Implementation-Level Analysis of Cryptographic Protocols and their Applications to E-Voting Systems

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Enrico Scapin

aus Verona, Italien

Hauptberichter: Prof. Dr. Ralf Küsters
Mitberichter: Prof. Dr. Bernhard Beckert

Tag der mündlichen Prüfung: 11. Mai 2018

Institut für Informationssicherheit der Universität Stuttgart

2018

Abstract

Formal verification of security properties of both cryptographic operations, such as encryption, and protocols based on them, such as TLS, has been the goal of a substantial research effort in the last three decades. One fundamental limitation in the verification of these security properties is that analyses are typically carried out at the design level and hence they do not provide reliable guarantees on the implementations of these operations/protocols. To overcome this limitation, in this thesis we aim at establishing formally justified cryptographic guarantees directly at the implementation level for systems that are coded in Java and use cryptography. Our approach is based on a general framework for the cryptographic verification of Java programs (the CVJ framework) which formally links cryptographic indistinguishability properties and noninterference properties. In this way, it enables existing tools that can check standard noninterference properties, but a priori cannot deal with cryptography, to also establish cryptographic privacy properties for Java systems. The CVJ framework is stated and proven for a Java-like formal language which however does not cover all the data types and features commonly used in Java programs. Moreover, the framework originally supports only one cryptographic operation, namely public-key encryption.

The first contribution of this thesis is hence to extend and to instantiate the CVJ framework in order to make it more widely applicable. We extend the underlying formal language with some features of Java which have not been captured yet, such as concurrency, and we restate and prove all the results of the framework to carry them over into this extended language. We then further instantiate the framework with additional cryptographic operations: digital signatures and public-key encryption, both now also including a public-key infrastructure, (private) symmetric encryption, and nonce generation. The methods and techniques developed within the CVJ framework are relevant and applicable independently of any specific tool employed. However, to illustrate the usefulness of this approach, we apply the framework along with two verification tools for Java programs, namely the fully automated static checker Joana and the interactive theorem prover KeY, to establish strong cryptographic privacy properties for systems which use cryptography, such as client-server applications and e-voting systems.

In this context, the second major contribution of this thesis is the design, the implementation, and the deployment of a novel remote voting system called sElect (secure/simple elections). sElect is designed to be particularly simple and lightweight in terms of its structure, the cryptography it uses, and the user experience. One of its unique features is a fully automated procedure which does not require any user interaction and it is triggered as soon as voters look at the election result, allowing them to verify that their vote has been properly counted. The component of sElect which provides vote privacy is implemented in Java such that we can establish cryptographic guarantees directly on its implementation: by combining the techniques of the CVJ framework with a hybrid approach for proving noninterference, we are able to show that the Java implementation ensures strong cryptographic privacy of the votes cast with our proposed voting system.

sElect is therefore the first full-fledged e-voting system with strong cryptographic security guarantees not only at the design level, but also on its implementation.

Kurzzusammenfassung

Die formale Verifizierung der Sicherheitseigenschaften von kryptographischen Operationen, wie Verschlüsselung, und darauf basierenden Protokollen, wie TLS, war in den letzten drei Jahrzehnten das Ziel umfangreicher Forschung. Eine grundlegende Einschränkung bei der Verifikation von Sicherheitseigenschaften besteht darin, dass Analysen typischerweise auf der Entwurfsebene durchgeführt werden und daher keine verlässlichen Garantien für die Implementierung bieten. Um diese Einschränkung zu überwinden, zielt diese Arbeit darauf ab, formal untermauerte kryptographische Garantien für Systeme, die in Java geschrieben sind und Kryptographie verwenden, direkt auf der Implementierungsebene zu etablieren. Unser Ansatz basiert auf einem allgemeinen Framework für die kryptographische Verifikation von Java-Programmen (das CVJ-Framework), das kryptographische Ununterscheidbarkeitseigenschaften und Nichtinterferenzeigenschaften formal miteinander verknüpft. Dieses Framework ermöglicht es bestehenden Tools, die bereits Standard-Nichtinterferenzeigenschaften prüfen können, aber a priori nicht mit Kryptographie umgehen können, auch kryptographische Privacy-Eigenschaften für Java-Systeme zu etablieren. Das CVJ-Framework ist ausgelegt für eine Java-ähnliche formale Sprache, die jedoch nicht alle in Java-Programmen üblichen Datentypen und Features abdeckt. Außerdem unterstützt das Framework ursprünglich nur eine einzige kryptographische Operation, nämlich die Public-Key-Verschlüsselung.

Der erste Beitrag dieser Arbeit ist daher die Erweiterung und Instanziierung des CVJ-Frameworks, um es breiter anwendbar zu machen. Wir erweitern die zugrundeliegende formale Sprache um einige Features von Java, die noch nicht erfasst wurden, wie z.B. Concurrency, und überarbeiten und beweisen alle Ergebnisse des CVJ-Frameworks, um sie in die erweiterte Sprache zu übertragen. Anschließend instanziieren wir das Framework mit zusätzlichen kryptographischen Operationen, nämlich digitale Signaturen und Public-Key-Verschlüsselung, die beide nun auch eine Public-Key-Infrastruktur, (private) symmetrische Verschlüsselung und Nonce-Generierung unterstützen. Die im Rahmen des CVJ-Framework entwickelten Methoden und Techniken sind auch unabhängig von den verwendeten Werkzeugen relevant und anwendbar. Um die Nützlichkeit des Ansatzes zu verdeutlichen, wenden wir das Framework zusammen mit zwei Verifikationswerkzeugen für Java-Programme an, nämlich dem vollautomatischen statischen Analysewerkzeug Joana und dem interaktiven Theorembeweiser KeY, um starke kryptographische Privacy-Eigenschaften für Systeme zu etablieren, die Kryptographie verwenden (wie Client-Server-Anwendungen und E-Voting-Systeme).

In diesem Zusammenhang ist der zweite wichtige Beitrag dieser Arbeit das Design, die Umsetzung und der Einsatz eines neuartigen Remote-Voting-System namens sElect (secure/simple elections). sElect ist so konzipiert, dass es in Bezug auf seine Struktur, die verwendete Kryptographie und die Benutzerfreundlichkeit besonders einfach und leichtgewichtig ist. Eine seiner einzigartigen Eigenschaften ist ein vollautomatisches Verfahren zur Verfikation von Stimmabgaben. Die Komponente von sElect, die die Vertraulichkeit der Stimmabgabe gewährleistet, ist in

Java implementiert, so dass wir kryptographische Garantien direkt für die Implementierung etablieren können: Durch die Kombination der Techniken des CVJ-Frameworks mit einem hybriden Ansatz zum Nachweis der Nichtinterferenz können wir zeigen, dass die Java-Implementierung eine starke kryptographische Vertraulichkeit der abgegebenen Stimmen mit unserem vorgeschlagenen Abstimmungssystem gewährleistet.

Damit ist sElect das erste vollwertige E-Voting-System mit starker kryptographischer Sicherheit, die nicht nur auf der Design-, sondern auch auf der Implementierungsebene garantiert ist.

Contents

Abstract 1

Kurzzusammenfassung 3

1. Introduction 9

1.1. Extension of the CVJ Framework 12
1.2. Instantiation of the CVJ Framework 13
1.3. Application of the CVJ Framework 13
1.4. E-voting Systems and sElect 15
1.5. Structure of the thesis 16

I The CVJ Framework, a Framework for the Cryptographic Verification of Java Programs 19

2. Extending the CVJ Framework to Java-Interfaces, Abstract Classes, and Strings 21

2.1. Jinja+: A Java-like language 22
2.2. Indistinguishability 25
2.3. Simulatability and Universal Composition 26
2.4. I-Noninterference, Noninterference in Open Systems 27
2.5. From I-Noninterference to Computational Indistinguishability 28
2.6. A Proof Technique for proving I-Noninterference 28
2.6.1. Communication through Primitive Types Only 29
2.6.2. Communication through Arrays, Simple Objects, and Exceptions . . . 31
2.6.3. Communication through Strings 32

3. Extending the CVJ Framework to Java Concurrency 37

3.1. Concurrency in Java 38
3.2. SyncJinja+ systems 40
3.2.1. Single-Threaded Semantics of SyncJinja+ 40
3.2.2. Multi-Threaded Semantics of SyncJinja+ 41
3.2.3. Run of a SyncJinja+ program 43
3.3. Indistinguishability 44
3.3.1. Interfaces and Composition 44
3.3.2. Environments 45
3.3.3. Programs with security parameter 46
3.3.4. Perfect Indistinguishability 47
3.3.5. Polynomially Bounded Systems 48

3.3.6. Computational Indistinguishability 49
3.4. Simulatability and Universal Composition 50
3.5. From Perfect to Computational Indistinguishability 53
3.6. Perfect Indistinguishability and Noninterference 55
3.7. From Noninterference to Computational Indistinguishability 57
3.8. From Single-Threaded to Multi-Threaded Programs 58

4. Instantiating and Applying the CVJ Framework 61
4.1. Public-Key Encryption with a Public Key Infrastructure 62
4.1.1. The Interface for Public-Key Encryption 63
4.1.2. The Ideal Functionality for Public-Key Encryption 64
4.1.3. The Realization of Ideal-PKIEnc 66
4.1.4. Realization Result 66
4.2. Digital Signatures with a Public Key Infrastructure 68
4.2.1. The Interface for Digital Signatures 68
4.2.2. The Ideal Functionality for Digital Signatures 69
4.2.3. The Realization of Ideal-Sig 70
4.2.4. Realization Result 70
4.3. Private Symmetric Encryption 71
4.4. Nonce Generation 72
4.5. Joana, a Static Checker for proving Noninterference 72
4.6. The Case Study: A Cloud Storage System 73

5. Related Work and Discussion 79

II sElect, a Lightweight Verifiable Remote Voting System 85

6. E-voting Systems and their Security Properties 87

7. The sElect E-voting System and its main features 91
7.1. sElect in a nutshell 91
7.2. Main features of sElect 92

8. Design, Implementation, and Deployment of the sElect E-voting System 95
8.1. Design of sElect 95
8.2. Implementation of sElect 99
8.3. Deployment of sElect 107

9. Formal Verification of the sElect E-voting System 111
9.1. Verification of the Mix Server 111
9.2. A Hybrid Approach for Proving Noninterference of Java Programs 113
9.3. KeY, a Theorem Prover for sequential Java Programs 115
9.4. Applying the Hybrid Approach to Verify the Mix Server 119

10. Related Work and Discussion **123**

11. Conclusion and Future Work **129**

III Appendices **133**

A. Security Notions for Cryptographic Schemes **135**
A.1. IND-CCA2-secure Public-Key and Symmetric Encryption Schemes 135
A.2. EUF-CMA-secure Digital Signatures Schemes 136

B. The Jinja+ and SyncJinja+ languages **139**
B.1. Small-Step Semantics of Jinja, Jinja+, and SyncJinja+ 143
B.1.1. Semantics Rules of Jinja 143
B.1.2. Semantics Rules of the Jinja+ extension 148
B.1.3. Semantics Rules for the data type `String` 150
B.1.4. Semantics Rules of the SyncJinja+ extension 151

C. The Environment/Adversary **155**

D. Real and Ideal Cryptographic Functionalities **157**
D.1. The Public Key Infrastructure 157
D.2. PKIEnc: Public Key Encryption with a Public Key Infrastructure 159
D.2.1. Ideal Functionality for Public Key Encryption without Corruption . . . 163
D.3. PKISig: Digital Signature with a Public Key Infrastructure 163
D.3.1. Ideal Functionality for Digital Signatures without Corruption 166
D.4. Private Symmetric Encryption 167
D.5. Nonce Generation 168

E. Case Studies **169**
E.1. A Cloud Storage System 169
E.2. An E-voting Machine with Auditing Procedures 176
E.3. The Mix Server of sElect 181

F. Formal Proofs **187**
F.1. Proof of Theorem 2.5 187
F.2. Proof of Equivalence Relation of $\equiv_{\mathsf{comp}}$ 193
F.3. Proof of Theorem 3.6 195
F.4. Proof of Theorem 4.1 201
F.4.1. Proof of Lemma F.11 203
F.4.2. Proof of Lemma F.12 213
F.5. Proof of Theorem 4.2 216
F.5.1. The Functionality without Corruption 217
F.5.2. Proof of Lemma F.22 222

Bibliography **225**

Academic Curriculum **243**

1. Introduction

Security and correctness of software systems have become a fundamental problem for our society: severe vulnerabilities are found in security critical applications and cryptographic protocols used by hundreds of millions of people, e.g., for on-line transactions, and attacks compromising millions of computers are reported in the news almost daily.

To address these issues, in the last three decades or so there has been a substantial research effort in formally analyzing cryptographic operations, such as encryption, and protocols based on them, such as TLS. Nevertheless, while severe vulnerabilities in widely used cryptographic protocols keep on being uncovered (see, e.g., [APW09, ABD$^+$15, BL16, VP17]), performing security analyses on these protocols still remains a challenging and tricky task. We can distinguish three main types of approach aiming at formally proving security properties of cryptographic operations and/or protocols.

The first type is based on symbolic/Dolev-Yao models [DY83]. These models abstract from cryptographic details by modeling messages as terms of some algebra and cryptographic primitives as deterministic operations on that algebra. These terms are handled symbolically in the proofs, i.e., they are never evaluated to bitstrings. Tools like ProVerif [Bla01] and Tamarin [MSCB13] have been proposed to support symbolic analyses of various cryptographic properties, such as secrecy and authentication. Although simple and efficient, these models suffer from some limitations: the adversary they consider is restricted to be a non-deterministic state machine defined as a set of rules for manipulating the terms of the specific algebra. Moreover, these models make the unrealistic assumption of "perfect cryptography", i.e., that cryptography cannot be broken by any means.

The second type of approach is based on computational/cryptographic models [GM82b, Yao82], where messages are concretely considered as bitstrings and cryptogtraphic primitives as functions from bitstrings to bitstrings. In this more precise models, the adversary is defined as a probabilistic polynomial-time Turing machine and the security of the cryptographic primitives is reduced to well-known hard problems, such as factorization or discrete logarithm. Tools like CryptoVerif [Bla06] and EasyCrypt [BGHB11] have been proposed to automatically construct game-based cryptographic proofs [Mih04] of both cryptographic primitives and security protocols.

However, one fundamental limitation of both these two types of approach is that analyses are typically carried out at the specification level, i.e., on the design level of the cryptographic protocols, and hence, they do not provide reliable guarantees on the actual implementations of such protocols. To overcome this limitation, over the last few years some approaches have been developed in the domain of *code-based provable security*, where cryptographic analyses are performed directly at the implementation level.

In this thesis, we focus in one of this last type of approach: we are concerned with the problem of building computationally sound analyses of systems that are coded in Java and involve the use of cryptography. More specifically, our aim is to establish formally justified cryptographic

security guarantees for such systems on the implementation level, that is, directly for the running Java code of these systems.

Our approach is based on a general framework for the cryptographic verification of Java programs [KTG12a] (henceforth, we refer to this framework as the CVJ framework) which allows us to carry out semantically sound cryptographic analyses of programs written in (a large fragment of) Java. Due to its intrinsic entanglement, implementation-level analysis of cryptographic properties has only recently gained the attention for practical programming languages and the CVJ framework is currently the only approach to formally establish cryptographic security guarantees for Java programs. The only other approach aiming at establishing cryptographic guarantees directly at the implementation level is based on F* [SHK+16] and other closely related languages (such as F# [Don15] and RF* [BFG+14]), a family of functional-first programming languages built with the specific aim of program verification (see, e.g., [CB13, BFK+13, DFK+17] for some works using this approach to verify cryptographic implementations).

The security guarantees that we want to establish on Java implementations containing cryptographic operations are related to the concept of *indistinguishability* [GM82b]: Two systems S_1 and S_2 are considered to be indistinguishable if no probabilistic polynomially bounded adversary is able to distinguish, with more than negligible probability, whether it interacts with S_1 or S_2. For example, an encryption scheme is defined to be "secure" according to this notion if no adversary is able to distinguish the encryption of the bit 0 from the encryption of the bit 1. Indistinguishability plays a central role in precisely asserting the level of confidentiality of both cryptographic operations and security protocols and it is in fact the most fundamental definition in the formal evaluation of many security critical applications, such as anonymous communication, secure message transmission, e-voting, etc.

To formally establish this definition on the code level, results within the CVJ framework formally link indistinguishability properties with *noninterference* properties. Noninterference is a confidentiality property for programs which characterizes secure information flow by the requirement that private data stored in so-called "high variables" does not flow to public "low variables" which can directly be observed by an attacker [GM82a].

Existing program analysis tools for checking noninterference properties cannot deal with cryptography directly. In particular, the noninterference properties that they ensure are with respect to unbounded adversaries, rather than probabilistic polynomially bounded adversaries, the kind of adversaries considered in cryptography. Hence, if a message is encrypted and the ciphertext is given to the adversary, the tools consider this to be an illegal information flow (or a declassification), because a computationally unbounded adversary could in fact decrypt the message. The approach taken in the CVJ framework to enable these tools to nevertheless deal with cryptography and, in the end, to provide cryptographic security guarantees is to use techniques from simulation-based security/universal composability (see, e.g., [Can00, PW01, Küs06, KT13]). The central idea of this framework is to first analyze noninterference properties for a Java program where cryptographic operations (such as encryption) are performed by so-called ideal functionalities. Such functionalities typically provide guarantees even in the face of unbounded adversaries and can often be formulated without probabilistic operations. One central result of the CVJ framework is that, given that the Java program with ideal functionalities enjoys noninterference, we can replace the ideal functionalities by their realizations (the actual

cryptographic operations) and then obtain strong cryptographic indistinguishability properties for the original Java program, that is, the one without idealized components. The methods and techniques of the CVJ framework are relevant and usable independently of any specific application domain. In particular, they are not tailored to any specific cryptographic operation. However, to show the applicability of this approach, the authors of [KTG12a] have proposed an ideal functionality written in Java for public-key encryption, provided its realization, i.e., a Java implementation of a real public-key encryption scheme, and proven their relation under strong cryptographic assumptions. They have already presented a simple case study where clients (whose number is determined by an active adversary) encrypt secrets under the public key of a server and send them, over an untrusted network controlled by the active adversary, to a server who decrypts these messages. Using the program analysis tool Joana [HS09a, GHM13], which is a fully automatic tool for proving noninterference properties of Java programs, the client-server application has been shown to be noninterferent (with the secrets stored in high variables) when the ideal functionality is used instead of the real public-key encryption scheme. By the results of the CVJ framework, this system then enjoys the desired indistinguishability property when the ideal functionality is replaced by its realization, i.e., the real public-key encryption scheme.

Our contribution. Starting from the results and the aforementioned first application of the CVJ framework, we make the following contributions:

1. We extend Jinja+, the formal language the framework is stated for, with some not yet modeled features of Java: Java-interfaces, abstract classes, strings, and two features of Java concurrency, namely thread creation and synchronized blocks. We then restate all the definitions and prove all the results to carry them over into this extended language (see Section 1.1 below).

   The extension of the CVJ framework with Java-interfaces, abstract classes, and strings has been presented at the *Workshop on Foundations of Computer Security (FCS 2015)* [Sca15a] and the technical report is available at [Sca15b]. The extension of the CVJ framework with the two concurrency features is not yet published but presented in details in Chapter 3.

2. We make the CVJ framework more widely applicable by further instantiating it with real and ideal functionalities for the most common cryptographic operations used in security critical applications (see Section 1.2 below).

   The instantiation with real and ideal cryptographic functionalities has been presented at the *3rd Conference on Principles of Security and Trust (POST 2014)* [KSTG14a] and the technical report is available at [KSTG14b]. The runnable code of these functionalities can be found in [TSK13].

3. We illustrate the applicability of the CVJ framework by applying it to three non-trivial case studies (see Section 1.3 below).

   The first two non-trivial case studies the CVJ framework has been applied to are part of a research paper presented at the *3rd Conference on Principles of Security and Trust (POST 2014)* [KSTG14a] and another research paper presented at the *IEEE 28th Computer*

Security Foundations Symposium (CSF 2015) [KTB+15], respectively. For the latter research paper, our contribution has been to provide the code for the case study as well as the code for the "previous experiments" mentioned in the related work section of [KTB+15].

The full code of these two case studies as well as the instructions on how to use the employed tools to carry out their analyses can be found in [STG13] and [STB+14a], respectively. The code of the aforementioned "previous experiments" can be found in [STB+14b].

At the time of writing, the analysis of the third case study, namely the cryptographic core of the sElect e-voting system (see below), is mostly finished but ongoing work. We refer the reader to Chapter 9 for the full details. The code of this case study as well as the part of the verification which has already been carried out can be found in [SHM17].

4. We introduce a novel remote e-voting system called *sElect* (secure/simple elections), which starting from simply being a case study for the CVJ framework eventually evolved into a full-fledged and already deployed remote e-voting system (see Section 1.4 below).

   sElect has been presented at the *IEEE 29th Computer Security Foundations Symposium (CSF 2016)* [KMST16a] and the technical report is available at [KMST16b]. Our proposed implementation of the system and an election manager for its fast deployment can be found in [SST16] and [SS16], respectively.

