. H. Winsborough. Design of A Role-based Trust-management Framework. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 114–130. IEEE Press, 2002.
40. L.-C. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Analysing Security Threats and Vulnerabilities Using Abuse Frames. Technical Report 2003/10, The Open University, 2003.
41. L. Liu, E. S. K. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of RE'03*, pages 151–161. IEEE Press, 2003.
42. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. of UML'02*, *LNCS 2460*, pages 426–441. Springer-Verlag, 2002.

43. F. Massacci, M. Prest, and N. Zannone. Using a Security Requirements Engineering Methodology in Practice: The compliance with the Italian Data Protection Legislation. *Comp. Standards & Interfaces*, 27(5):445–455, 2005. An extended version is available as Technical report DIT-04-103 at `eprints.biblio.unitn.it`.
44. J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proc. of ACSAC'99*, pages 55–66. IEEE Press, 1999.
45. H. Mouratidis, P. Giorgini, and G. Manson. Modelling secure multiagent systems. In *Proc. of AAMAS'03*, pages 859–866. ACM Press, 2003.
46. H. Nwana. Software agents: An overview. *Knowledge Engineering Review J.* , 11(3), 1996.
47. S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *TISSEC*, 3(2):85–106, 2000.
48. L. Ponemon. What Keeps Security Professionals Up At Night?, April 2003. URL: http://www.darwinmag.com/read/040103/threats.html.
49. I. Ray, N. Li, R. France, and D.-K. Kim. Using UML to visualize role-based access control constraints. In *Proc. of SACMAT'04*, pages 115–124. ACM Press, 2004.
50. P. Samarati and S. D. C. di Vimercati. Access Control: Policies, Models, and Mechanisms. In *FOSAD 2001/2002*, *LNCS 2946*, pages 137–196. Springer-Verlag, 2001.
51. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Comp.*, 29(2):38–47, 1996.
52. G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Eng.*, 10(1):34–44, 2005.
53. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.
54. T. Syrjänen. *Lparse 1.0: User's Manual*. Helsinki University of Technology, 2000.
55. A. Toval, A. Olmos, and M. Piattini. Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In *Proc. of RE'02*, pages 95 –103. IEEE Press, 2002.
56. T. Tryfonas, E. Kiountouzis, and A. Poulymenakou. Embedding security practices in contemporary information systems development approaches. *Inform. Management and Comp. Sec.*, 9:183–197, 2001.
57. A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of PODS'89*, pages 1–10. ACM Press, 1989.
58. A. van Lamsweerde, S. Brohez, R. De Landtsheer, and D. Janssens. From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering. In *Proc. of RHAS'03*, pages 49–56, 2003.
59. A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *TSE*, 26(10):978–1005, 2000.
60. J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.
61. E. S. K. Yu. Agent-Oriented Modelling: Software versus the World. In *Proc. of AOSE'01*, *LNCS 2222*, pages 206–225. Springer-Verlag, 2001.
62. P. Zave. Classification of research efforts in requirements engineering. *CSUR*, 29(4):315–321, 1997.

Author Index