In the rest of the introduction, we discuss these contributions in more detail.

1.1. Extension of the CVJ Framework

The CVJ framework is stated and proven for a sequential language called *Jinja+* and is proven w.r.t. the formal semantics of this language. Jinja+ is a Java-like language that extends the language Jinja [KN06] with some useful additional features of Java, such as arrays and the primitive types `int`, `boolean`, and `byte`. Moreover, since the kind of systems considered in cryptography are possibly randomized, Jinja+ also includes the primitive `randomBit()` returning a random bit each time it is called.

In Chapter 2, we further extend Jinja+ with Java-interfaces, abstract classes, and the data type `String`. Furthermore, in Chapter 3, we model two concurrent features of Java, namely thread creation and synchronized blocks. While these features do not capture all the aspects of Java concurrency, they are sufficient to model an interesting class of programs, such as multi-threaded client-server applications. These extensions of Jinja+ result in an extended language called SyncJinja+ whose small-step semantics comprehends 25 newly introduced reduction rules in addition to the 70 already part of Jinja+. We then extend all the definitions, restate and prove all the results of the CVJ framework to deal with the newly introduced types of values, multi-threaded constructs, and semantics rules. In particular, to formally link cryptographic indistinguishability properties to noninterference properties in a concurrent setting, we propose a novel notion of noninterference for concurrent Java programs, which substantially differs from all the other concurrent noninterference definitions present in the literature.

An adversary running concurrently with an honest program can measure the runtime of that program. Therefore, when extending the framework with concurrency, proving cryptographic guarantees based on standard cryptographic assumptions would not be possible anymore: cryptographic security definitions are usually stated as sequential games between the cryptographic scheme and the adversary, and hence, they do not capture timing attacks. SyncJinja+ therefore allows one to declare parts of the code to be executed *atomically*, i.e., without interleaving by any other component of the modeled system, such as another thread or the scheduler.[1] Then, the main result of Chapter 3 is that if a single-threaded system R realizes another single-threaded system F, then this realization carries over also to multi-threaded adversaries, as long as the assumption of considering R and F atomic is maintained.

1.2. Instantiation of the CVJ Framework

The theorems proven within the CVJ framework are very general in that they guarantee that any ideal functionality can be replaced by its realization. In particular, they are not tailored to any specific cryptographic operation. However, to make the framework directly applicable to a wide range of cryptographic software, i.e., software that uses cryptographic operations, in Chapter 4, we further instantiate the CVJ framework with ideal functionalities formulated in Java for the most common cryptographic operations:

(1) public-key encryption,

(2) digital signatures,

both with a public-key infrastructure for distributing the (public) keys among parties and both handling possibly corrupted encryptors/signers,

(3) private symmetric encryption,

(4) nonce generation.

For each ideal functionality, we propose a corresponding real cryptographic operation also implemented in Java and we prove, in the universal composability model, that this real implementation realizes the corresponding ideal functionality under strong cryptographic assumptions. Once again, showing that a Java system using any of these ideal functionalities enjoys the noninterference property implies establishing cryptographic indistinguishability for the same system when the ideal functionalities are replaced by the corresponding real cryptographic operations.

1.3. Application of the CVJ Framework

To illustrate the applicability and the usefulness of the CVJ framework, besides the simple client-server application proposed as a first case study, we use the aforementioned cryptographic operations and ideal functionalities to develop and verify three other non-trivial applications:

[1] In our formalization, the scheduler is modeled as a separate, single-threaded Jinja+ program.

(a) A cloud storage system allowing a user to store data on a remote server in such a way that confidentiality of the data stored on the server is guaranteed even if the server itself is malicious (see Section 4.6 and [STG13]).

(b) An e-voting machine which gathers all the votes, calculates the election result, and publishes it, along with some other information for auditing, to a publicly available bulletin board (see Section 9.3 and [STB+14a]).

(c) The component of the sElect e-voting system (see next section) hiding the link between the voters' encrypted ballots and their original choices, namely the mix server: upon receiving a list of ballots encrypted with its public key, the mix server decrypts all the ciphertexts, shuffles the decrypted ballots, and outputs them digitally signed (see Section 9.4 and [SST16]).

We used the results of the CVJ framework to establish cryptographic privacy properties on these three case studies. In particular, for the cloud storage system we are interested in establishing confidentiality of the messages stored on the remote server. For the two e-voting systems our aim is instead to establish confidentiality of the votes cast by honest voters.

However, when checking noninterference properties in e-voting systems, the *functional dependency* between the secret voters' choices and the public outcome of the election raises additional challenges in establishing cryptographic vote privacy. Due to the approximations employed, fully automatic tools are in fact unable to establish functional properties on the programs they verify: for instance, given an e-voting system, automatic tools cannot state whether two different vectors of votes produce the same election result or not.

To cope with this problem, a hybrid approach for proving noninterference has been proposed in [KTB+15] combining the convenience of using fully automatic tools, such as static checkers, with the precision of interactive tools, such as theorem provers. The idea underlying this approach is as follows. If the verification of noninterference of a program using an automatic tool fails due to (what we think are) false positives, then additional code is added to the program in order to make it more explicit for the automatic tool that there is no illegal information flow, and by this, to avoid false positives. If the automatic tool now establishes that the extended program enjoys the desired noninterference property, it remains to show that the extended program is a *conservative extension* of the original program. Intuitively, this means that the additional code did not change the behavior of the original program in an essential way, including, importantly, noninterference properties. Proving that an extension is conservative requires to prove functional properties of (parts of) the program and it can therefore only be performed by a more precise but interactive theorem prover.

The verification of the case studies (b) and (c) has been carried out using a combination of Joana and KeY [ABB+14], a program verification system for sequential Java which is based on an interactive theorem prover for first-order dynamic logic (JavaDL) [Bec00]. The logical calculus for JavaDL which allows for the functional verification with KeY precisely reflects the semantics of sequential Java, i.e., it does not make any approximation. This however comes at the price of a major human interaction in building such proofs. Therefore, on the one hand, the precision of KeY is used to address the problem of the functional dependency between the secret voters' choices and the public outcome of the election. On the other hand, the high degree

of automation of Joana is used to quickly establish the noninterference property on the overall voting systems.

1.4. E-voting Systems and sElect

Using e-voting systems as motivating examples and case studies for the methods and techniques developed to perform cryptographic analyses on the code level also contributes to the active research field of studying and verifying e-voting protocols. In the last decade or so, there has been an extensive effort in formally defining the security properties that e-voting systems are supposed to provide (see, e.g., [KTV10a, KTV10b, KTV11, BCG$^+$15, CGK$^+$16]):

(i) *Privacy* guarantees that nobody is able to tell how each voter voted.

(ii) *Verifiability* ensures the possibility to verify the correctness of the election result, even if some components of the e-voting system or some election authorities are (partially) malicious.

(iii) *Accountability* is a stronger form of verifiability which requires not only that a fraudulent election result is detected but also that misbehaving parties are properly singled out and held accountable.

(iv) *Coercion-resistance* protects voters against vote buying and vote coercion.

The formalization of these security properties has been used to perform cryptographic analyses on several e-voting systems finding attacks and revealing some misconceptions concerning their relations (see, e.g., [KTV10a, KTV10b, KTV10c, KTV11, CS11, KTV12b, ACW13, CEK$^+$15, KZZ15a, KMST16a, CCFG16, KZZ17, CDD$^+$17, CW17]).

However, when deploying such systems, besides theoretical cryptographic analysis, many other more practical aspects have to be taken into account, considering possible vulnerabilities, implementation flaws, deployment misconfigurations, and so on. In addition, human actors (voters, clerks, authorities, auditors, etc.) play also a central role in the overall electoral process, making sociotechnical facets, such as usability, perceived security, and legal requirements, as relevant as security technicalities during the design, the implementation, and the deployment of such systems. To increase transparency and reliability in the overall electoral process, newly proposed e-voting systems therefore strive also for simplicity and high usability: In this way, their architecture can easily be understood even by non experts and human actors can take part in the electoral process effortlessly.

Simplicity and usability were in fact the main requirements of *sElect* (simple/secure elections), a novel remote voting system which we designed, implemented, and deployed in real-world elections. sElect is meant for low-risk elections,[2] such as elections within companies, clubs, associations, etc., and it is designed to be particularly simple and lightweight in terms of its

[2]Low-risk elections are elections where coercion of human actors is unlikely and where there is no pragmatical need to defend against sophisticated attacks on the voter supporting devices (VSDs; these are the devices that voters use to cast their ballots).

structure, the cryptography it uses, and the user experience. One of its unique features is a fully automated procedure which does not require any user interaction and it is triggered as soon as voters look at the election result, allowing them to verify that their vote has been properly counted. In addition, voters can also make sure that their vote is in the final tally by manually checking whether the verification code they have chosen themselves when casting their vote appears in the election result along with their choice. Thanks to these two verification procedures, besides vote privacy, sElect enjoys a high level of verifiability and accountability.

The use of encryption, digital signatures, and nonce generation as the sole cryptographic primitives combined with the two forms of verification procedures makes the architecture of the system and the overall voting process understandable also by non experts, potentially raising the level of transparency and reliability perceived by the human actors involved in the electoral process. Furthermore, sElect comes with a rigorous cryptographic analysis at design level with respect to the three security properties mentioned above: privacy, verifiability, and accountability. As shown in [KMST16a], despite its simplicity and high usability, the level of security of sElect is comparable to other prominent remote voting systems, such as Helios [Adi08] (we refer the reader to Chapter 10 for a comprehensive discussion between the two systems). However, since the cryptographic analysis does not consider the actual code of the system but rather a more abstract cryptographic model, implementation flaws and vulnerabilities can still go undetected. Therefore, in Chapter 9 we employ the techniques of the CVJ framework in combination with the aforementioned hybrid approach for proving noninterference to establish strong cryptographic guarantees on the implementation of the mix server, the component of sElect which provides vote privacy. While this analysis does not ensure the absence of any vulnerability, we are able to guarantee that the proposed implementation provides strong cryptographic privacy of the votes cast with sElect.

1.5. Structure of the thesis

The thesis is structured in two parts.

In the first part, we propose two extensions of the CVJ framework, its instantiations, and its applications. More specifically, in Chapter 2, we extend Jinja+, the Java-like language the framework is stated for, to Java-interfaces, abstract classes, and strings. We then recall the main definitions and results of the CVJ framework, restating the only result which requires non trivial modifications to carry it over into this extended language, namely a proof technique for proving noninterference in open systems. In Chapter 3, we extend Jinja+ with two features of Java concurrency, namely thread creation and synchronized blocks. Again, we restate and prove all the results of the framework w.r.t. this concurrent language, introducing a novel notion of noninterference which is suitable for our purposes. In Chapter 4, we instantiate the CVJ framework with the most common cryptographic primitives and we illustrate the usefulness of this instantiation by applying the framework along with the fully automatic tool Joana to establish cryptographic privacy properties of a (non-trivial) cloud storage application, where clients can store private information on a remote server. Finally, in the related work (Chapter 5) we mainly discuss how our novel notion of concurrent noninterference relates to the other concurrent noninterference definitions in the literature.

In the second part, after introducing in Chapter 6 the problem of e-voting and the main security properties of e-voting systems, we describe sElect [KMST16a, KMST16b], a novel verifiable remote voting system, as well as the cryptographic analysis perfomed on its implementation. In particular, after introducing, in Chapter 7, the main characteristics of sElect, in Chapter 8 we describe the components (Section 8.1), the development (Section 8.2), and the deployment (Section 8.3) of our proposed e-voting system. In Chapter 9, we explain our efforts in establishing strong cryptographic guarantees on the component of sElect which provides privacy of votes, the mix server (Section 9.1). In particular, we explain the technique (Section 9.2), the tools (Section 9.3), and the method (Section 9.4) employed to obtain these guarantees. In the related work (Chapter 10), we compare sElect with the most prominent remote voting system in the literature, the Helios voting system [Adi08].

In Appendix A, we present the formal definitions of the cryptographic assumptions which our cryptographic functionalities and their realization results are based on. In Appendix B, we present the language at the base of all our formal results, SyncJinja+, including all its 95 small-step reduction rules. After explaining in Appendix C our modeling of the environment/adversary in SyncJinja+, in Appendix D, we list the code of the real and ideal functionalities of the cryptographic primitives the realization results are stated and proven for. The case studies employing these functionalities and for which we established security guarantees on the code level are listed in Appendix E. Finally, Appendix F contains all the formal proofs of the theorems which are too long for being incorporated in the main body of the thesis.

PART I

The CVJ Framework, a Framework for the Cryptographic Verification of Java Programs

2. Extending the CVJ Framework to Java-Interfaces, Abstract Classes, and Strings

As already mentioned in the introduction, the CVJ framework considers the problem of establishing security guarantees for Java-like programs which use cryptography. In particular, the CVJ framework allows tools that can check standard noninterference properties [GM82a] but a priori cannot deal with cryptography, in particular probabilities and polynomially bounded adversaries, to establish cryptographic indistinguishability properties [GM82b], such as privacy properties, for Java programs. The framework combines techniques from language-based information-flow security [SM03] and universal composability [Can00, PW01, Küs06, KT13], a well-established concept in cryptography. The idea is to first check noninterference properties for the Java program to be analyzed where cryptographic operations (such as encryption) are performed within so-called ideal functionalities. Such functionalities typically provide guarantees even in the face of unbounded adversaries and can often be formulated without probabilistic operations. Therefore, such analyses can be carried out by tools that a priori cannot deal with cryptography, as they enforce security properties w.r.t. unbounded adversaries. Theorems within the framework now imply that the Java program enjoys strong cryptographic indistinguishability properties when the ideal functionalities are replaced by their realizations, i.e., the actual cryptographic operations.

The CVJ framework is formulated in [KTG12a] for a *sequential* language called Jinja+ and is proven w.r.t. the formal semantics of this language. Jinja+ is a Java-like language that extends the language Jinja [KN06] with, among others, arrays, the type `byte`, and the `abort` primitive. We refer the reader to Appendix B for the details of the language.

To make this framework applicable to a wider class of Java programs, in this chapter we further extend the syntax and the semantics of Jinja+ introduced in Section 2.1 with:

(i) Java-interfaces,

(ii) abstract classes,

(iii) strings.

Except for one result, namely a proof technique for checking noninterference in open systems, all the definitions and results of the CVJ framework carry over easily to the extended language. That is, the new types of values and the new rules of the augmented small-step semantics do not affect the proof in a significant way. These definitions and results are therefore presented here in a more simplified and informal way which should anyway suffice to follow the rest of the thesis. We refer the reader to [KTG12a] for the details and to [KTG12b] for the proofs of all the results.

As to checking noninterference, many program analysis tools can only deal with closed Java programs. The systems to be analyzed are, however, often open: they interact with a network or use some libraries which are not necessarily trusted and, hence, are not part of the code to be

analyzed; instead, they are considered to be part of the environment with unspecified behavior. A *proof technique* has therefore been proposed to reduce the problem of checking noninterference in an open system to checking noninterference for a single (almost) closed system. Technically, this result shows how to construct, for an open system S, a family of environments $\tilde{E}_{\vec{u}}$ parameterized by an input sequence $\vec{u}$, such that S is noninterferent if and only if S composed with $\tilde{E}_{\vec{u}}$ is noninterferent for all $\vec{u}$. Importantly, the latter property can be verified using existing tools for program analysis. This result requires non-trivial modifications to model the exchange of data between the system and the environment when also strings are involved: In fact, the exchange of data through string references introduces subtle changes in the original result and, technically, invalidates the main assumption the result is based on, i.e., the separation between the state of the system and the state of the environment. In this chapter, we therefore extend the construction of $\tilde{E}_{\vec{u}}$ to also handle the exchange of string references. Relying on the fact that the Java (Jinja+) strings are *immutable*, we relax the state separation assumption, we adapt the proof in a non-trivial way to work with the new (relaxed) assumption, and we finally reshape the proof technique for proving noninterference in open systems taking into account string references, too.

2.1. Jinja+: A Java-like language

In this section, we present Jinja, a Java-like programming language with a formal semantics, its extension Jinja+, which adds additional features that are useful in the context of the CVJ framework, and finally the three aforementioned extensions: abstract classes, Java-interfaces, and strings.

Jinja. Expressions in Jinja are constructed recursively and include: (a) creation of a new object, (b) casting, (c) literal values (constants) of types boolean and int, (d) `null`, (e) binary operations, (f) variable access and variable assignment, (g) field access and field assignment, (h) method call, (i) blocks with locally declared variables, (j) sequential composition, (k) conditional expressions, (l) while loop, (m) exception throwing and catching.

A *program* or a *system* is a set of class declarations. A *class declaration* consists of the name of the class and the class itself. A *class* consists of the name of its direct superclass (optionally), a list of field declarations, and a list of method declarations, where we require that different fields and methods have different names. A *field declaration* consists of a type and a field name. A *method declaration* consists of the method name, the formal parameter names and types, the result type, and an expression (the method body). Note that there is no `return` statement, as a method body is an expression; the value of such an expression is returned by the method. In what follows, by a *system* we will mean a set of classes which is syntactically correct (can be compiled), but possibly incomplete (can call methods of not defined classes). In particular, a system can be extended to a program. By a *program* we will mean a complete program (one that is syntactically correct and can be executed). We assume that a program contains an unique static method `main` (declared in exactly one class); this method is the first to be called in a run. Jinja comes equipped with a type system and a notion of well-typed programs. In this thesis we consider only well-typed programs.

Following [KN06], we briefly sketch the small-step semantics of Jinja. The full set of rules can be found in the appendix, Figures B.0, B.1, B.2, and B.3. A *state* s is a pair of *heap*

```
1  class A extends Exception {
2    protected int a; // field with an access modifier
3    static public int[] t = null; // static field
4    static public void main() { // static method
5      t = new int[10]; // array creation
6      for (int i=0; i<10; i++) // loops
7        t[i] = 0; // array assignment
8      B b = new B(); // object creation
9      b.bar(); // method invocation
10   }
11 }
12 class B extends A { // inheritance
13   private int b;
14   public B() // constructor
15     { a=1; b=2; } // field assignment
16   int foo(int x) throws A { // throws clause
17     if (a<x) return x+b;  // field access (a, b)
18     else throw (new B()); // exception throwing
19   }
20   void bar() {
21     try { b = foo(A.t[2]); } // static field access
22     catch (A a) { b = a.a; } // exception catching
23   }
24 }
```

Figure 2.1.: An example Jinja+ program (in Java-like notation).

h, a *store* l. A *store* is a map from variable names to values, $l\colon Var \to Val$, where $Val ::= int \mid bool \mid loc \mid null \mid unit$. A *heap* is a map from locations (we indicate them also as references or addresses) to *object instances*, $h\colon Loc \to Obj$. An *object instance* is a pair consisting of a class name and a *field table*, $Obj\colon Name \times Field$. A field table is a map from field names (which include the class where a field is defined) to values, $Field\colon Name \to Val$. The small-step semantics of Jinja is given as a set of rules (Rules B.1-B.54) of the form $P \vdash \langle e,s\rangle \to \langle e',s'\rangle$, describing a single step of the program execution (reduction of an expression). We will call $\langle e,s\rangle$ (and $\langle e',s'\rangle$) a *configuration*. In this rule, P is a program in the context of which the evaluation is carried out, e and e' are expressions and s and s' are states. Such a rule says that, given a program P and a state s, an expression e can be reduced in one step to e', changing the state to s'.

Jinja+. Jinja+ extends the Jinja language with: (a) the primitive type `byte` with natural conversions from and to `int`, (b) arrays, (c) `abort` primitive, (d) static fields (with the restriction that they can be initialized by literals only), (e) static methods, (f) access modifier for classes, fields, and methods (such as `private`, `protected`, and `public`), (g) final classes (classes that cannot be extended), (h) the `throws` clause of a method declaration.

Exceptions, which are already part of Jinja, are particularly critical for the security properties stated in the CVJ framework because they provide an additional way information can be transferred from one part of the program to another. We assume Jinja+ programs have unbounded

memory. The reason for this modeling choice is that the formal foundation for the security notions adopted in the CVJ framework are based on *asymptotic* security. This kind of security definitions only makes sense if the memory is not bounded, since the security parameter grows indefinitely. Furthermore, since the CVJ framework needs to consider also randomized program, Jinja+ programs may use the primitive `randomBit()` that returns a random bit each time it is used. Jinja+ programs that do not make use of `randomBit()` are (called) *deterministic*, and otherwise, they are called *randomized*. Some constructs of Jinja+ are illustrated by the program in Figure 2.1, where we use Java-like syntax (we will use this syntax as long as it translates in a straightforward way to a Jinja/Jinja+ syntax).

Figures B.4, B.5 in the appendix contain the small-step rules for the Jinja+ extension: together with the rules of Jinja (in Figures B.0, B.1, B.2, and B.3), they describe the small-step semantics of Jinja+ as a set of rules (Rules B.1-B.70). A more comprehensive presentation of the Jinja+ extensions can be found in Appendix B, with the small-step semantics rule in Appendices B.1.1 and B.1.2.

Java-Interfaces and Abstract Classes. We extend the *abstract syntax* of a Jinja+ system to now also include abstract classes and Java-interfaces.[3] A *system* or a *program* is a triple of three lists: a list of *Java-interface* declarations, a list of *abstract class* declarations, and a list of *concrete class* declarations. Each declaration consists in the name of the Java-interface/class and the Java-interface/class itself. An *Java-interface* consists in a list of its direct superinterfaces (optionally), a list of its constant declarations, and a list of method signatures. A *constant declaration* consists of a type, a constant name, and a literal of the appropriate type (the value of the constant). A *method signature* consists of the method name, the formal parameter names and types, and the result type. An *abstract class* consists of the name of its direct superclass (optionally), a list of the implemented Java-interfaces (optionally), a list of field declarations, a list of method signatures (commonly called abstract methods), and a list of method declarations. A *field declaration* consists of a type and a field name. A *method declaration* consists in a method signature and in an expression (the method body). Note that there is no `return` statement, as a method body is an expression; the value of such an expression is returned by the method. A *concrete class* is defined as an abstract class without the list of method signatures, i.e., all methods contain the body. In the above definitions we assume that different syntactic categories (Java-interfaces, classes, constants, fields, and methods) have different names. We assume that Java-interfaces and abstract classes are provided by a compiler that, first, ensures that the policies expressed by these clauses are respected (e.g., a concrete class implementing an interface indeed implements all its methods) and then produces Jinja+ code without them. Regarding the Jinja+ rules (see Appendix B.1.2), we extend the convention used in [KN06] that symbols C and D denote (concrete or abstract) classes to denoting Java-interfaces, too. According to this convention, there are a few changes in the interpretation of the expressions and predicates of the small-step semantics rules:

- In the expression *new* C corresponding to the creation of a new object we assume that a compiler already enforced C to be a concrete class. Moreover, we require that the predicate

[3] We note that the concept of *Java-interface* follows the abstract type that is indeed used to specify interfaces in Java and it is not to be confused with the concept of interface used in the CVJ framework (see Section 2.2).

$P \vdash C$ *has-fields FDTs* used in the object creation rule (Rule B.16) collects information about the fields both in the class and in the Java-interface hierarchy.

- In the expression *Cast C* now C can be either a (concrete or abstract) class or an Java-interface. Therefore, we extend the meaning of the predicate $P \vdash D \preceq^* C$ (used in rules B.17, B.18, B.31, B.39 and B.40): it means D is a subclass of C if C is a class, while D implements C if C is an Java-interface.

- In the expression $e.F\{D\}$ (field access) D can now either be the class or the Java-interface where F is declared (in the latter case F is defined as a constant). In the expression $e.F\{D\} := e_2$ (field assignment) D can only be the (abstract or concrete) class where F is declared. However, the rules where these expressions are evaluated (Rule B.3 and B.4) remain unchanged.

- In the expression `try` e_1 `catch` $(C\ V)\ e_2$ we assume that a compiler already enforced C to be a concrete class (which must extend the class *Throwable*). Therefore, in rules B.39 and B.40, the predicate $P \vdash D \preceq^* C$ is indeed always interpreted as D is a subclass of C.

- The predicate $P \vdash C\ sees\ M :\ Ts \to T = (pns,\ body)\ in\ D$ used in the rule B.24 is supposed to look up for the method declaration of M in the class hierarchy and therefore now also in abstract classes but obviously not in the Java-interfaces (since they contain only method signatures).

The interpretation of the other expressions and rules remains the same as in [KN06].

Strings. We extend the language *Jinja* [KN06] with strings. We introduce a new value *litS* detoning a *string literal*, i.e., a quoted sequence of characters representing a string value within the source code. In the heap, we represent a string as a pair consisting of an array of the characters in the string literal and its length. The extension of the (small-step) semantic rules to deal with strings is quite straightforward and can be found in Figure B.6 of Appendix B.1.

2.2. Indistinguishability

We now define what it means for two Jinja+ systems to be indistinguishable by an environment interacting with them. As already mentioned in the introduction of the thesis, indistinguishability is a fundamental relation between systems which is interesting in its own right, for example, to define privacy properties, and to define simulation-based security, as we will present in the next section.

For this purpose, we first recall from [KTG12a] what we mean by an "interface" that a system uses/provides, how systems are composed, and environments. We then present the notion of computational indistinguishability which follows the spirit of definitions of (computational) indistinguishability in the cryptographic literature (see e.g., [Can00, Küs06]), but, of course, instead of interactive Turing machines, here we consider Jinja+ systems/programs.

Interfaces, Composition, and Environments. An *interface I* is defined like a (Jinja+) system but where (i) all private fields and private methods are dropped and (ii) method bodies as well as

static field initializers are dropped. A system S *implements* an interface I, written $S : I$, if I is a subinterface of the public interface of S, i.e., the interface obtained from S by dropping method bodies, initializers of static fields, private fields, and private methods. We say that *a system S uses an interface I*, written $I \vdash S$, if, besides its own classes, S uses at most classes/methods/fields declared in I. We write $I_0 \vdash S : I_1$ for $I_0 \vdash S$ and $S : I_1$. We also say that two interfaces are *disjoint* if the sets of class names declared in these interfaces are disjoint.

For two systems S and T we denote by $S \cdot T$ the *composition* of S and T which, formally, is the union of (declarations in) S and T. Clearly, for the composition to make sense, we require that there are no name clashes in the declarations of S and T. Of course, S may use classes/methods/fields provided in the public interface of T, and vice versa.

A system E is called an *environment* if it declares a distinct private static variable `result` of type `boolean` with initial value `false`. Given a system $S : I$, we call E an *I-environment for S* if there exists an interface I_E disjoint from I such that $I_E \vdash S : I$ and $I \vdash E : I_E$. Note that $E \cdot S$ is a complete program. The value of the variable `result` at the end of the run of $E \cdot S$ is called the *output* of the program $E \cdot S$; the output is `false` for infinite runs. If $E \cdot S$ is a deterministic program, we write $E \cdot S \rightsquigarrow \texttt{true}$ if the output of $E \cdot S$ is `true`. If $E \cdot S$ is a randomized program, we write $\mathsf{Prob}\{E \cdot S \rightsquigarrow \texttt{true}\}$ to denote the probability that the output of $E \cdot S$ is `true`.

We assume that all systems have access to a security parameter (modeled as a public static variable of a class SP). We denote by $P(\eta)$ a program P running with security parameter η.

To define computational equivalence and computational indistinguishability between (probabilistic) systems, we consider systems that run in (probabilistic) polynomial time in the security parameter. We omit the details of the runtime notions used in the CVJ framework here, but note that the runtimes of systems and environments are defined in such a way that their composition results in polynomially bounded programs.

Let P_1 and P_2 be (complete, possibly probabilistic) programs. We say that P_1 and P_2 are *computationally equivalent*, written $P_1 \equiv_{\mathsf{comp}} P_2$, if $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow \texttt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow \texttt{true}\}|$ is a negligible function in the security parameter η.[4]

Definition 2.1 (**Computational Indistinguishability** [KTG12a]). *Let S_1 and S_2 be probabilistic polynomially bounded systems. Then S_1 and S_2 are* computationally indistinguishable w.r.t. *I, written $S_1 \approx^I_{\mathsf{comp}} S_2$, if S_1 and S_2 use the same interface I and for every bounded Jinja+ I-environment E for S_1 (and hence, S_2) we have that $E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$.*

2.3. Simulatability and Universal Composition

We now define what it means for a system to realize another system, in the spirit of universal composability, a well-established approach in cryptography. Security is defined by an ideal system F (also called an ideal functionality), which, for instance, models ideal encryption, signatures, MACs, key exchange, or secure message transmission. A real system R (also called a real protocol) realizes F if there exists a simulator S such that no polynomially bounded

[4] As usual, a function f from the natural numbers to the real numbers is *negligible*, if for every $c > 0$ there exists η_0 such that $f(\eta) \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$.

environment can distinguish between R and $S \cdot F$. The simulator tries to make $S \cdot F$ look like R for the environment (see the subsequent sections for examples). More formally:

Definition 2.2 (**Strong Simulatability** (simplified) [KTG12a]). *Let F and R be probabilistic polynomially bounded systems which implement the same interface I_{out} and use the same interface I_E, except that in addition F may use some interface I_S provided by a simulator. Then, we say that R* realizes *F* w.r.t. *I_{out}, written $R \leq^{I_{out}} F$ or simply $R \leq F$, if there exists a probabilistic polynomially bounded system S (the simulator) such that $R \approx^{I_{out}}_{\mathsf{comp}} S \cdot F$.*

We notice that the relation $\leq$ is both reflexive and transitive (we refer the reader to [KTG12a] for the formal proof).

A main advantage of defining security of real systems by the realization relation $\leq$ is that systems can be analyzed and designed in a modular way: The following theorem implies that it suffices to prove security for the systems R_0 and R_1 separately in order to obtain security of the composed system $R_0 \cdot R_1$.

Theorem 2.1 (**Composition Theorem** (simplified) [KTG12a]). *Let I_0 and I_1 be disjoint interfaces and let R_0, F_0, R_1, and F_1 be probabilistic polynomially bounded systems such that $R_0 \leq^{I_0} F_0$ and $R_1 \leq^{I_1} F_1$. Then, $R_0 \cdot R_1 \leq^{I_0 \cup I_1} F_0 \cdot F_1$, where $I_0 \cup I_1$ is the union of the class, method, and field names declared in I_0 and I_1.*

2.4. I-Noninterference, Noninterference in Open Systems

The (standard) noninterference notion for confidentiality [GM82a] requires the absence of information flow from high to low variables within a program. Here, we define noninterference for a deterministic (Jinja+) program P with some static variables $\vec{x}$ of primitive types that are labeled as high. Also, some other static variables of primitive types are labeled as low. We say that $P[\vec{x}]$ is a *program with high variables $\vec{x}$* (and low variables). By $P[\vec{a}]$ we denote the program P where the high variables $\vec{x}$ are initialized with values $\vec{a}$ and the low variables are initialized as specified in P.

Now, noninterference for a deterministic program is defined as follows: Let $P[\vec{x}]$ be a program with high variables. Then, $P[\vec{x}]$ *has the noninterference property* if the following holds: for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), if $P[\vec{a}_1]$ and $P[\vec{a}_2]$ terminate, then, at the end of their runs, the values of the low variables are the same. Note that this defines *termination-insensitive* noninterference.

The above notion of noninterference deals with complete programs (closed systems). The systems to be analyzed are, however, often open: they interact with a network or use some libraries which are not necessarily trusted and, hence, are not part of the code to be analyzed; instead, they are considered as part of the environment with unspecified behavior. Therefore, the notion of noninterference is generalized to open systems as follows.

Definition 2.3 (**Noninterference in an open system** [KTG12a]). *Let I be an interface and let $S[\vec{x}]$ be a (not necessarily closed) deterministic system with a security parameter and high variables $\vec{x}$ such that $S : I$. Then, $S[\vec{x}]$ is I-*noninterferent *if for every deterministic I-environment E for $S[\vec{x}]$ and every security parameter η, noninterference holds for the system $E \cdot S[\vec{x}](\eta)$, where the variable `result` declared in E is considered to be the only low variable.*

Note that here neither E nor S are required to be polynomially bounded.

2.5. From I-Noninterference to Computational Indistinguishability

The central theorem that immediately follows from (the more general) results proven within the CVJ framework is the following.

Theorem 2.2 (**The CVJ Theorem** [KTG12a]). *Let I and J be disjoint interfaces. Let F, R, $S[\vec{x}]$ be systems such that*

i) $R \leq^J F$,

ii) not both $S[\vec{x}]$ and F (and hence, R) contain the method `main`*,*

iii) $S[\vec{x}] \cdot F$ *is deterministic,*

iv) $S[\vec{x}] \cdot F : I$ *(and hence,* $S[\vec{x}] \cdot R : I$*).*

Now, if $S[\vec{x}] \cdot F$ is I-noninterferent, then, for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), we have that $S[\vec{a}_1] \cdot R \approx^I_{\mathsf{comp}} S[\vec{a}_2] \cdot R$.

The intuition and the typical use of this theorem is that the cryptographic operations that S needs to perform are carried out using the system R (e.g., a cryptographic library). The theorem now says that to prove cryptographic privacy of the secret inputs ($\forall\ \vec{a}_1, \vec{a}_2$: $S[\vec{a}_1] \cdot R \approx^I_{\mathsf{comp}} S[\vec{a}_2] \cdot R$) it suffices to prove I-noninterference for $S[\vec{x}] \cdot F$, i.e., the system where R is replaced by the ideal counterpart F (the ideal cryptographic library). The ideal functionality F, which in our case will model cryptographic primitives in an ideal way, can typically be formulated without probabilistic operations and also the ideal primitives specified by F will be secure even in presence of unbounded adversaries.

Therefore, the system $S[\vec{x}] \cdot F$ can be analyzed by standard tools that a priori cannot deal with cryptography (probabilities and polynomially bounded adversaries).

As mentioned before, F relies on the interface $I_E \cup I_S$ (which, for example, might include an interface to a network library) provided by the environment and the simulator, respectively. This means that when checking noninterference for the system $S[\vec{x}] \cdot F$ the code implementing this library does not have to be analyzed. Being provided by the environment/simulator, it is considered completely untrusted and the security of $S[\vec{x}] \cdot F$ does not depend on it. In other words, $S[\vec{x}] \cdot F$ provides noninterference for all implementations of the interface. Similarly, R relies on the interface I_E provided by the environment. Hence, $S[\vec{x}] \cdot R$ enjoys computational indistinguishability for all implementations of I_E. This has two big advantages:

1. One obtains very strong security guarantees.

2. The code to be analyzed in order to establish noninterference/computational indistinguishability is kept small, considering the fact that libraries tend to be very big.

2.6. A Proof Technique for proving I-Noninterference

Tools for checking noninterference often consider only a single closed program. However, I-noninterference is a property of a potentially open system $S[\vec{x}]$, which is composed with an

arbitrary I-environment. Therefore, in [KTG12a] a *proof technique* has been developed which reduces the problem of checking I-noninterference to checking noninterference for a single (almost) closed system. More specifically, it was shown that to prove I-noninterference for a system $S[\vec{x}]$ with $I_E \vdash S : I$ it suffices to consider a single environment $\tilde{E}_{\vec{u}}^{I,I_E}$ (or $\tilde{E}_{\vec{u}}$, for short) only, which is parameterized by a sequence $\vec{u}$ of values. To keep $\tilde{E}_{\vec{u}}$ simple, the analysis technique assumes some restrictions on interfaces between $S[\vec{x}]$ and E. In particular, $S[\vec{x}]$ and E should interact only through primitive types, arrays, exceptions, and simple objects.

In Section 2.6.3, we reshape the proof technique to allow $S[\vec{x}]$ and E to communicate through strings, too.

2.6.1. Communication through Primitive Types Only

We now present how $\tilde{E}_{\vec{u}}$ is constructed in [KTG12a] when it interacts with a system S such that (1) method `main` is defined in S and (2) $I_E \vdash S$, for some interface I_E, where all methods are static, use primitive types only (for simplicity of presentation we will consider only the type `int`), and have empty `throws` clause. Moreover, the kind of E we consider are not allowed to call methods of S directly (formally, we require I to be $\emptyset$). However, since S can call methods of E, this is not an essential limitation.

For a finite sequence $\vec{u} = u_1, \ldots, u_n$ of values of type `int`, we denote by $\tilde{E}_{\vec{u}}^{I_E}$ the following system. First, $\tilde{E}_{\vec{u}}^{I_E}$ contains two static methods: `untrustedOutput` and `untrustedInput`, as specified in Figure 2.2. The method `untrustedInput` returns consecutive values of $\vec{u}$ and, after the last element of $\vec{u}$ has been returned, it returns 0. Note that the consecutive values returned by this method are hardwired in line 9 (determined by $\vec{u}$) and do not depend on any input to $\tilde{E}_{\vec{u}}^{I_E}$. The method `untrustedOutput`, depending on the values given by `untrustedInput()`, either ignores its argument or compares its value to the next integer it obtains, again, from `untrustedInput()` and stores the result of this comparison in the (low) variable `result`. The intuition is the following: `untrustedOutput` will get, as we will see below, all the data the environment gets from S. If the two variants of S (with different high values) behave differently, then there must be some point where the environment gets different data from the two systems in the corresponding runs. By choosing an appropriate $\vec{u}$ this can be detected by `untrustedOutput`, which will assign different values to `result`.

Finally, for every method declaration m in I_E, the system $\tilde{E}_{\vec{u}}^{I_E}$ contains the implementation of m as illustrated by the example in Figure 2.3. This completes the definition of $\tilde{E}_{\vec{u}}^{I_E}$. The next theorem states that, to prove I-noninterference, it is enough to only consider environments $\tilde{E}_{\vec{u}}^{I_E}$.

Theorem 2.3 ([KTG12a])**.** *Let I_E be an interface with only static methods of primitive (argument and return) types as introduced above. Let $S[\vec{x}]$ be a deterministic program such that* `main` *is defined in S and $I_E \vdash S$. Then, I-noninterference, for $I = \emptyset$, holds for $S[\vec{x}]$ if and only if for all sequences $\vec{u}$ noninterference holds for $\tilde{E}_{\vec{u}}^{I_E} \cdot S[\vec{x}]$.*

Automatic analysis tools, such as Joana [HS09b, GHM13], often ignore or can ignore specific values encoded in a program, such as an input sequence $\vec{u}$. Hence, such an analysis of $E_{\vec{u}}^{I_E} \cdot S[\vec{x}]$ implies noninterference for all sequences $\vec{u}$, and by the theorem, this implies I-noninterference for $S[\vec{x}]$.

```
1   class Node {
2       int value;
3       Node next;
4       Node(int v, Node n) { value = v;  next = n; }
5   }
6   private static Node list = null;
7   private static boolean listInitialized = false;
8   private static Node initialValue()
9       { return new Node(u1, new Node(u2, ...)); }
10  static public int untrustedInput() {
11      if (!listInitialized)
12          { list = initialValue(); listInitialized = true; }
13      if (list==null) return 0;
14      int tmp = list.value;
15      list = list.next;
16      return tmp;
17  }
18  static public void untrustedOutput(int x) {
19      if (untrustedInput()!=0) {
20          result = (x==untrustedInput());
21          abort();
22      }
23  }
```

Figure 2.2.: Implementation of `untrustedInput` and `untrustedOutput` in $\tilde{E}^{I_E}_{\vec{u}}$. We assume that class `Node` is not used anywhere else.

```
24  static public int foo(int x) {
25      untrustedOutput(FOO_ID);
26      untrustedOutput(x);
27      return untrustedInput();
28  }
```

Figure 2.3.: $\tilde{E}^{I_E}_{\vec{u}}$: the implementation of a method of I_E with the signature `static public int foo(int x)`, where `FOO_ID` is an integer constant serving as the identifier of this method (we assign different identifier to every method)

2.6.2. Communication through Arrays, Simple Objects, and Exceptions

The result of the Theorem 2.3 has also been extended in [KTG12a] to cover some cases where E and S exchange information non only through values of primitive types, but also arrays, simple objects (object whose fields are not static and either of primitive types or of type `byte[]`), and possible exceptions. Some conditions, however, have to be imposed on I_E and the program S. These conditions guarantee that, although references are exchanged between E and S, the communication resembles exchange of pure data. More precisely, the result stated below works for a restrict class of systems S. Let I_E be the *minimal* interface such that $I_E \vdash S$. On the one hand, we impose on I_E the following conditions:

E.1. Fields of classes in I_E are non-static and either of primitive types, or simple objects, or arrays of primitive types. Simple objects denote objects of classes defined in I_E.

E.2. Methods of classes in I_E are static. Their arguments and return values may be either of primitive types, or simple objects, or arrays of primitive types.

E.3. Exceptions thrown by methods of classes in I_E are either standard system exceptions or exceptions defined in I_E.

On the other hand, we impose on S the following condtions:

S.1. Whenever a simple object or an array (i.e. the reference to a simple object or to an array, respectively) is passed to the environment, this reference is not used by S afterwards. This property can be easily guaranteed by a syntactical restriction to pass only fresh copies of these reference to the environment.

S.2. Whenever a method of I_E returns a reference r the system S is only allowed to immediately produce a fresh copy of r and not to use r afterwards.

S.3. For every try-catch statement in S of the form

```
try { ...  } catch (C  r) { B }
```

if C or a subclass of C is listed in the `throws` clause of some method in I_E (and thus this statement may potentially catch an exception thrown by E), then again S is only allowed to immediately produce a fresh copy of r and not to use r afterwards.

For such programs, in [KTG12a] it has also been constructed a fixed $\tilde{E}^{I_E}_{\vec{u}}$ such that it is enough to consider only this system (for all $\vec{u}$). This system consists of the static methods `untrustedInput` and `untrustedOutput` as defined above and, for every class C of I_E, the declaration of C with the implementation of (static) methods as illustrated by the example given in Figure 2.4.

This example illustrates the case when an array is returned. When an object of class D is to be returned, then a fresh object of this class is created and the values of its fields that are specified in I_E are filled using `untrustedInput`, as for the array in the example. The following theorem is a generalisation of Theorem 2.3 to the richer family of programs considered in this section.

```
1  class T extends Exception {};
2  class T1 extends T {};
3  class T2 extends T {};
4  static public byte[] foo(int x, byte t[]) throws T {
5      // consume the input:
6      untrustedOutput(FOO_ID);
7      untrustedOutput(x);
8      untrustedOutput(t.length);
9      for( int i=0; i<t.length; ++i )
10         untrustedOutput(t[i]);
11     // decide whether to throw an exception:
12     if(untrustedInput()==0) throw new T();
13     if(untrustedInput()==0) throw new T1();
14     if(untrustedInput()==0) throw new T2();
15     // determine the array to return:
16     int len = untrustedInput();
17     if (len<0) return null;
18     byte[] result = new byte[len];
19     for( int i=0; i<len; ++i)
20         result[i] = (byte) untrustedInput();
21     return result;
22 }
```

Figure 2.4.: $\tilde{E}_{\vec{u}}$: the implementation of a method with the signature `static public byte[] foo(int x, byte[] t) throws T`, where $T1$ and $T2$ are subclasses of T (there could be an arbitrary number of them) which are classes in I_E.

Theorem 2.4 ([KTG12a]). *Let I_E be an interface with only static methods of (arguments and return) primitive types,* arrays, simple objects, and exceptions *with the restrictions E.1-E.3 introduced above. Let S be a system with the restrictions S.1-S.3 introduced above and with high and low variables such that* `main` *is defined in S and $I_E \vdash S$. Then, I-noninterference, for $I = \emptyset$, holds for S if and only if for all sequences $\vec{u}$ as defined in Section 2.6.1 noninterference holds for $\tilde{E}_u^{I_E} \cdot S$.*

2.6.3. Communication through Strings

In this section, we further extend the proof technique for proving I-noninterference to deal with the exchange of strings, too.

Although strings are commonly used to exchange data among components of a program, the communication between the system and the environment through string references would introduce subtle changes in Theorems 2.3 and 2.4 and, technically, invalidate the main assumption these results are based on, i.e., the separation between the state of the system and the state of the environment. Therefore, we need to extend the construction of $\tilde{E}_{\vec{u}}^{I_E}$ with two other methods handling the exchange of string references, too. Relying on the fact that Java (and Jinja+) strings

are *immutable* objects, i.e., their internal state remains unchanged after their creation, we relax the state separation assumption and adapt the proof in a non-trivial way to work with the new (relaxed) assumption. Based on this premise, we then restate the proof technique result for proving I-noninterference taking into account string references, too.

We model the exchange of references of type `String` by extending the environment $\tilde{E}_{\vec{u}}^{I_E}$ presented above with the two static methods presented in Figure 2.5: The method `untrustedOutputString` gets all string references passed by S to $\tilde{E}_{\vec{u}}^{I_E}$, whereas the method `untrustedInputString` determines which string reference the environment passes on to S. We notice that these methods rely on `untrustedOutput` and `untrustedInput` defined for primitive types and, hence, the sequence $\vec{u} = u_1, \ldots, u_n$ remains the same as defined in Section 2.6.1. More specifically, the method `untrustedInputString`, depending on `untrustedInput`, either returns a new string (built calling, character by character, `untrustedInput`) or it returns one of the strings previously exchanged between the systems. In particular, each new string returned to S is previously added to a list `stringList` containing all the strings exchanged between the two systems so far.

The method `untrustedOutputString`, besides adding its string argument to `stringList`, forwards to `untrustedOutput` its length, then each one of its characters, and, finally, the result of the comparison for reference equality between its string argument and each element in `stringList`.

The intuition is the following: if two instances of S, say $S[\vec{a}_1]$ and $S[\vec{a}_2]$, which the environment tries to distinguish behave differently, then there must be a point in the two runs where the environment gets either two strings with different values or two strings whose references were already been exchanged before (more precisely, at least one of them), but in different points of the two runs. In the former case, there must be that either the length or at least one character of the two strings is different. In the latter case, there must exist two elements at the same position in `stringList`, whose comparison with the two strings the environment received give different results. In any case, `untrustedOutput` will be invoked with different values, say the value x takes at this point are b_1 and b_2, respectively. As for primitive types, by choosing an appropriate $\vec{u}$, this can be detected by `untrustedOutput`: $\vec{u}$ should be defined in such a way that the method `untrustedInput` returns 1 at this point and that the value `untrustedInput` returns next is equals to b_1, say (b_2 would also work). Then, in the run of the environment with $S[\vec{a}_1]$ at the variable `result` will be assigned 1, while in the run with $S[\vec{a}_2]$ at the variable `result` will be assigned 0. Hence, the environment successfully distinguished $S[\vec{a}_1]$ and $S[\vec{a}_2]$.

Methods of I_E dealing with strings are implemented as the corresponding methods dealing with primitive values: the arguments are forwarded to `untrustedOutputString`, whereas `untrustedInputString` determines the returned value. The restrictions $E.1$ and $E.2$ presented in Section 2.6.2 on I_E must be updated to also include the data type string: fields of classes in I_E, as well as their methods' arguments and return value, might be of type string too. In the restriction $S.2$ also reported in Section 2.6.2, we do not require references of type strings returned to S to be immediately copied and not used afterwards.

The system $\tilde{E}_{\vec{u}}^{I_E}$, introduced in Section 2.6.1, firstly extended in Section 2.6.2, and further extended here to allow for the communication through strings, consists of the four static methods `untrustedInput`, `untrustedOutput`, `untrustedInputString`, and `untrustedOutputString`,

```
1  class NodeList {
2    public class Node {
3      public String entry;
4      public Node next;
5      public Node(String entry) {
6        this.entry = entry;
7        this.next=null;
8      }
9      public Node head, last;
10     public void add(String entry) {
11       Node newEntry=new Node(entry);
12       if (head==null) head=last=newEntry;
13       else {last.next=newEntry; last=newEntry;}
14     }
15 }
16 private static NodeList stringList = null;
17 static public String untrustedInputString() {
18   int choice = untrustedInput();
19   if(choice==1){
20     int l=untrustedInput(); String s="";
21     for(int i=0; i<l; i++)
22       s += (char) untrustedInput();
23     if(stringList==null) stringList = new NodeList();
24     stringList.add(s);
25     return s;
26   } else if(choice==2){
27     if(stringList==null) return "";
28     for(NodeList.Node node=stringList.head; node!=null;
29      node=node.next)
30       if(untrustedInput()==1) return node.entry;
31   }
32   return "";
33 }
34 static void untrustedOutputString(String s) {
35   if(stringList==null)
36       stringList = new NodeList();
37   // values comparison
38   untrustedOutput(s.length());
39   for (int i = 0; i < s.length(); i++)
40       untrustedOutput(s.charAt(i));
41   // references comparison
42   for(NodeList.Node node=stringList.head; node!=null;
43     node=node.next)
44      untrustedOutput(s==node.entry ? 1:0);
45   stringList.add(s);
46 }
```

Figure 2.5.: Implementation of `untrustedInputString` and `untrustedOutputString` in $\tilde{E}^{I_E}_{\vec{u}}$. We notice that these methods rely on methods `untrustedInput` and `untrustedOutput` presented in Figure 2.2. We assume that class `NodeList` is not used anywhere else.

```
1  class T extends Exception {};
2  class T1 extends T {};
3  class T2 extends T {};
4  class D { int y; String w; byte[] v; }
5  class Foo {
6   static public String[] fooExt(String s, D obj) throws T {
7    // consume the input:
8    untrustedOutput(0x100); // fooExt id
9    untrustedOutputString(s);
10   untrustedOutput(obj.hashCode());
11   untrustedOutput(obj.y);
12   untrustedOutputString(obj.w);
13   untrustedOutput(obj.v.length);
14   for (int i = 0; i<obj.v.length; i++)
15     untrustedOutput(obj.v[i]);
16   // decide whether to throw an exception:
17   if (untrustedInput()==0) throw new T();
18   if (untrustedInput()==0) throw new T1();
19   if (untrustedInput()==0) throw new T2();
20   // determine the object to return:
21   int length=untrustedInput();
22   String[] retStr = new String[length];
23   for(int i=0; i<length; ++i)
24    retStr[i]=untrustedInputString();
25   return retStr;
26  }
27 }
```

Figure 2.6.: $\tilde{E}^{I_E}_{\vec{u}}$: the implementation of the class `Foo` in I_E with only a method whose signature is `static public String[] fooExt(String s, D obj) throws T`, and where `T`, `T1`, `T2`, and `D` are classes in I_E, too. We assume that in Jinja+ (as in Java) each object has an unique identifier provided by the method `hashCode`.

and, for every class C of I_E, the declaration of C with the implementation of (static) methods as illustrated by the example given in Figure 2.6.

We now state the novel result of this chapter, which is a generalization of Theorem 2.4 to the case where the communication occurs, not only through primitive types, arrays, simple objects, and exceptions, but also through strings.

Theorem 2.5. *Let I_E be an interface with only static methods of (arguments and return) primitive types, arrays, simple objects, exceptions, and* strings *with the restrictions E.1-E.3 introduced in Section 2.6.2 and extended above to deal with strings. Let S be a system with the restrictions S.1-S.3 introduced in Section 2.6.2 and extended above to deal with strings and with high and low variables such that* `main` *is defined in S and $I_E \vdash S$. Then, I-noninterference, for $I = \emptyset$, holds for S if and only if for all sequences $\vec{u}$ as defined in Section 2.6.1 noninterference holds for $\tilde{E}^{I_E}_{\vec{u}} \cdot S$.*

The proof of this theorem, as long with the state separation assumption and all the other lemmas to demonstrate it, is stated in Appendix F.1.

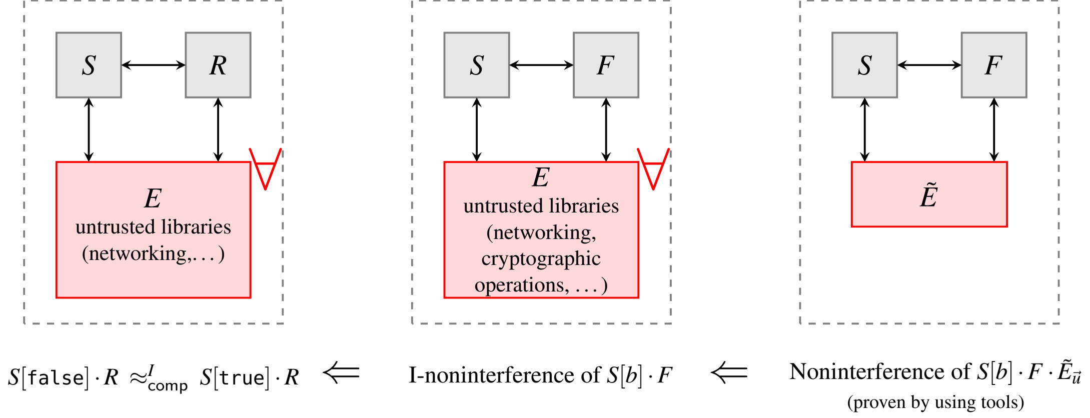

Figure 2.7.: The CVJ Framework: from Noninterference to Computational Indistinguishability.

Now, Figure 2.7 shows the complete picture of the CVJ framework where, by Theorems 2.5 and 2.2, tools for proving noninterference can be used to prove cryptographic privacy of a system S that uses some cryptographic operations (real cryptographic functionalities) R and is connected to untrusted libraries subsumed by the environment E (see the left-most system of the picture). For simplicity of presentation, in Figure 2.7 we assume S to have only one high boolean variable. By these theorems, to prove cryptographic privacy of this system (that is, $S[\mathtt{false}] \cdot R \approx^{I}_{\mathsf{comp}} S[\mathtt{true}] \cdot R$), it is enough to show (classical) noninterference of the system $S[b] \cdot F \cdot \tilde{E}$, where F are the ideal functionalities corresponding to R and $\tilde{E}_{\vec{u}}$ is the specific environment given in Theorem 2.5. Note that F often uses cryptographic libraries. However, the cryptographic guarantees are established by F itself (e.g., F might call an encryption function, but with $0^{|m|}$ instead of the actual message m, and hence, by the definition of F and independently of the encryption function, the ciphertext will not reveal information about m, except for the length of m). Therefore, the cryptographic libraries that F uses can be untrusted, and hence, provided by the environment. Conversely, the cryptographic library implemented in R is supposed to realize F. So, this library cannot be subsumed by the environment.

3. Extending the CVJ Framework to Java Concurrency

One of the main features of the Java programming language not covered by Jinja+ is Java concurrency. In this chapter, we therefore extend the CVJ framework to deal with concurrency. That is, we extend the Jinja+ language (see Section 2.1) to model concurrent Java programs and then we make the CVJ framework applicable to this extended language. Java concurrency revolves around threads, parallel strands of execution with shared memory, and offers different types of synchronization between them. More specifically, we extend the sequential Jinja+ language to model thread creation and only one type of synchronization mechanism, i.e, synchronized blocks.[5] We refer to this language as SyncJinja+.

To define runs of SyncJinja+ programs, we first extend the (single-threaded) Jinja+ semantics with reduction rules for the language constructs modeling thread spawning and synchronized blocks. We then propose a multi-threaded semantics describing the run of multi-threaded programs under a scheduler (defined as a separate, single-threaded Jinja+ program) arranging the interleaving between the spawned threads. In a similar way as it has been done in [Loc12], each multi-threaded reduction rule relies on the (extended) set of the single-threaded Jinja+ reduction rules describing the execution of each thread. Interaction between single-threaded and multi-threaded semantics only happens through designated actions communicating either the spawning of a new thread or the beginning/ending of a synchronized block.

An adversary running concurrently with an honest program can measure the runtime of a program. Therefore proving cryptographic guarantees based on standard cryptographic assumption would not be possible anymore: cryptographic security games, e.g., for defining IND-CCA2 security, are inherently sequential, and hence, they do not capture timing attacks. SyncJinja+ therefore allows one to declare parts of code to be executed *atomically*, i.e., without interleaving neither of any other thread nor of the scheduler. We denote such code *atomic* and refer to classes/systems/programs containing only atomic code as atomic classes/systems/programs.

Based on the definition of a run of a multi-threaded program (which may contain atomic code, typically some cryptography related code), we then restate all the definitions and the results of the CVJ framework presented in Sections 2.2-2.5 of Chapter 2.

The notion of *computational indistinguishability* which, in the spirit of the definitions of indistinguishability in the cryptographic literature (see, e.g., [Can00, Küs06, HUMQ09]) defines what it means for two SyncJinja+ systems to be indistinguishable from the point of view of a multi-threaded adversary interacting with them. We also formalize the notion of *strong simulatability* which, in the spirit of the simulation-based approach (see, again, [Can00, PW01, Küs06, KT13]), defines what it means for a SyncJinja+ system to realize another SyncJinja+ system.

We then state and prove the *composition theorem* which informally says that if a SyncJinja+

[5] Modeling the other three types of synchronization mechanisms offered by Java, namely the `wait-notify` mechanism, thread joining, and interrupts, would in principle be possible (it has been done, for example, in [Loc12]), but would lead to a cumbersome representation of the multi-threaded state and hence of the multi-threaded semantic rules.

system R_0 realizes a SyncJinja+ system F_0 and a SyncJinja+ system R_1 realizes a SyncJinja+ system F_1, then the composed real system $R_0 \cdot R_1$ realizes the composed ideal system $F_0 \cdot F_1$. This theorem allows us to replace ideal functionalities, formulated as SyncJinja+ systems, by their realization, also formulated as SyncJinja+ systems, in more complex, composed systems. Thanks to the composition theorem, we can show that if two SyncJinja+ systems that both use an ideal functionality (e.g., an ideal cryptographic operation) are perfectly indistinguishable, then these systems are computationally indistinguishable when the ideal functionality is replaced by its realization, where perfect indistinguishability is defined just as computational indistinguishability but for deterministic SyncJinja+ program and w.r.t. unbounded adversaries.

Finally, we propose a noninterference definition which extends the notion of (termination-insensitive) noninterference for confidentiality [GM82a], requiring the absence of information flowing from high to low variables, to multi-threaded SyncJinja+ systems. As in the single-threaded setting, proving that this definition is equivalent to the definition of perfect indistinguishability allows us to assert that noninterference of a SyncJinja+ system which uses ideal cryptographic functionalities implies computational indistinguishability of the same system using their realizations.

An interesting question arising when extending the CVJ framework to concurrency is whether and how a multi-threaded adversary is able to distinguish two single-threaded Jinja+ systems S_1 and S_2 which are indistinguishable w.r.t. a single-threaded adversary. We show that, under the assumption that the two systems S_1 and S_2 are considered to be atomic, these two systems are computational indistinguishable also from the point of view of a multi-threaded adversary. Thanks to this result, we can also prove that if a single-threaded system R realizes a single-threaded system F, then this realization carries over also to multi-threaded adversaries, as long as the assumption of considering R and F atomic is maintained.

3.1. Concurrency in Java

In this section, we introduce the main concurrency features of Java. As reported in [Loc12], Java concurrency revolves around threads, i.e., parallel strands of execution with shared memory. A program controls a thread through its associated object of (a subclass of) class *Thread*. To spawn a new thread, it allocates a new object of (subclass of) class *Thread* and invokes the start method. The new thread will then execute the `run` method of the object in parallel with all the other threads. Each thread can be only started once, otherwise an `IllegalThreadState` exception is thrown. The thread terminates when `run` terminates, either normally or abruptly due to an exception.

Java offers four types of *synchronization* between threads: (1) locks, (2) wait sets, (3) joining, and (4) interrupts. Every object has an associated monitor with a lock and a wait set.

(1) Locks are *mutually exclusive*, i.e., at most one thread can hold the lock at a time, but *re-entrant*, i.e., each thread can acquire a lock multiple times. For locking, Java uses `synchronized` blocks that take a reference to an object. A thread must acquire the lock before executing the lock's body, and releases the lock afterwards. If another thread already holds the lock, the executing thread must wait until the other thread has released it. That is, `synchronized` blocks on the same object never execute in parallel.

(2) To avoid busy waiting, a thread can suspend ifself to the wait set of an object by calling

the object's method `wait` declared in the class *Object*. To enter the wait set, the thread must have locked the object's monitor and must not be interrupted (see below for the explanation of interrupts): if the thread is interrupted, an `InterruptedException` is thrown. If successful, the call also releases the monitor's lock completely. The thread remains in the wait set until (i) another thread interrupts or notifies it, or (ii) it wakes up spuriously.[6] After having been removed from the waiting set, the thread reacquires the lock on the monitor before its execution proceeds normally (again, in case of interruption, an `InterruptedException` is thrown). The methods `notify` and `notifyAll` remove one (unspecified) or all threads from the wait set of the call's receiver object. As in the case of the `wait` method, the calling thread must hold the lock on the monitor. Thus, the notified thread continues its execution only after the notifying thread has released the lock.

(3) When a thread calls `join` on another thread, it blocks until (i) the thread that the receiver object identifies has terminated, (ii) another thread interrupts the joining thread (see below for more explanation), or/and (iii) an optionally-specified (as parameter) amount of time has elapsed. In the second case, an `InterruptException` is thrown; otherwise, it returns normally.

(4) Interruption provides asynchronous communication between threads. Calling the `interrupt` method of a thread sets its interrupt status. If the interrupted thread is waiting or joining, it aborts the operation throwing an `InterruptException` and clearing its interrupt status. Otherwise, interruption has no immediate effect on the interrupted thread. However, class *Thread* implements two methods to observe the interrupt status: the method `isInterrupted` returns the interrupt status of the receiver object's thread, while the method `interrupted` returns the interrupt status of the receiver object's thread and, in addition, resets its interrupt status.

All the other (not deprecated) constructs provided by Java to support concurrency are either recommendations for the schedulers (the `yield` and `sleep` methods) or they can be expressed with the four kinds of synchronization mechanisms introduced above. For instance, Java allows one to declare `volatile` variables: when a thread reads from a volatile variable, it synchronizes the access to this variable with all other threads sharing it. This can also be easily implemented by protecting all accesses to shared data via locks. Other ways of synchronization, such as invocation of the methods `stop` or `suspend` of objects of class *Thread* (or any subclass thereof), are deprecated in the latest versions of the language.

Java also specifies how shared memory behaves under concurrent access, which is known as the *Java Memory Model* (JMM). The Java Memory Model is based on *Sequential Consistency* (SC) [Lam97]: There is a global notion of time, only one thread executes at a time, and every write to a memory location immediately becomes visible to all threads. However, for efficiency reasons, the JMM relaxes the SC memory model to allow for local caches and code optimization: if a variable is shared between two threads and one thread performs a write operation on this variable, this operation might still be in the processor's local cache when the second thread performs a read operation from the main memory. Similarly, compiler optimizations might reorder the independent statements in each thread.

Nevertheless, the JMM provides the intuitive SC semantics under additional assumptions called

[6] The Java Virtual Machine allows spurious wake-ups to happen i.e., threads are woken up even if they are neither interrupted nor notified by other threads. It is therefore a best coding practice to always call the `wait` method inside a loop whose guard is the condition for which the thread is waiting.

data-race freedom (DRF) guarantees. A data-race occurs if two accesses to the same location are conflicting, that is, if (i) they originate from different threads, (ii) at least one is a write, and (iii) the location is not explicitly declared as `volatile` (or not protected by the same lock among all threads). If the program contains no data races, the JMM ensures that it behaves like under SC.

3.2. SyncJinja+ systems

We extend the semantics introduced in the Section 2.1 in order to be able to model (runs of) multi-threaded programs. In particular, we extend the Jinja+ language to model thread creation and locks, i.e., synchronized blocks. As already mentioned in the introduction of this Chapter, modeling also the `wait-notify` mechanism, thread joining, and thread interruption would in principle be possible (it has been done, for example, in [Loc12]) but would lead to a cumbersome representation of the multi-threaded state and hence of the multi-threaded semantic rules. We leave the extension to the other synchronization mechanisms as future work.

In multi-threaded systems, a *thread* is a subset of the set of class declarations $\mathbb{K}$ of a program, whose strand of execution can take place in parallel with other threads sharing the heap. We define a *thread pool* Π as a map from thread identifiers $ThreadID$ to pairs of expressions and stores, $\Pi\colon ThreadID \to Expr \times Loc$, where thread identifiers are defined as bit strings. Moreover, we need to extend the definition of a *state* as a triple of the set of the stores inside Π, namely $\{l_{tID_i} | tID_i \mapsto \langle e_{tID_i}, l_{tID_i}\rangle \in \Pi\}$, the shared *heap* and a *lock*. A *lock* is a map from references (locations) to pair of thread identifiers and integers, $lock\colon Loc \to \mathtt{dom}(\Pi) \times Int$. We define their range as a pair of thread identifiers and integers in order to fully model Java locks, which are mutually exclusive and re-entrant: at most one thread may hold a lock at a time, but the same lock may be acquired multiple times by the same thread.

In multi-threaded systems a *configuration* is therefore defined as a triple $q = \langle \Pi, h, lock\rangle$, and a *state* as $s = (L, h, lock)$, where $L = \{l_{tID_0}, \dots, l_{tID_n}\}$ with $n = |\Pi|$.

To define the run of a multi-threaded system, we extend the single-step semantics of Jinja+ presented in Section 2.1 and we add a multi-threaded single-step semantics to model the interleaving of individual threads (in a similar way as in [Loc12]), as well as the run of the scheduler arranging the execution of them. That is, we now have two different kinds of semantic rules: single-threaded semantic rules modeling the execution of each thread (and of the scheduler, too) and multi-threaded semantic rules modeling the execution of the overall system.

3.2.1. Single-Threaded Semantics of SyncJinja+

The single-threaded semantics of SyncJinja+ is an extension of the semantics of Jinja+ [KTG12a]. The idea is that each rule of this semantics (Rules B.1-B.87) is provided with an action $\mathscr{A} ::= \emptyset \mid Spawn(a) \mid Lock(a) \mid Unlock(a)$ that triggers a specific rule of the multi-threaded one (Rules B.88-B.95). Depending on the single-threaded rule executed, the triggered multi-threaded rule changes the expression and the local state of the thread performing the step of execution, and the global heap. Depending on the action that the single-threaded rule transmits to the multi-threaded one, a new thread is spawned or a lock is either acquired or released.

In Figure B.7, Rules B.77-B.87 contain the language constructs which produce the thread actions: `start`(`addr` a) spawns a new thread starting from the class pointed by the location a (Rule B.80), `sync`(`addr` a)$\{e\}$ acquires a lock on the location a (Rule B.81), and `insync`(`addr` a)$\{e\}$ keeps the lock on the location a until the expression e is reduced to a value v, then it releases the lock (Rule B.82). The other rules in Figure B.7 are subexpression reduction rules or exceptional expression reduction rules of these constructors.

3.2.2. Multi-Threaded Semantics of SyncJinja+

The multi-threaded semantics of SyncJinja+ is given as a set of rules $P \vdash \langle \Pi, h, lock \rangle \stackrel{tID_k}{\Longrightarrow} \langle \Pi', h', lock' \rangle$. In this rule, P is a program in the context of which the evaluation is carried out. Such a rule says that, given a program P, it exists a thread with tID_k as identifier such that $\{tID_k \mapsto \langle e, l \rangle\} \in \Pi$ and whose expression e can be reduced in one step to e' changing: *(1)* the local state from l to l' such that $\Pi' = \Pi \setminus \{tID_k \mapsto \langle e, l \rangle\} \cup \{tID_k \mapsto \langle e', l' \rangle\}$, *(2)* the heap h and the lock map $lock$ to h' and $lock'$, respectively.

However, Π usually contains several threads which can perform a step of execution. Hence, we need to define a scheduler that decides within which thread the next computational step takes place.

Definition 3.1. *We define a* scheduler *as a single-threaded Jinja+ program $\mathscr{S}$ which declares two public static variables: (1)* `actThreads` *containing the list of the active threads' identifiers at each scheduler invocation and (2)* `nextThread` *containing the identifier of the thread (in the program running under this scheduler) which is going to execute the next step.*

We note that, since $\mathscr{S}$ is a single-threaded Jinja+ system, its code does not contain any construct introduced in SyncJinja+ (see Figure B.7) i.e., `start`, `sync`, and `insync`. That is, the run of the scheduler takes place by the application of rules B.1-B.76.

The triple $\langle \Pi, h, lock \rangle$ defining a configuration of a multi-threaded system has to be extended to express the configuration of the scheduler under which the system runs. That is, we add the configuration of the scheduler $\mathscr{S}$, $\langle e, (l, h) \rangle$, as a subscript of the configuration of the multi-threaded system: $\langle \Pi, h, lock \rangle_{\langle e,(l,h) \rangle_{\mathscr{S}}}$. In what follows (and also in the semantic rules), whenever we want to make explicit that a configuration, an expression, a store, a heap, or a variable belongs to the execution of the scheduler, we add $\mathscr{S}$ as subscript of them. Since every multi-threaded program P can be executed only under a scheduler $\mathscr{S}$, we write $P_{\mathscr{S}}$ to indicate a *complete* program, where $\mathscr{S}$ is the scheduler that, at each step of the computation of P, establishes which thread is going to execute the next step (i.e., it establishes the value of the variable `nextThread`).

Since not every thread could be able to progress (for instance, a thread could be waiting for acquiring a lock owned by another one), we need to define the set of the threads which, at each a point of the run, can perform a step of execution.

Definition 3.2. *Given a configuration $q = \langle \Pi, h, lock \rangle_{\langle e,(l,h) \rangle_{\mathscr{S}}}$, we define an* active thread *as a thread tID_k in Π which is able to progress, namely there exists a SyncJinja+ rule which can be*

applied to the configuration q such that $q \overset{tID_k}{\Longrightarrow}_\ell q'$ *for some* tID_k *and* $q' = \langle \Pi', h', lock' \rangle_{\langle e,(l,h)\rangle_{\mathscr{S}}}$. *The subset of the active threads in q is then defined as:*

$$\mathsf{Act}(\langle \Pi, h, lock \rangle) = \{tID_k \in \mathtt{dom}(\Pi) | \exists q' \ s.t. \ q \overset{tID_k}{\Longrightarrow}_\ell q'\}$$

Since in *SyncJinja+* we do not have the data type *set* but only the data type *list*, without loss of generality, we assume that in the SyncJinja+ rules we can assign the set of bit strings $\mathsf{Act}(\langle \Pi, h, lock \rangle)$ to the variable `actThreads`, whose type is a list of bit strings. That is, the order of the elements in the list `actThreads` is irrelevant.

Before explaining the multi-threaded semantic rules, we need to discuss another restriction: Some part of the code of the multi-threaded program *P* must have to be executed atomically by one thread, i.e., without the interleaving neither of other threads nor of the scheduler: That is, the code implementing the cryptographic operations cannot be executed in parallel with the code of other threads, since their security is defined by a *sequential* game between the encryption scheme and the adversary, as, for instance, shown in Appendix A.1 for IND-CCA2-secure public key and symmetric encryption schemes [BDPR98], and in Appendix A.2 for EUF-CMA-secure digital signatures schemes [GMR88].

We have, therefore, to define a set of classes which enclose a specific part of the code that has to be executed sequentially.

Definition 3.3. *Given the set of class declarations* $\mathbb{K}$ *of a program, we define a set of* atomic classes $\mathbb{A}$ *as a subset of the class declarations* $\mathbb{K}$ *(*$\mathbb{A} \subseteq \mathbb{K}$*) which contain only Jinja+ statements and whose code must be executed atomically by the current thread, i.e., without any interleaving neither of other threads nor of the scheduler. In the same way we define S a system* atomic *if it only contains classes in* $\mathbb{A}$.

We note that atomic classes contain no `start`, no `sync`, and no `insync` and, therefore, during their execution the thread remains always *active*.

In Figure B.7, rules B.88-B.95 define the multi-threaded semantics of SyncJinja+.

Rule B.88 manages the execution of the scheduler $\mathscr{S}$: until the value of the variable $\mathtt{nextThread}_{\mathscr{S}}$ is *null*, we keep on reducing $e_{\mathscr{S}}$ in a store where the value of the variable $\mathtt{actThreads}_{\mathscr{S}}$ is $\mathsf{Act}(\{\Pi, h, lock\})$, i.e., the list of the threads which can progress. Whenever the scheduler sets the value of $\mathtt{nextThread}_{\mathscr{S}}$ to the identifier of a thread (among those which can progress), this thread performs the step by the application of one of the other rules.

If this step is performed by an expression which belongs to an atomic class, then the value of $\mathtt{nextThread}_{\mathscr{S}}$ remains unchanged (*Rule B.90*). That is, in the next step we keep on reducing the expression of this thread. As already mentioned above, if $\ell \in \mathbb{A}$ the sub expression which is going to be reduced does not contain any construct introduced in SyncJinja+, i.e., the thread can always progress (unless, obviously, the primitive `abort` is executed).

Rules B.89-B.94 define the execution of a thread which is not executing a code in an atomic class. Depending on the thread action $\mathscr{A} ::= \emptyset \mid Spawn(a) \mid Lock(a) \mid Unlock(a)$, the specific rule is executed. We notice that, after the execution of each rule, the value of the variable $\mathtt{nextThread}_{\mathscr{S}}$ is set again to *null*, i.e., the scheduler executes the next step by the application of rule B.88.

If, during the execution of a thread, we have to reduce the primitive `abort`, then all the threads and the scheduler must abort their execution, leaving their states unchanged. That is, after the execution of *Rule B.95*, both the thread which performed the step and the scheduler cannot progress leading the whole program to halt.

3.2.3. Run of a SyncJinja+ program

We can now define a run of a SyncJinja+ program P under a scheduler $\mathscr{S}$.

Definition 3.4 (Run under a scheduler). *A* run *of a (possibly) randomized multi-threaded program P under a scheduler $\mathscr{S}$ is a sequence of configurations obtained using both the single-threaded and the multi-threaded semantics of SyncJinja+ (Rules B.1-B.95) from the initial configuration q_0 of the form*

$$q_0 = \langle \{tID_0 \mapsto \langle e_0, l_0 \rangle\}, h_0, lock_0 \rangle_{\langle e_0, (l_0, h_0) \rangle_{\mathscr{S}}},$$

where:

- *Concerning the program P, tID_0 is the bit string identifying the (only) thread spawned at the beginning, $e_0 = C.\mathtt{main}$, for C being the (unique) class where* `main` *is defined in P, $h_0 = lock_0 = \emptyset$ are respectively the empty heap and the empty lock map, and l_0 is the store mapping the static (global) variables to their initial values (if the initial value for a static variable is not specified in the program, the default initial value for its type is used).*
- *Concerning the scheduler $\mathscr{S}$, $e_0^{\mathscr{S}} = D.\mathtt{main}$, for D being the (unique) class where* `main` *is defined in $\mathscr{S}$, $h_0^{\mathscr{S}}$ is the empty heap of $\mathscr{S}$, and $l_0^{\mathscr{S}}$ is the store of $\mathscr{S}$ mapping* `actThreads` *to tID_0 (the identifier of the only thread present at the beginning),* `nextThread` *to null and all the other static (global) variables to their initial values.*

A randomized program induces a distribution of runs in the obvious way. Formally, such a program is a random variable from the set $\{0,1\}^\omega$ of infinite bit strings into the set of runs of deterministic programs, with the usual probability space over $\{0,1\}^\omega$, where one infinite bit string determines the outcome of randomBit(), and hence, induces exactly one run.

Based on the definition of the initial configuration q_0 and on the multi-threaded semantics defined in Figure B.7, we notice that:

- At the beginning of the computation the scheduler $\mathscr{S}$ starts running and it runs until it updates the variable `nextThread` to the identifier of a thread in P which is going to execute the next step (the first time `nextThread` $= tID_0$ since tID_0 is the only thread).
- After the thread executed the step, if this step has not been performed within an atomic class ($\ell \notin \mathbb{A}$), the scheduler reacquires the control of the run to compute the identifier of the thread which is going to execute the following step.
- On the contrary, if this step is performed within an atomic class ($\ell \in \mathbb{A}$), then the current thread continues its execution without being interrupted neither by another thread nor by the scheduler,

until the first step outside the atomic class is performed (i.e, the first instruction after the return from a method of this class), when the scheduler is again allowed to take the control of the execution in order to establish which thread is going to progress.

That is, unless some code belonging to atomic classes is executed, each step of the run of a multi-threaded SyncJinja+ system S is interleaved with several steps of the scheduler $\mathscr{S}$.

3.3. Indistinguishability

In a similar way as it has been done in Section 2.2, we now define what it means for two SyncJinja+ systems to be indistinguishable by an environment interacting with them.

We first define interfaces that systems use/provide, how systems are composed, and environments. We then define two forms of indistinguishability, namely perfect and computational indistinguishability. Since we consider asymptotic security, this involves to define programs that take a security parameter as input and that run in polynomial time in the security parameter.

We note that our definitions of indistinguishability follow the spirit of definitions of (computational) indistinguishability in the cryptographic literature (see e.g., [Can00, Küs06]), but, of course, also here instead of interactive Turing machines, we consider SyncJinja+ systems/programs. In particular, the simple communication model based on tapes is replaced by rich object-oriented interfaces between subsystems.

3.3.1. Interfaces and Composition

Before we define the notion of an interface, we emphasize that it should not be confused with the concept of interfaces in Java: Although our definition harks back to the informal definition of Java interfaces, we use here this term with a different, more theoretical connotation.

Definition 3.5. *An* interface *I is defined like a (SyncJinja+) system but where (i) all private fields and methods are dropped and (ii) method bodies as well as static field initializers are dropped.*

If I and I' are interfaces, then I' is a *subinterface* of I, written $I' \sqsubseteq I$, if I' can be obtained from I by dropping whole classes (with their method and field declarations), dropping methods and fields, dropping `extends` clauses, and/or adding the `final` modifier to class declarations.

Two interfaces are called *disjoint* if the set of class names declared in these interfaces are disjoint.

If S is a system, then the *public interface of S* is obtained from S by (1) dropping all private fields and methods from S and (2) dropping all method bodies and initializers of static fields.

Definition 3.6. *A system S* implements *an interface I, written $S : I$, if I is a subinterface of the public interface of S.*

Clearly, for every system S we have that $S : \emptyset$.

Definition 3.7. *We say that* a system S uses an interface *I, written $I \vdash S$, if, besides its own classes, S uses* at most *classes/methods/fields declared in I. We always assume that the public interface of S and I are disjoint.*

We note that if $I \sqsubseteq I'$ and $I \vdash S$, then $I' \vdash S$. We write $I_0 \vdash S : I_1$ for $I_0 \vdash S$ and $S : I_1$. If $I = \emptyset$, i.e., I is the empty interface, we often write $\vdash S$ instead of $\emptyset \vdash S$. Note that $\vdash S$ means that S is a program.

Definition 3.8. *Interfaces I_1 and I_2 are* compatible *if there exists an interface I such that $I_1 \sqsubseteq I$ and $I_2 \sqsubseteq I$.*

Intuitively, if two compatible interfaces contain the same class, the declarations of methods and fields of this class in those interfaces must be consistent (for instance, a field with the same name, if declared in both interfaces, must have the same type). Note that if I_1 and I_2 are disjoint, then they are compatible. Systems that use compatible interfaces and implement disjoint interfaces can be composed:

Definition 3.9 (Composition). *Let I_S, I_T, I'_S and I'_T be interfaces such that I_S and I_T are disjoint and I'_S and I'_T are compatible. Let S and T be systems such that not both S and T contain the method `main`, $I'_S \vdash S : I_S$, and $I'_T \vdash T : I_T$. Then, we say that S and T are* composable *and denote by $S \cdot T$ the* composition *of S and T which, formally, is the union of (declarations in) S and T. If the same classes are defined both in S and T (which may happen for classes not specified in I_S and I_T), then we always implicitly assume that these classes are renamed consistently in order to avoid name clashes.*

We emphasize that the interfaces between subsystems as considered here are quite different and much richer than the interfaces between interactive Turing machines considered in cryptography. Instead of plain bit strings that are sent over tapes between different machines, objects can be created, classes of another subsystem can be extended by inheritance, and data of different types, including references pointing to possibly complex objects, can be passed between different objects. Also, the flow of control is different. While in the Turing machine model, sending a message gives control to the receiver of the message and this control might not come back to the sender, in the object-oriented setting communication goes through method calls and fields. After a method call, control comes back to the caller, provided that potential exceptions are caught and the execution is not aborted.

We also emphasize that while a setting of the form $\vdash S : I$ and $I \vdash T$ i.e., in $S \cdot T$ the system T uses the interface I implemented by S, suggests that the initiative of accessing fields and calling methods always comes from T, it might also come from S by using *callback objects*: T could extend classes of S by inheritance, create objects of these classes and pass references to these objects to S (by calling methods of S). Then, via these references, S can call methods specified in T. (This, in fact, is a common programming technique.)

3.3.2. Environments

An environment will interact with one of two systems and it has to decide with which system it interacted (see Definitions 3.13 and 3.19). Its decision is written to a distinct static boolean variable `result`. That is, the environment plays the role the adversary plays in the cryptographic security definitions: it tries to distinguish two systems which are supposed to be indistinguishable.

For multi-threaded programs we also consider the scheduler as defined in Definition 3.1. As already mentioned in the previous section, in the indistinguishability definitions the scheduler is

also considered to be part of the adversary. We notice that the environment and the scheduler can be treated as a single entity (the adversary) since, although the scheduler, when invoked, knows only the identifiers of the currently active threads, it can establish a duplex communication with the environment and then collude: The environment could transfer information to the scheduler, for instance, by spawning either one or two threads to encode a zero or an one, respectively. The scheduler could establish a communication with the environment, for instance, by keeping on scheduling two threads where the scheduling of the first encodes a zero, whereas the scheduling of the second an one.[7]

Definition 3.10. *A (possibly multi-threaded) system E is called an* environment *if it declares a distinct private static variable* `result` *of type* `boolean` *with initial value* `false`*.*

In the rest of the Chapter, we (often implicitly) assume that the variable `result` is unique in every Java program, i.e., it is declared in at most one class of a program, namely, one that belongs to the environment.

Definition 3.11. *Let S be a system with $S : I$ for some interface I. Then an environment E is called an I-*environment for *S if there exists an interface I_E disjoint from I such that (i) $I_E \vdash S : I$ and $I \vdash E : I_E$ and (ii) either S or E contains* `main`*.*

Note that E and S, as in the above definition, are composable and $E \cdot S$ is a program.

For single-threaded programs i.e., when both E and S are Jinja+ systems, $E \cdot S$ already defines a *complete program*, i.e., a runnable program. In this case, for a finite run of $E \cdot S$, we call the value of `result` at the end of the run the *output of E* or the *output of the program $E \cdot S$*. For infinite runs, we define the output to be `false`. If $E \cdot S$ is a deterministic program, then we write $E \cdot S \rightsquigarrow$ `true` if the output of $E \cdot S$ is `true`. If $E \cdot S$ is a randomized program, we write $\mathsf{Prob}\{E \cdot S \rightsquigarrow \texttt{true}\}$ to denote the probability that the output of $E \cdot S$ is `true`.

For multi-threaded programs, i.e., if at least one between E and S contains multi-threaded constructs introduced in SyncJinja+, the notation introduced remains basically the same with the difference that now the program $E \cdot S$ runs under a scheduler $\mathscr{S}$. In this case, a *complete program* is denoted as $\{E \cdot S\}_{\mathscr{S}}$.

Definition 3.12 (same interface). *The systems S_1 and S_2* use the same interface *if (i) for every I_E, we have that $I_E \vdash S_1$ iff $I_E \vdash S_2$, and (ii) S_1 contains the method* `main` *iff S_2 contains* `main`*.*

Observe that if S_1 and S_2 use the same interface and we have that $S_1 : I$ and $S_2 : I$ for some interface I, then every I-environment for S_1 is also an I-environment for S_2.

3.3.3. Programs with security parameter

As mentioned at the beginning of this section, we need to consider programs that take a security parameter as input and run in polynomial time in this security parameter.

[7] The scheduler can establish a communication with the environment even if only one thread is available, using a covert (timing) channel: In order to encode zero, the scheduler could, for instance, make the (only) thread run for n steps whereas, to encode one, make this thread run for $n+1$ steps.

We require that for each program run under a scheduler $\mathscr{S}$, the number of steps performed in the run is a polynomial function in the security parameter. To ensure that all parts of a system have access to the security parameter, we fix a distinct interface I_{SP} consisting of (one class containing) one *public static final* variable `securityParameter`. We assume that, in all the considered systems/programs, this variable (after being initialized) is only read but never written to. Therefore, all parts of the considered system can, at any time, access the same, initial value of this variable. We fix the access of the scheduler $\mathscr{S}$ to the variable `securityParameter` (whose value is the same as the `securityParameter` in the system) in the same way.

For a natural number η, we define a system SP_η that implements the interface I_{SP} by setting the initial value of `securityParameter` to η. We do not fix here how this value is represented because the representation is not essential for our results; it could be represented as a linked list of objects or an array (see also the discussion below).

We call a system P such that $I_{SP} \vdash P$ a *program with a security parameter* or simply a *program* if the presence of a security parameter is clear from the context. Note that by this, $SP_\eta \cdot P$ is also a program, which we abbreviate by $P(\eta)$.

As far as asymptotic security is concerned, although our framework works fine with the definitions we have introduced so far, they are not perfectly aligned with the common practice of programming in Java. More specifically, messages, such as keys, ciphertexts, digital signatures, etc., are typically represented as arrays of bytes. However, this representation is bounded by the maximal length of an array, which is the maximal value of an integer (`int`). Therefore, following common programming practice, there would be a strict bound on, for example, the maximal size of keys (if represented as arrays of bytes). Since we consider asymptotic security, the size of keys should, however, grow with the security parameter.

One solution could be to use another representation of messages, such as lists of bytes. This, however, would result in unnatural programs and we, of course, want to be able to analyze programs as given in practice. Another solution could be to use concrete instead of asymptotic security definitions. However, most results in simulation-based security are formulated w.r.t. asymptotic security, and hence, we would not be able to reuse these results and avoid, for example, reproving from scratch realizations of ideal functionalities.

Therefore, we prefer the following solution.

We parameterize the semantics of SyncJinja+ with the maximal (absolute) value integers can take. So, if P is a deterministic, possibly multi-threaded, complete program (i.e., it might include the scheduler under which it runs), the *run of P with integer size $s \geq 1$* is a run of P where the maximal (absolute) value of integers is s; analogously for randomized programs. We write $P \rightsquigarrow_s \texttt{true}$ if the output of such a run is `true`; analogously, we define $\mathsf{Prob}\{P \rightsquigarrow_s \texttt{true}\}$. In our asymptotic formulations of indistinguishability, the size of integers may depend on the security parameter.

3.3.4. Perfect Indistinguishability

We now extend the notion of perfect indistinguishability introduced in [KTG12a] to a SyncJinja+ *deterministic* programs which run under a scheduler.

We say that a deterministic program P under a scheduler $\mathscr{S}$ *terminates for integer size s*, if the run of P with integer size s is finite.

Definition 3.13 (**Perfect indistinguishability**). *Let S_1 and S_2 be deterministic (possibly multi-threaded) systems with a security parameter and such that $S_1 : I$ and $S_2 : I$ for some interface I. Then, S_1 and S_2 are* perfectly indistinguishable w.r.t. *I, written* $S_1 \approx^{I,MT}_{\mathsf{perf}} S_2$, *if*

i) *S_1 and S_2 use the same interface and*

ii) *for every deterministic I-environment E for S_1 (and hence, S_2) with security parameter, for every security parameter η, for every integer size $s \geq 1$, and for every deterministic scheduler $\mathscr{S}$ it holds that if $\{E \cdot S_1(\eta)\}_{\mathscr{S}}$ and $\{E \cdot S_2(\eta)\}_{\mathscr{S}}$ terminate for integer size s, then $\{E \cdot S_1(\eta)\}_{\mathscr{S}} \rightsquigarrow_s \mathtt{true}$ iff $\{E \cdot S_2(\eta)\}_{\mathscr{S}} \rightsquigarrow_s \mathtt{true}$.*

We note that the notion of perfect indistinguishability introduced above is *termination-insensitive*, i.e. it puts no restrictions on non-terminating runs. This (weak) form of perfect indistinguishability, nevertheless implies computational indistinguishability (see Theorem 3.3).

3.3.5. Polynomially Bounded Systems

As already mentioned at the beginning of this section, in order to define the notion of computational indistinguishability we need to define programs, environments and *schedulers* whose runtime is polynomially bounded in the security parameter. For this purpose, we fix now and for the rest of this section a polynomially computable function *intsize* that takes a security parameter η as input and outputs a natural number ≥ 1. We require that the numbers returned by this function are bounded by a fixed polynomial in the security parameter. All notions defined in what follows are parameterized by that function. However, due to ease of notion this will not be made explicit.

Our runtime definitions follow the spirit of definitions in cryptographic definitions of simulation-based security, in particular, [Küs06].

We start with the definition of bounded schedulers. The number of steps such a scheduler performs in a run is bounded by a fixed polynomial independently of the scheduled program.

Definition 3.14 (**Bounded scheduler**). *A (possibly) randomized scheduler $\mathscr{S}$ is called* bounded *if there exists a polynomial q such that, for every SyncJinja+ program P with security parameter η running under $\mathscr{S}$ and for every run of $P(\eta)_{\mathscr{S}}$ (with integer size intsize(η)), the number of steps performed in the code of $\mathscr{S}$ does not exceed $q(\eta)$.*

We define now almost bounded programs. These are programs that, with overwhelming probability, terminate after a polynomial number of steps.

Definition 3.15 (**Almost bounded**). *A SyncJinja+ program P with security parameter η is* almost bounded *if for each bounded scheduler $\mathscr{S}$, there exists a polynomial q such that the probability that the sum of the length of a run of $P(\eta)$ (with integer size intsize(η)) and of the length of a run of $\mathscr{S}$ exceeds $q(\eta)$ is a negligible function in η.*[8]

[8] As usual, a function f from the natural numbers to the real numbers is *negligible*, if for every $c > 0$ there exists η_0 such that $f(\eta) \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$. A function f is overwhelming if 1-f is negligible.

It is easy to see that an almost bounded program P can be simulated by a probabilistic polynomial time Turing machine that simulates at most $q(\eta)$ steps of a run of $P(\eta)$ (with integer size $intsize(\eta)$) and produces output that is distributed the same up to a negligible difference.

We also define the notion of a bounded environment. As for bounded schedulers, the number of steps such an environment performs in a run is bounded by a fixed polynomial independently of the system the environment interacts with and on the scheduler under which the composed program it is supposed to run.

Definition 3.16 (**Bounded environment**). *An environment E is called* bounded *if there exists a polynomial q such that, for every system S such that E is an I-environment for S (for some interface I), for every bounded scheduler $\mathscr{S}$ and for every run of $\{E \cdot S(\eta)\}_{\mathscr{S}}$ (with integer size $intsize(\eta)$), the number of steps performed in the code of E does not exceed $q(\eta)$.*

The definitions of bounded scheduler and bounded environment make sense since both $\mathscr{S}$ and E can abort a run by calling `abort`.

If an environment E is both bounded and an I-environment for some system S, we call E a *bounded I-environment for S*. For the cryptographic analysis of systems to be meaningful, we study systems that run in polynomial time (with overwhelming probability) with any bounded environment and scheduler.

Definition 3.17 (**Environmentally I-bounded**). *A system S is* environmentally I-bounded*, if $S : I$, for each bounded I-environment E for S, the program $E \cdot S$ is almost bounded.*

It is typically easy to see that a system is environmentally I-bounded (for all functions *intsize*).

We note that environmentally I-bounded systems, as defined above, are reactive systems that are free to process an unbounded number of requests of the environment E. In particular, a reactive system S does not need to terminate after some fixed and bounded number of requests. Clearly, every bounded I-environment, being bounded, will only invoke S a bounded number of times. More precisely, the number of invocations the environment makes is bounded by some polynomial in the security parameter.

3.3.6. Computational Indistinguishability

Having defined polynomially bounded systems and programs, we are now ready to define computational indistinguishability of systems, where, again, we fix the function *intsize*. (However, computational guarantees for Java programs will be independent of a specific function *intsize*.) We start with the notion of a computationally equivalent programs.

Definition 3.18 (**Computational Equivalence**). *Let P_1 and P_2 be (complete, possibly probabilistic and multi-threaded) programs with security parameter η. Then P_1 and P_2 are* computationally equivalent*, written $P_1 \equiv_{\mathsf{comp}} P_2$, if $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}|$ is a negligible function in the security parameter η.*

We notice that $\equiv_{\mathsf{comp}}$ is an equivalence relation (see Appendix F.2 for the proof). Moreover, we hightlight that, in the above definition, P_1 could for instance be a single-threaded Jinja+ system, whereas P_2 could represent a multi-threaded SyncJinja+ program $\hat{P}$ running under a scheduler $\mathscr{S}$.

That is, as long as, at the end of the two runs, the variable `result` is the same (up to negligible probability), the two systems are said to be computationally equivalent.

Definition 3.19 (Computational indistinguishability). *Let S_1 and S_2 be environmentally I-bounded (possibly multi-threaded) systems. Then S_1 and S_2 are* computationally indistinguishable w.r.t. *I, written $S_1 \approx^{I,MT}_{\mathsf{comp}} S_2$, if*

i) S_1 and S_2 use the same interface, and

ii) for every bounded I-environment E for S_1 (and hence, S_2) and for every bounded scheduler $\mathscr{S}$ we have that $\{E \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S_2\}_{\mathscr{S}}$.

This definition is typically applied to programs that do not use the statement `abort`. However, our results also work in this case.

Moreover, we note that we do neither require S_1/S_2 nor E to necessarily be multi-threaded systems: it is enough that one of them is multi-threaded to obtain a multi-threaded composed program. This becomes relevant in the extension of the indistinguishability and simulatability results already stated w.r.t. single-threaded adversaries. In Section 3.8, we formally state and prove under which assumptions the level of security of a single-threaded systems remains unchanged when composed with a multi-threaded adversary.

The above definition of indistinguishability is w.r.t. uniform environments. A definition w.r.t. non-uniform environments (i.e., environments which use also some external interface) can be obtained in a straightforward way by giving the environment additional auxiliary input (besides the security parameter).

Furthermore, we point out that in the above definition two cases can occur: (1) `main` is defined in E or (2) `main` is defined in both S_1 and S_2. In the first case, E can freely create objects of classes in the interface I (which is a subset of classes of S_1/S_2) and initiate calls. Eventually, even in case of exceptions, E can get back control (method calls return a value to E and E can catch exceptions if necessary), unless S_1/S_2 uses `abort`. On the other hand, the kind of control E has in the case (2), heavily depends on the specification of S_1/S_2. This can go from having as much control as in case (1) to being basically a passive observer. For example, `main` (as specified in S_1/S_2) could call a method of E and from then on E can use the possibly very rich interface I as in case (1). The other extreme is that I is empty, say, so E cannot create objects of (classes of) S_1/S_2 by itself, only S_1/S_2 can create objects of (classes of) E and of S_1/S_2. Hence, S_1/S_2 has more control and can decide, for instance, how many and which objects are created and when E is contacted. Still even in this case, if so specified, S_1/S_2 could give E basically full control by callback objects (see Section 3.3.1). To further illustrate the richness of the interfaces compared to Turing machine models, we also note that E could also extend classes of S_1/S_2 and by this, if not properly protected, might get access to information kept in these classes.

3.4. Simulatability and Universal Composition

We now formulate what it means for a system to realize another system, in the spirit of the simulation-based approach. As before, we fix a function *intsize* (see Section 3.3.5) for the rest of this section. Typically, one would prove that one system realizes the other for all such functions.

Our formulation of the realization of one system by another follows the spirit of strong simulatability in the simulation-based approach (see e.g., [Küs06]). In a nutshell, the definition says that, given (real) system R, it realizes an (ideal) system F if there exists a simulator S such that R and $S \cdot F$ behave almost the same in every bounded environment.

Definition 3.20 (Strong Simulatability). *Let $I_{in}, I_{out}, I_E, I_S$ be disjoint interfaces. Let F and R be two (possibly multi-threaded) systems. Then R* realizes *F* w.r.t. the interfaces I_{out}, I_{in}, I_E, and I_S, *written $R \leq^{(I_{out}, I_{in}, I_E, I_S), MT} F$ or simply $R \leq^{MT} F$, if*

i) $I_E \cup I_{in} \vdash R : I_{out}$ *and* $I_E \cup I_{in} \cup I_S \vdash F : I_{out}$;

ii) either both F and R or neither of these systems contain the method `main`;

iii) R is an environmentally I_{out}-bounded system (F does not need to be);

iv) there exists a (possibly multi-threaded) system S, the simulator, such that S does not contain `main`, *$I_E \vdash S : I_S$, $S \cdot F$ is environmentally I_{out}-bounded, and $R \approx^{I_{out}, MT}_{\mathsf{comp}} S \cdot F$.*

The intuition behind the way the interfaces between the different components (environment, ideal and real functionalities, simulator) are defined is as follows: Both R and F provide the same kind of functionality/service, specified by the interface I_{out}. They may require some (trusted) services I_{in} from another system component and some services I_E from an (untrusted) environment, for example, networking and certain other libraries. In addition, the ideal functionality F may require services I_S from the simulator S, which in turn may require services I_E from the environment. We recall from the discussion in Section 3.3.1 that the interfaces can be very rich, as they model communication and method calls in both directions.

In the applications we envision F will typically be an ideal functionality for one or more cryptographic primitives and its realization R will basically be the actual cryptographic schemes.

The notion of strong simulatability, as introduced above, enjoys important basic properties, namely, reflexivity and transitivity, and allows one to prove a fundamental composition theorem.

To show these results, we need the following lemma.

Lemma 3.1. *Let I_E and I be disjoint interfaces and let S_1 and S_2 be environmentally I-bounded systems such that $S_1 \approx^{I,MT}_{\mathsf{comp}} S_2$ (in particular, S_1 and S_2 use the same interface) and $I_E \vdash S_1 : I$, and hence, $I_E \vdash S_2 : I$. Let E be a not necessarily bounded SyncJinja+ I-environment for S_1 (and hence, S_2) with $I \vdash E : I_E$ such that $E \cdot S_1$ is almost bounded. Then $E \cdot S_2$ is almost bounded and for each bounded scheduler $\mathscr{S}$ we have $\{E \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S_2\}_{\mathscr{S}}$.*

Proof. Let I, I_E, S_1, S_2, E, and $\mathscr{S}$ be given as stated in the lemma. We need to show that $E \cdot S_2$ is almost bounded and that $\{E \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S_2\}_{\mathscr{S}}$.

Since $E \cdot S_1$ is almost bounded, for each bounded scheduler $\mathscr{S}$ there exists a polynomial p such that the probability that the sum of the length of the run of the scheduler $\mathscr{S}$ and of the length of the run of the system $E \cdot S_1$ with security parameter η (and integer size $intsize(\eta)$) exceeds $p(\eta)$ is negligible. Now let us denote by $[E]$ the system that is defined just as E, but which in addition has a private static counter (defined in some new class in $[E]$) and where the code of E is modified such that whenever a step in the code of E is performed (according to the small-step SyncJinja+ semantics), then the counter is increased. Once the bound $p(\eta)$ is reached, $[E]$ performs `abort`.

By construction of $[E]$ it is easy to see that $[E]$ is a bounded environment because $[E]$ does not simulate more than $p(\eta)$ steps of E, where each step of E can be simulated in a number of steps bounded by a constant. Also, $[E]$ behaves exactly like E up to the point where the bound $p(\eta)$ is reached.

Let $\mathscr{S}$ be a bounded scheduler. As further explained below, from the construction of $[E]$ we obtain:

$$\begin{aligned}\{E \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{[E] \cdot S_1\}_{\mathscr{S}} &\equiv_{\mathsf{comp}} \{[E] \cdot S_2\}_{\mathscr{S}} \\ &\equiv_{\mathsf{comp}} \{E \cdot S_2\}_{\mathscr{S}}.\end{aligned}$$

The first equivalence holds because, since $E \cdot S_1$ is almost bounded, E reaches the bound $p(\eta)$ when running with S_1 only with negligible probability. Hence, the assignment of E and $[E]$ to `result` is the same with overwhelming probability.

The second equivalence is true because $S_1 \approx^{I,MT}_{\mathsf{comp}} S_2$, $\mathscr{S}$ is bounded and $[E]$ is a *bounded* I-environment for S_1 and S_2.

The third equivalence holds because in the system $\{E \cdot S_2\}_{\mathscr{S}}$ the bound $p(\eta)$ is reached also only with negligible probability: Otherwise, we could easily turn $[E]$ into a bounded environment E' that distinguishes S_1 and S_2, namely, E' works just as $[E]$ but outputs `true`, i.e., assigns `true` to the variable `result` if and only if the bound $p(\eta)$ is reached. So, if, when E interacts with S_2, the bound were reached with non-negligible probability, E' could distinguish between S_1 and S_2. It follows that $E \cdot S_2$ is also almost bounded and that the last equivalence holds. □

Now we can prove reflexivity and transitivity of strong simulatability. The proofs are similar to those for Jinja+ systems in [KTG12a].

Lemma 3.2 (Reflexivity of strong simulatability). *Let I_{out}, I_{in}, and I_E be disjoint interfaces and let R be a system such that $I_E \cup I_{in} \vdash R : I_{out}$ and R is environmentally I_{out}-bounded. Then, $R \leq^{MT} R$, i.e., R realizes itself.*

Proof. We define $S = \emptyset$ i.e., S does not contain any class, and immediately obtain that $R \approx^{I_{out},MT}_{\mathsf{comp}} R = S \cdot R$. □

Lemma 3.3 (Transitivity of strong simulatability). *Let I_{out}, I_{in}, I_E, I^0_S, and I^1_S be disjoint interfaces and let R_0, R_1, and R_2 be environmentally I-bounded systems. If $R_1 \leq^{(I_{out},I_{in},I_E \cup I^1_S, I^0_S),\, MT} R_0$ and $R_2 \leq^{(I_{out},I_{in},I_E,I^1_S),\, MT} R_1$, then $R_2 \leq^{(I_{out},I_{in},I_E,I^0_S \cup I^1_S),\, MT} R_0$.*

Proof. Under the assumption of the lemma, we know that there exist S_0 and S_1 such that $I_E \cup I^1_S \vdash S_0 : I^0_S$, $I_E \vdash S_1 : I^1_S$, $S_0 \cdot R_0$ and $S_1 \cdot R_1$ are environmentally I_{out}-bounded, and $R_1 \approx^{I_{out},MT}_{\mathsf{comp}} S_0 \cdot R_0$ and $R_2 \approx^{I_{out},MT}_{\mathsf{comp}} S_1 \cdot R_1$. We define $S = S_0 \cdot S_1$. Obviously, we have that $I_E \vdash S : I^0_S \cup I^1_S$. Now let E be a bounded I_{out}-environment for R_2 and let $\mathscr{S}$ be a bounded scheduler. Then, we obtain:

$$\begin{aligned}\{E \cdot R_2\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S_1 \cdot R_1\}_{\mathscr{S}} &\equiv_{\mathsf{comp}} \{E \cdot S_1 \cdot S_0 \cdot R_0\}_{\mathscr{S}} \\ &\equiv_{\mathsf{comp}} \{E \cdot S \cdot R_0\}_{\mathscr{S}} \ .\end{aligned}$$

The first equivalence holds because of our assumptions. For the second equivalence, first note that $\{E \cdot S_1 \cdot R_1\}_{\mathscr{S}}$ is almost bounded and $R_1 \approx^{I_{out},MT}_{\mathsf{comp}} S_0 \cdot R_0$. By Lemma 3.1, we now obtain that $\{E \cdot S_1 \cdot S_0 \cdot R_0\}_{\mathscr{S}}$ is almost bounded and that the second equivalence holds. □

In short, the following composition theorem says that if R_0 realizes F_0 and R_1 realizes F_1, then the composed real system $R_0 \cdot R_1$ realizes the composed ideal system $F_0 \cdot F_1$. In other words, in order to obtain security of a composed systems, it suffices to prove the realizations of its components separately.

Theorem 3.1 (Composition Theorem). *Let I_0, I_1, I_E, I_S^0, and I_S^1 be disjoint interfaces and let R_0, F_0, R_1, and F_1 be systems such that $R_0 \leq^{(I_0,I_1,I_E,I_S^0),\,MT} F_0$, $R_1 \leq^{(I_1,I_0,I_E,I_S^1),\,MT} F_1$, not both R_0 and R_1 contain `main`, and $R_0 \cdot R_1$ are environmentally $(I_0 \cup I_1)$-bounded. Then, $R_0 \cdot R_1 \leq^{(I_0 \cup I_1,\emptyset,I_E,I_S^0 \cup I_S^1),\,MT}$ $F_0 \cdot F_1$.*

Proof. Under the assumptions of this theorem, there exist S_0 and S_1 such that $I_E \vdash S_i : I_S^i$, $S_i \cdot F_i$ is environmentally I_i-bounded and $R_i \approx_{\mathsf{comp}}^{I_i,MT} S_i \cdot F_i$ for $i \in \{0,1\}$.

We define $S = S_0 \cdot S_1$, $R = R_0 \cdot R_1$, $F = F_0 \cdot F_1$, $I = I_0 \cup I_1$, and $I_S = I_S^0 \cup I_S^1$. In order to show that $R \leq^{(I,\emptyset,I_E,I_S),\,MT} F$ it suffices to prove that

(a) $S \cdot F$ is environmentally I-bounded and

(b) $R \approx_{\mathsf{comp}}^{I,MT} S \cdot F$;

the remaining conditions of Definition 3.20 are obvious.

Let E be a bounded I-environment for R, and let $\mathscr{S}$ be a bounded scheduler. Similarly to the proof of Lemma 3.3, by Lemma 3.1 we obtain the following sequence of equivalences:

$$\begin{aligned}\{E \cdot R\}_{\mathscr{S}} &= \{E \cdot R_0 \cdot R_1\}_{\mathscr{S}} \\ &\equiv_{\mathsf{comp}} \{E \cdot R_0 \cdot (S_1 \cdot F_1)\}_{\mathscr{S}} \\ &\equiv_{\mathsf{comp}} \{E \cdot (S_0 \cdot F_0) \cdot (S_1 \cdot F_1)\}_{\mathscr{S}} = \{E \cdot S \cdot F\}_{\mathscr{S}},\end{aligned}$$

which imply Item (b).

By Lemma 3.1 we, in particular, get that all the above systems are almost bounded. Since we quantified over all bounded I-environments for R (and hence, $S \cdot F$) it follows that $S \cdot F$ is environmentally I-bounded, thus Item (a) holds as well. □

Note that with $R_0 \cdot R_1 \leq^{(I_0 \cup I_1,\emptyset,I_E,I_S^0 \cup I_S^1),\,MT} F_0 \cdot F_1$ we have that $R_0 \cdot R_1$ realizes $F_0 \cdot F_1$ also for all subinterfaces of $I_0 \cup I_1$.

For simplicity, Theorem 3.1 is stated in such a way that the trusted service that R_i may use is completely provided by R_{i-1}, namely through I_{i-1}. It is straightforward (only heavy in notation) to state and prove the theorem for the more general case that the trusted service is only partially provided by the other system.

3.5. From Perfect to Computational Indistinguishability

We now prove that if two systems that use an ideal functionality are perfectly indistinguishable, then these systems are computationally indistinguishable if the ideal functionality is replaced by its realization.

As before, we fix a function *intsize* (see Section 3.3.5) for the rest of this section. The proof is done via two theorems. The first says that if two systems that use an ideal functionality are computationally indistinguishable, then they are also computationally indistinguishable if the ideal functionality is replaced by its realization.

Theorem 3.2. *Let I, J, I_E, I_S, and I_P be disjoint interfaces with $J \sqsubseteq I_P \cup I$. Let F, R, P_1, and P_2 be systems such that*

i) $I_E \cup I \vdash P_1 : I_P$ *and* $I_E \cup I \vdash P_2 : I_P$*;*

ii) $R \leq^{(I,I_P,I_E,I_S),\,MT} F$*, in particular,* $I_E \cup I_P \vdash R : I$ *and* $I_E \cup I_P \cup I_S \vdash F : I$*;*

iii) P_1 *contains* `main` *iff* P_2 *contains* `main`*;*

iv) not both P_1 *and* F *(and hence,* R*) contain the method* `main`*;*

v) $F \cdot P_i$ *and* $R \cdot P_i$*, for* $i \in \{1,2\}$*, are environmentally* J*-bounded.*

Then, $F \cdot P_1 \approx^{J,MT}_{\mathsf{comp}} F \cdot P_2$ *implies* $R \cdot P_1 \approx^{J,MT}_{\mathsf{comp}} R \cdot P_2$*.*

Proof. Assume that $F \cdot P_1 \approx^{J,MT}_{\mathsf{comp}} F \cdot P_2$. In particular, $F \cdot P_1$ and $F \cdot P_2$ use the same interface and, therefore, $R \cdot P_1$ and $R \cdot P_2$ use the same interface as well.

Let E be a bounded J-environment for $R \cdot P_1$, and let $\mathscr{S}$ be a bounded scheduler. We need to show that $\{E \cdot R \cdot P_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot R \cdot P_2\}_{\mathscr{S}}$.

Because $R \leq^{(I,I_P,I_E,I_S),\,MT} F$, there exists a simulator S such that $I_E \vdash S : I_S$, $S \cdot F$ is environmentally I-bounded and

$$R \approx^{I,MT}_{\mathsf{comp}} S \cdot F \tag{3.1}$$

Now, because $R \cdot P_i$, $i \in \{1,2\}$, is environmentally J-bounded, the system $E \cdot R \cdot P_i$ is almost bounded. By (3.1) and Lemma 3.1 we can conclude that $E \cdot S \cdot F \cdot P_i$ is almost bounded and

$$\{E \cdot R \cdot P_i\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S \cdot F \cdot P_i\}_{\mathscr{S}} \ . \tag{3.2}$$

As we have assumed that $F \cdot P_1 \approx^{J,MT}_{\mathsf{comp}} F \cdot P_2$, by Lemma 3.1 we obtain

$$\{E \cdot S \cdot F \cdot P_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S \cdot F \cdot P_2\}_{\mathscr{S}}. \tag{3.3}$$

Combining (3.2) and (3.3), we obtain $\{E \cdot R \cdot P_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot R \cdot P_2\}_{\mathscr{S}}$ i.e., $R \cdot P_1 \approx^{J,MT}_{\mathsf{comp}} R \cdot P_2$. □

For simplicity of presentation, the theorem is formulated in such a way that P_i, $i \in \{1,2\}$, uses only I as a (trusted) service and F/R uses I_P. It is straightforward to also allow for other external services.

We now show that perfect indistinguishability implies computational indistinguishability.

Theorem 3.3. *Let I be an interface and let S_1 and S_2 be deterministic, environmentally I-bounded systems such that $S_i : I$, for $i \in \{1,2\}$, and S_1 and S_2 use the same interface. Then, $S_1 \approx^{I,MT}_{\mathsf{perf}} S_2$ implies $S_1 \approx^{I,MT}_{\mathsf{comp}} S_2$.*

Proof. Let E be a bounded I-environment for S_1 (and hence, S_2). For a finite bit string r, let E_r denote the deterministic system obtained from E by fixing its randomness by r in the following way: The primitive randomBit() is replaced by a method (along with a new static field) declared within E_r such that the first $|r|$ bits returned by the method are chosen according to r; all the remaining bits returned by this method are 0. It follows with $S_1 \approx^{I,MT}_{\mathsf{perf}} S_2$ that (*) for all security parameters η, for all r, for all integer sizes $s \geq 1$, and for all deterministic schedulers $\mathscr{S}$ such that $\{E_r \cdot S_1(\eta)\}_{\mathscr{S}}$ and $\{E_r \cdot S_2(\eta)\}_{\mathscr{S}}$ terminate for integer size s it holds that $\{E_r \cdot S_1(\eta)\}_{\mathscr{S}} \rightsquigarrow_s \mathtt{true}$ iff $\{E_r \cdot S_2(\eta)\}_{\mathscr{S}} \rightsquigarrow_s \mathtt{true}$. This implies

$$\{E \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E \cdot S_2\}_{\mathscr{S}}$$

because: By assumption, $E \cdot S_1$ and $E \cdot S_2$ are almost bounded. Hence, there exists a polynomial p such that the probability that the length of a run (with integer size $intsize(\eta)$) of $\{E \cdot S_1(\eta)\}_{\mathscr{S}}$ or $\{E \cdot S_2(\eta)\}_{\mathscr{S}}$, both calculated as the sum of the length of the run of $E \cdot S_{i\in\{1,2\}}(\eta)$ and of $\mathscr{S}$, exceeds $p(\eta)$ is negligible. So in all runs, except for a negligible fraction, at most $p(\eta)$ random bits are needed. Moreover, for almost all bit strings r of length at most $p(\eta)$ and integers of size $intsize(\eta)$, we have that the runs of $\{E_r \cdot S_1(\eta)\}_{\mathscr{S}}$ and $\{E_r \cdot S_2(\eta)\}_{\mathscr{S}}$ terminate for integer size $intsize(\eta)$. Now by (*) we know that for such r the output of the runs of $\{E_r \cdot S_1(\eta)\}_{\mathscr{S}}$ and $\{E_r \cdot S_2(\eta)\}_{\mathscr{S}}$ with integer size $intsize(\eta)$ is the same.

Hence $S_1 \approx^{I,MT}_{\mathsf{comp}} S_2$. □

By combining Theorem 3.2 and Theorem 3.3, we obtain the desired result explained at the beginning of this section.

Corollary 3.1. *Under the assumption of Theorem 3.2 and moreover assuming that $P_1 \cdot F$ and $P_2 \cdot F$ are deterministic systems, it follows that $P_1 \cdot F \approx^{J,MT}_{\mathsf{perf}} P_2 \cdot F$ implies $P_1 \cdot R \approx^{J,MT}_{\mathsf{comp}} P_2 \cdot R$.*

Recall that $P_1 \cdot R \approx^{J,MT}_{\mathsf{comp}} P_2 \cdot R$ is (implicitly) defined w.r.t. the integer size function $intsize(\eta)$. However, since the statement $P_1 \cdot F \approx^{J,MT}_{\mathsf{perf}} P_2 \cdot F$ does not depend on any integer size function, we obtain that computational indistinguishability holds independently of a specific integer size function.

3.6. Perfect Indistinguishability and Noninterference

In this section we relate the notion of perfect indistinguishability for multi-threaded SyncJinja+ systems (Definition 3.13) with a notion of noninterference also for multi-threaded SyncJinja+ systems.

This notion naturally extends the corresponding notion of (standard) noninterference for confidentiality [GM82a] which requires the absence of information flowing from high to low variables within a program. More precisely, we now extend the notion of noninterference for Jinja+ programs introduced in [KTG12a] to SyncJinja+ programs. Let P be SyncJinja+ program with some static variables $\vec{x}$ of primitive types that are labeled as high. Also, some other static variables of primitive types are labeled as low. We say that $P[\vec{x}]$ is a *program with high and low variables*. As in [KTG12a], by $P[\vec{a}]$ we denote the program P where the high variables $\vec{x}$ are

initialized with values $\vec{a}$ and the low variables are initialized as specified in P. We assume that the length of $\vec{x}$ and $\vec{a}$ are the same and $\vec{a}$ contains values of appropriate types; in such a case we say that $\vec{a}$ is valid. Now, noninterference for a deterministic SyncJinja+ program is defined as follows.

Definition 3.21 (**Noninterference for SyncJinja+ programs).** *Let $P[\vec{x}]$ be a deterministic SyncJinja+ (hence, possibly multi-threaded) program with high and low variables. Then, $P[\vec{x}]$* has the noninterference property *if the following holds: For all deterministic schedulers $\mathscr{S}$, for all valid $\vec{a}_1$ and $\vec{a}_2$, for all integer sizes $s \geq 1$, if $P[\vec{a}_1]_{\mathscr{S}}$ and $P[\vec{a}_2]_{\mathscr{S}}$ terminate for integer size s, then at the end of these two runs, the values of the low variables are the same.*

Similarly to the definition of perfect indistinguishability (Definition 3.13), the above definition captures *termination-insensitive* noninterference for SyncJinja+ programs. We note that the noninterference property is quite powerful: P could have just one high variable of type boolean. Depending on the value of this variable P could run one of two systems S_1 and S_2, illustrating that the noninterference property can be as powerful as perfect indistinguishability.

The above notion of noninterference deals with programs, i.e., closed systems. The systems to be analyzed are, however, often open: they interact with a network or use some libraries which are not necessarily trusted and, hence, are not part of the code to be analyzed; instead, they are considered as part of the environment with unspecified behavior. Therefore, as part of the CVJ framework [KTG12a], the notion of noninterference has been generalized to open systems i.e., systems not completely defined.

Definition 3.22 (**Noninterference in an open system).** *Let I be an interface and let $S[\vec{x}]$ be a (not necessarily closed, but possibly multi-threaded) SyncJinja+ deterministic system with a security parameter, high and low variables, and such that $S : I$. Then, $S[\vec{x}]$ is I-*noninterferent *if for every deterministic I-environment E for $S[\vec{x}]$ and every security parameter η, noninterference holds for the program $E \cdot S[\vec{x}](\eta)$, where the variable* `result` *declared in E is considered to be a low variable.*

We now show that noninterference for SyncJinja+ is equivalent to perfect indistinguishability as defined in Definition 3.13.

Theorem 3.4. *Let $S[\vec{x}]$ be as given in Definition 3.22 with no variable of S labeled as low (only the variable* `result` *declared in the environment is labeled as low). Then, the following two statements are equivalent:*

a) I-noninterference holds for $S[\vec{x}]$.

b) For all valid inputs $\vec{a}_1$ and $\vec{a}_2$ for $S[\vec{x}]$, we have $S[\vec{a}_1] \approx^{I,MT}_{\mathsf{perf}} S[\vec{a}_2]$.

Proof. Since $S[\vec{x}]$ is I-noninterferent, for every deterministic I-environment E for $S[\vec{x}]$ and every security parameter η, noninterference holds for the system $E \cdot S[\vec{x}](\eta)$, where the parameters $\vec{x}$ denote the only high variables while `result` declared in E is the only low one. By the definition of noninterference for SyncJinja+ programs (Definition 3.21), for all valid inputs $\vec{a}_1$ and $\vec{a}_2$, for all deterministic schedulers $\mathscr{S}$, and for all integer sizes $s \geq 1$, if $\{E \cdot S[\vec{a}_1](\eta)\}_{\mathscr{S}}$ and

$\{E \cdot S[\vec{a}_2](\eta)\}_{\mathscr{S}}$ terminate for integer size s, then, at the end of these two runs, the values of the low variables `result` are the same. This statement is anlogous to the statement in the notion of perfect indistinguishability introduced in Definition 3.13, where $S_1(\eta)$ is $S[\vec{a}_1](\eta)$ and $S_2(\eta)$ is $S[\vec{a}_2](\eta)$. □

3.7. From Noninterference to Computational Indistinguishability

The first main result of this chapter immediately follows from the combination of Theorem 3.4 and Corollary 3.1.

Theorem 3.5 (The CVJ Theorem for SyncJinja+ systems). *Let I and J be disjoint interfaces. Let F, R, $S[\vec{x}]$ be (possibly multi-threaded) SyncJinja+ systems such that*

i) $R \leq^{J,MT} F$,

ii) not both $S[\vec{x}]$ and F (and hence, R) contain the method `main`,

iii) $S[\vec{x}] \cdot F$ *is deterministic, and*

iv) $S[\vec{x}] \cdot F : I$ *(and hence,* $S[\vec{x}] \cdot R : I$*).*

Now, if $S[\vec{x}] \cdot F$ is I-noninterferent, then, for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), we have that $S[\vec{a}_1] \cdot R \approx^{I,MT}_{\mathsf{comp}} S[\vec{a}_2] \cdot R$.

Proof. Let F, R, and $S[\vec{x}]$ be such that $R \leq^{J,MT} F$ and not both $S[\vec{x}]$ and F (and hence, R) contain the method `main`. Let $S[\vec{x}] \cdot F$ be a deterministic I-noninterferent system.

Then, by Theorem 3.4, for all valid inputs $\vec{a}_1$ and $\vec{a}_2$ for $S[\vec{x}]$ we have $S[\vec{a}_1] \cdot F \approx^{I,MT}_{\mathsf{perf}} S[\vec{a}_2] \cdot F$.

Since $S[\vec{x}] \cdot F$ is deterministic, not both $S[\vec{x}]$ and F (and hence, R) contain `main`, and $R \leq^{J,MT} F$, by Corollary 3.1, we have that $S[\vec{a}_1] \cdot F \approx^{I,MT}_{\mathsf{perf}} S[\vec{a}_2] \cdot F$ implies $S[\vec{a}_1] \cdot R \approx^{I,MT}_{\mathsf{comp}} S[\vec{a}_2] \cdot R$. □

As for single-threaded systems, the typical use of this theorem is that the cryptographic operations that S needs to perform are carried out using the system R (e.g., a cryptographic library). The theorem now says that to prove cryptographic privacy of the secret inputs ($\forall\ \vec{a}_1, \vec{a}_2$: $S[\vec{a}_1] \cdot R \approx^{I,MT}_{\mathsf{comp}} S[\vec{a}_2] \cdot R$) it suffices to prove I-noninterference for $S[\vec{x}] \cdot F$, i.e., the system where R is replaced by the ideal counterpart F (the ideal cryptographic library). The ideal functionality F, which in our case will model cryptographic primitives in an ideal way, can typically be formulated without probabilistic operations and also the ideal primitives specified by F will be secure even in presence of unbounded adversaries, the kind of adversaries considered in information-flow security.

Therefore, as already mentioned in the introduction of this chapter, this result reduces the problem of checking computational indistinguishability for a SyncJinja+ system that use real cryptographic operations to checking noninterference for the same system, where the cryptographic operations have been replaced by their ideal counterpart.

3.8. From Single-Threaded to Multi-Threaded Programs

Based on the definition of indistinguishability for Jinja+ systems presented in Section 2.2, we can now state the second main result of this chapter which shows that, as long as a (environmentally I-bounded) system S is considered to be atomic, every multi-threaded adversary interacting with S through I is as powerful as a single-threaded one.

Theorem 3.6. *Let S_1 and S_2 be two environmentally I-bounded Jinja+ (hence single-threaded) systems such that they are computationally indistinguishable according to Definition 2.1: $S_1 \approx^{I,ST}_{\mathsf{comp}} S_2$. Then, S_1 and S_2, if considered to be* atomic*, are computationally indistinguishable also according to Definition 3.19, i.e., for every bounded multi-threaded SyncJinja+ I-environment E^{MT} for S_1/S_2 and for every bounded scheduler $\mathscr{S}$, we have $\{E^{MT} \cdot S_1\}_{\mathscr{S}} \equiv_{\mathsf{comp}} \{E^{MT} \cdot S_2\}_{\mathscr{S}}$.*

We note that the assumption of considering S_1 and S_2 to be atomic is fundamental to ensure (and to prove) their computationally indistinguishability property: The scheduler, which is considered to be part of the adversary, could otherwise distinguish beetween them by simply measuring the difference in the number of steps performed by these two systems. Furthermore, as already mentioned in the introduction of this chapter, the systems we are interested in contain cryptographic code whose security guarantees are based on cryptographic security games, e.g., for defining IND-CCA2 security (see Appendix A), which are inherently *sequential* and hence they do not capture timing attacks. To maintain these guarantees, the execution of the cryptographic operations in S_1 and S_2 must therefore neither be interleaved by any another thread nor by the scheduler.

Thanks to Theorem 3.6, whose proof can be found in Appendix F.3, we can also prove that if a single-threaded system R realizes another single-threaded system F, then this realization carries over also against multi-threaded adversaries, as lons as the assumption of considering R and F atomic is maintained.

To properly state and prove the theorem, we need to first recall from [KTG12a] the notion of strong simulatability for Jinja+ systems. We note that this notion has been already presented in Chapter 2 (see Definition 2.2), but only in a simplified way.

Definition 3.23 (Strong Simulatability for Jinja+ systems [KTG12a]). *Let $I_{in}, I_{out}, I_E, I_S$ be disjoint interfaces. Let F and R be two Jinja+ systems. Then R* realizes F w.r.t. the interfaces I_{out}, I_{in}, I_E, and I_S, *written $R \leq^{(I_{out}, I_{in}, I_E, I_S),\, ST} F$ or simply $R \leq^{ST} F$, if*

i) $I_E \cup I_{in} \vdash R : I_{out}$ *and* $I_E \cup I_{in} \cup I_S \vdash F : I_{out}$,

ii) either both F and R or neither of these systems contain the method `main`,

iii) R is an environmentally I_{out}-bounded system (F does not need to be), and

iv) there exists a Jinja+ system S (the simulator) such that S does not contain `main`, $I_E \vdash S : I_S$, *$S \cdot F$ is environmentally I_{out}-bounded, and $R \approx^{I_{out},ST}_{\mathsf{comp}} S \cdot F$ according to Definition 2.1.*

We are now able to state the theorem that allows us to carry over to SyncJinja+ also all the realization results which are already stated and proven for Jinja+ systems, again under the assumption that these systems are considered to be atomic.

Theorem 3.7. *Let R and F be two Jinja+ systems such that R realizes F according to Definition 3.23 i.e., $R \leq^{ST} F$. Then, if R and F are considered to be* atomic*, R realizes F also according to Definition 3.20 i.e., $R \leq^{MT} F$.*

Proof. We have to show that, if $R \leq^{ST} F$ and both R and F are atomic systems, there exists a SyncJinja+ simulator S such that $R \approx_{\mathsf{comp}}^{I_{out},MT} S \cdot F$, for an interface I_{out} such that R is environmentally I_{out}-bounded.

Since $R \leq^{ST} F$, there exists a Jinja+ simulator $\tilde{S}$ such that $R \approx_{\mathsf{comp}}^{I_{out},ST} \tilde{S} \cdot F$. Since Jinja+ programs are a subset of SyncJinja+ programs, we can set the SyncJinja+ simulator S to be the Jinja+ simulator $\tilde{S}$, considering it as an atomic system. Then, R and $\tilde{S} \cdot F$ are atomic Jinja+ systems such that $R \approx_{\mathsf{comp}}^{I_{out},ST} \tilde{S} \cdot F$. Hence, by Theorem 3.6, we can assert $R \approx_{\mathsf{comp}}^{I_{out},MT} \tilde{S} \cdot F$. Since all the other conditions of Definition 3.20 are the same as the conditions of Definition 3.23, we can conclude $R \leq^{MT} F$. □

4. Instantiating and Applying the CVJ Framework

Cryptographic primitives, such as symmetric and asymmetric encryption, digital signatures, nonce generation, message authentication codes (MACs), Diffie-Hellmann key exchanges, key derivation, commitment schemes, and so on, are essential building blocks of many security protocols, such as TLS, SSH, IPsec, IEEE 802.11i,[9] etc., broadly used nowadays to protect the communication of sensible data over untrusted networks, such as the Internet. Due to their relevance in security critical applications, in last years there has been a substantial effort in (re)designing such cryptographic primitives in a modular way so that, once mathematically proven secure under strong cryptographic assumptions (see Appendix A for some formal definitions of these assumptions), they can securely be used as building blocks of more complex protocols. To prove security properties of such protocols, one can then rely on proofs of security already stated for the simpler cryptographic primitives without having to carry out every time a cumbersome and tedious reduction proof from the higher-level protocol down to the cryptographic primitives employed.

As already mentioned in the introduction, a feasible approach to design composable secure protocols is based on simulation-based security, where modular security analyses are performed within universal composability models (UC models; see, e.g., [Can00, PW01, Küs06, KT13]). In these models, the higher-level components of a protocol are designed and analyzed based on lower-level idealized components, called ideal functionalities. Universal composition theorems, such as Canetti's composition theorem [Can00] and Küsters' composition theorems [Küs06], then allow the replacement of the ideal functionalities by their realization. The resulting higher-level components without idealized sub-components enjoy then the desired security properties. It is also possible to show that the higher-level components realize ideal functionalities themselves, to then, in turn, use these ideal functionalities as lower-level idealized components in even more complex protocols.

While ideal functionalities and realization results for some cryptographic primitives mentioned above have already been presented in the literature (see, e.g., [Can01, KT08a, KT09, KT11b, KR17]), these functionalities are formulated in Turing machine models, making tool-assisted analyses of security properties of the protocols based on them impractical.

In this chapter, we formulate in *Java* (more precisely, in Jinja+) ideal functionalities for the cryptographic primitives which most commonly occur in cryptographic applications:

(1) public-key encryption (see Section 4.1),

(2) digital signatures (see Section 4.2),

both with a public-key infrastructure and both handling static corruption,

(3) private symmetric encryption (see Section 4.3),

[9] The IEEE 802.11i protocol is widely known as the "Wi-Fi Protected Access II" (WPA2) protocol.

(4) nonce generation (see Section 4.4).

More precisely, we instantiated the CVJ framework presented in [KTG12a] and extended in Chapters 2 and 3 with the aforementioned functionalities so that they can actually be used to analyze Java programs.[10]

Designing such functionalities and carrying out the proofs (w.r.t. the programming language semantics) is non-trivial and requires some care since the interaction between different classes is much more complex than between Turing machines, where in the former case we have to deal, for example, with exceptions, inheritance, references to potential complex objects that can be exchanged, and hence, the manipulation of one object can affect many other objects. Also, since the ideal functionalities we propose are part of the (Java) programs to be analyzed, they should be formulated in a "tool friendly" way. For example, for this reason, in our functionalities corruption is modeled in a quite different way than it is typically done in the Turing machine models (see, again, [KT08a, KT09, KT11b, KR17]).

For each ideal functionality mentioned above, we propose a corresponding real cryptographic operation also implemented in Java and we prove, in the universal composability model and w.r.t. the Jinja+ semantics, that this real implementation realizes the corresponding ideal functionality under strong cryptographic assumptions. The formulation in Java of these functionalities can be found in Appendices D.2-D.5, with the corresponding (real and ideal) cryptographic library provided in [TSK13].

To illustrate the usefulness and applicability of these proposed cryptographic functionalities, in Section 4.6 we apply the framework along with the tool Joana (see Section 4.5), which allows for the fully automatic verification of noninterference properties of Java programs, to establish cryptographic privacy properties of a (non-trivial) cloud storage application, where clients can store private information on a remote server.

4.1. Public-Key Encryption with a Public Key Infrastructure

We now propose an ideal functionality Ideal-PKIEnc, formulated in Java (Jinja+), for public-key encryption with a public-key infrastructure (PKI). This functionality is an extension of the more restricted public-key encryption functionality proposed in [KTG12a]. First, the functionality proposed here allows a user to encrypt messages for a given party based on the identifier of this party. The functionality uses the included public key infrastructure to obtain the public key of the party registered under the given identifier. In contrast, to encrypt a message, the user of the functionality without a public key infrastructure, have to provide a public-key herself, and hence, take care of the correct binding of public keys to parties herself. Second, in the functionality proposed here, we model *static corruption*, including dishonestly generated keys. For this, special care was needed to make sure that the resulting functionality is "tool-friendly".

We also provide an implementation (realization) of this ideal functionality, denoted by Real-PKIEnc, in Java (Jinja+) and prove, within the CVJ framework, that this implementation realizes the ideal functionality Ideal-PKIEnc under standard cryptographic assumptions.

[10] We note that the CVJ framework as presented in [KTG12a] already supports the functionality for public-key encryption, but without modeling corruption and without a public-key infrastructure.

As already mentioned in the introduction of this chapter, the design of such functionalities and the realization proofs pose additional challenges compared to the Turing machine based formulations proposed in the cryptographic literature.

In the rest of this section, we first provide the interface for Ideal-PKIEnc, and hence, Real-PKIEnc. Then, the actual ideal functionality and its realization are presented, along with a realization theorem.

4.1.1. The Interface for Public-Key Encryption

In this section, we present the interface I_{PKIEnc} of the ideal functionality Ideal-PKIEnc and its implementation Real-PKIEnc and discuss the intended way of using it. The interface I_{PKIEnc} is specified as follows:

```
1  public class Encryptor {
2    public Encryptor(byte[] publicKey);
3    public byte[] encrypt(byte[] message);
4    public byte[] getPublicKey();
5  }
6  public final class Decryptor {
7    public Decryptor();
8    public byte[] decrypt(byte[] message);
9    public Encryptor getEncryptor();
10 }
11
12 // public key infrastructure interface
13 public class RegisterEnc {
14  public static void registerEncryptor(int id, Encryptor encryptor, byte[] pki_domain)
15                                      throws PKIError, NetworkError;
16  public static Encryptor getEncryptor(int id, byte[] pki_domain)
17                                      throws PKIError, NetworkError;
18 }
```

Typical usage. The intended way for an honest user with identifier `ID_A` to create and register her keys is the following:

```
19 Decryptor decryptor = new Decryptor();
20 Encryptor encryptor = decryptor.getEncryptor();
21 try {
22     RegisterEnc.registerEncryptor(ID_A, encryptor, PKI_DOMAIN);
23 }
24 catch (PKIError e) {}      // registration failed: id already claimed
25 catch (NetworkError e) {}  // network problems
```

Intuitively, an object of class `Decryptor` encapsulates a public/private key pair, generated when the object is created (line 19 above). This object provides access to the method `decrypt`. The owner of this object (that is, the party who has created it) is not supposed to share it with any other parties. Instead, the owner of the decryptor shares an associated encryptor (obtained in line 20), which, intuitively, encapsulates only the public key. More precisely, to make her public key available within a PKI to other parties, the user registers the encryptor she has obtained (line 22). That is, she registers her encryptor under her identifier (`ID_A`) and what we call a PKI domain (which is a

publicly known identifier used to distinguish keys registered for different purposes/applications). This step may result in an error: i) if some key has been registered already under this identifier and PKI domain (exception `PKIError`), or ii) if some network failure occurred, e.g., the registration server was unavailable (exception `NetworkError`). We emphasize that we do not require the party who wants to register a public key to provide a proof of possession (PoP) of the private key corresponding to the public key.[11] After an encryptor has been registered, it can be used by other parties as follows:

```
26 try {
27   Encryptor encryptor = RegisterEnc.getEncryptor(ID_A, PKI_DOMAIN);
28   encryptor.encrypt(message);
29 } catch(PKIError e) {}     // id has not been successfully registered
30   catch(NetworkError e) {} // network problems
```

The encryptor of the party registered under `ID_A` and `PKI_DOMAIN` is obtained in line 27 and used in line 28 to encrypt a message. Note that a user can also obtain the public key encapsulated in the encryptor, using the method `getPublicKey`.

Corruption. To model (static) corruption, we allow encryptors also to be created directly, without creating associated decryptors, simply by providing an arbitrary bitstring `pubk` as the public key:

```
31 Encryptor enc = new Encryptor(pubk);
32 try {
33     RegisterEnc.registerEncryptor(ID, enc, PKI_DOMAIN);
34 } catch (PKIError | NetworkError e) {}
```

By this, a dishonest party (the adversary) can register any bitstring `pubk` as a public key, including dishonestly generated keys. This key can then be used by any other party (honest and dishonest) to encrypt messages for the dishonest party, just like public keys of honest parties. Note that since we do not require PoPs, a dishonest party can register any public key of another (possibly honest) party under his identity. (As mentioned before, the literature on PKIs recommends that applications should not rely on PoPs being performed [ANL03].)

An encryptor created in the above way is called *corrupted*. There is no corresponding (corrupted) decryptor, because the adversary can run the decryption algorithm himself. For messages encrypted with a corrupted encryptor (public key), no security guarantees are provided. (Jumping ahead to Section 4.1.2, the functionality will hand the message to be encrypted with a corrupted encryptor directly to the environment/adversary/simulator.)

We note that, as expected, when some party obtains an encryptor by the method `RegisterEnc.getEncryptor`, the party does not know a priori whether the obtained encryptor is corrupted (it has been generated directly) or uncorrupted (it has been generated via `Decryptor`).

4.1.2. The Ideal Functionality for Public-Key Encryption

We now present the ideal functionality for public-key encryption, Ideal-PKIEnc. This functionality provides the interface I_{PKIEnc}, introduced above, to its users (parties, environment) with ideal implementations of the methods declared in I_{PKIEnc}.

[11] In most applications, PoPs are not necessary and as argued in the literature (see, e.g., [ANL03]), applications should be designed in such a way that their security does not depend on the assumption of such proofs being performed.

The functionality Ideal-PKIEnc is defined on top of the interface $I_{\mathsf{CryptoLibEnc}}$ which contains methods for key generation, encryption, and decryption:

```
35 public class CryptoLib {
36   public static KeyPair pke_generateKeyPair();
37   public static byte[] pke_encrypt(byte[] message, byte[] publicKey);
38   public static byte[] pke_decrypt(byte[] ciphertext, byte[] privKey);
39 }
```

The Ideal-PKIEnc expects the above methods to be implemented outside of Ideal-PKIEnc. In the analysis of a system $P[\vec{x}]$ which uses Ideal-PKIEnc (i.e., in the analysis of the system $P[\vec{x}]\cdot$ Ideal-PKIEnc), such methods have to be provided by the environment, and thus, are completely untrusted. In particular, in the analysis of $P[\vec{x}]\cdot$ Ideal-PKIEnc the code for `CryptoLib`, which would typically be very large, does not have to be analyzed. This tremendously simplifies the analysis of $P[\vec{x}]\cdot$ Ideal-PKIEnc (see also the explanation in Section 2.6 and, in particular, the Theorem 2.5).

The basic idea of the implementation of Ideal-PKIEnc is that if a message m is to be encrypted with an (uncorrupted) public key, then not m but a sequence of zeros of the same length as m is encrypted instead, using method `pke_encrypt` of `CryptoLib`. By this, it is guaranteed that the resulting ciphertext c does not depend on m, except for the length of m. The functionality stores the pair (m,c) for later decryption. If some ciphertext c' is to be decrypted, the functionality first checks whether there exists a pair of the form (m',c') (the functionality guarantees that there is at most one such pair). Then, m' is returned as the plaintext. If no such pair exists (and hence, c' was not created using the functionality), c' is decrypted using method `pke_decrypt` of `CryptoLib`, and the resulting plaintext is returned. More specifically, Ideal-PKIEnc works as follows.

On initialization of an object of the class `Decryptor`, a public/private key pair is created by calling the key generation method of the class `CryptoLib`. At this point, the decryptor object also creates an (initially empty) list of message/ciphertext pairs. This list is used as a look-up table for decryption by the method `decrypt` of class `Decryptor` as sketched above.

Encryptors returned by the method `getEncryptor` of class `Decryptor` are objects of the class `UncorruptedEncryptor` (which is a subclass of the class `Encryptor`). An encryptor object contains the same public-key as the associated decryptor and shares (a reference to) the list of message/ciphertext pairs with the associated decryptor. When method `encrypt` of such an encryptor is called with a message m, the encryption method of class `CryptoLib` is called to encrypt a sequence of zeros of the same length as m, resulting in a ciphertext c (ciphertexts seen before are rejected). Then, the pair (m,c) is stored in the list and the ciphertext c is returned as the result of the encryption.

In contrast, a corrupted encryptor (i.e., an encryptor object created directly as in line 31 above, rather than being derived from a decryptor) implements encryptions simply by calling the encryption method of the class `CryptoLib` using the bitstring (the public key) it has been provided with upon creation. Note that in this case, no security guarantees are provided; the original message instead of zeros is encrypted.

The methods for registering and obtaining encryptors in class `RegisterEnc` are implemented in a straightforward way by Ideal-PKIEnc, using a list of registered encryptors along with associated identifiers and domains.

The most important part of the code of Ideal-PKIEnc is listed in Appendix D.2; see [TSK13] for the full code.

4.1.3. The Realization of Ideal-PKIEnc

We now provide the realization Real-PKIEnc of the ideal functionality Ideal-PKIEnc presented above.

The functionality Real-PKIEnc builds on a public key infrastructure. A public-key infrastructure is a trusted public key registry, where i) users can register their public keys under their identifiers and (PKI) domains (in the sense of Section 4.1.1) and ii) users can obtain other users' public keys by providing the identifiers and domains of these users. The interface I_{PKI} for the public key infrastructure used by Real-PKIEnc is the following:

```
40 public class PKI {
41   static void register(int id, byte[] domain, byte[] pubKey)
42                                         throws PKIError, NetworkError;
43   static byte[] getKey(int id, byte[] domain)
44                                         throws PKIError, NetworkError;
45 }
```

The method `register` is supposed to throw `PKIError` if the provided user identifier and domain pair has been claimed already, i.e., some other party has registered a key for the same identifier and domain pair before. The same exception is supposed to be thrown by the method `getKey` if the given identifier `id` has not been registered. Registering or fetching a public key typically involves to contact a public-key server. If this fails, the `NetworkError` is thrown. When proving that Real-PKIEnc realizes Ideal-PKIEnc we will assume that I_{PKI} is properly implemented (see Section 4.1.4 for details).

Now, based on I_{PKI}, the different classes and methods provided by Real-PKIEnc are implemented as presented next.

The methods `registerEncryptor` and `getEncryptor` of the class `RegisterEnc` work as follows. When an encryptor is to be registered by the method `registerEncryptor`, its public key is registered in the PKI using the method `register`. The method `getEncryptor` uses the method `getKey` to fetch the corresponding public key and wraps it into an encryptor which is then returned.

The classes `Encryptor` and `Decryptor` of Real-PKIEnc are implemented in a straightforward way using an encryption scheme: messages are simply encrypted/decrypted directly using such a scheme. Note that whether an encryptor was obtained from a decryptor (using the method `getEncryptor`) or whether it was created directly (as in line 31) leads to the same implementation, namely, invoking the encryption function of the encryption scheme. The only difference is that in one case the public/private key pair was created (honestely) within the class `Decryptor` of Real-PKIEnc and in the other case the public key was created outside of Real-PKIEnc (possibly in some dishonest way).

We refer the reader to Appendix D.2 as well as to [TSK13] for the code of Ideal-PKIEnc and Real-PKIEnc.

4.1.4. Realization Result

We now show that Real-PKIEnc realizes Ideal-PKIEnc, provided that

i) the encryption scheme used in the implementation of Real-PKIEnc is IND-CCA2-secure [BDPR98] and

ii) that the public-key infrastructure used by Real-PKIEnc works “properly”.

As for i), we note that IND-CCA2-security is a standard and widely used security notion for public-key encryption schemes (see Appendix A.1 for its formulation). Similarly to ideal functionality for public-key encryption proposed in the cryptographic literature, it has been shown that IND-CCA2-security is necessary to realize Ideal-PKIEnc (see, e.g., [Can01, KT08a]).

As for ii), the behavior of a “proper public-key infrastructure” is formalized by an ideal functionality Ideal-PKI, which operates in the obvious way: It maintains a list of registration records, each consisting of an identifier, a domain, and a key (the code is given in Appendix D.2). The adversary (simulator) is informed about registration requests and requests for obtaining public-keys and can schedule when these requests are answered by Ideal-PKI (because in a realization such requests typically involve communication over a network controlled by the adversary). We assume the existence of some public-key infrastructure Real-PKI that realizes Ideal-PKI. Note that there are various ways of realizing Ideal-PKI and that all of them will require certain trust assumptions. For example, one could assume the existence of one or more honest certificate authorities and that parties are provided with the (authentic) public keys of these authorities. Typically, one would use some existing public-key infrastructure (with appropriate assumptions) to realize Ideal-PKI. However, this is not the focus of this work. (In fact, proving the security of a full-fledged PKI would be a challenging task by itself.). In our case study (see Section 4.6), we consider a simple realization which involves a single certificate authority, the assumption being that it in fact realizes Ideal-PKI.

With this, we can now state our main theorem for public-key encryption.

Theorem 4.1. *If* Real-PKIEnc *uses an IND-CCA2-secure public-key encryption scheme and* Real-PKI $\leq^{I_{PKI}}$ Ideal-PKI*, then* Real-PKIEnc · Real-PKI $\leq^{I_{PKIEnc}}$ Ideal-PKIEnc.

The proof of Theorem 4.1 is given in Appendix F.4. The proof is highly modular and leverages such properties of the realization relation as the composition theorem, reflexivity, and transitivity. In the proof, we split Ideal-PKIEnc and Real-PKIEnc into two parts: one providing encryption and decryption and one providing key registration and retrieving. For the former part, we generalize the result of [KTG12a] for public-key functionality without corruption and without PKI to the case with corruption.

The IND-CCA2-secure public-key encryption scheme which we employ in [TSK13] is the `RSAES-OAEP` scheme which is part of the PKCS #1 v2.2 family of standards [KJRM16]. It combines the RSA algorithm [RSA78] with the Bellare and Rogaway’s Optimal Asymmetric Encryption Padding (OAEP) method [BR95]. The asymmetric RSA encryption algorithm is used only to exchange a session key among the parties which consecutively encrypt the data using the more efficient AES symmetric encryption scheme. This technique of combining the convenience of a public-key encryption algorithm with the efficiency of a symmetric encryption scheme is usually called “hybrid encryption”.

The `RSA-OAEP` encryption scheme has been proven IND-CCA2-secure by Fujisaki et al. in [FOPS01].

4.2. Digital Signatures with a Public Key Infrastructure

In this section, we propose an ideal functionality Ideal-Sig, formulated in Java (Jinja+), for digital signatures with a public key infrastructure, where, again, we model corruption. We also provide a real implementation Real-Sig of this functionality in Java (Jinja+) and prove, in the CVJ framework, that it realizes Ideal-Sig. Just as for public key encryption, similar functionalities for digital signatures have been proposed in the cryptographic literature before (see, e.g., [Can04, KT08a]). But again, the new contribution here is that we provide a formulation in Java, instead of the (simpler) Turing machine models, such that these functionalities can actually be used to analyze Java programs. This is non-trivial and needs some care. We first present the public interface of Ideal-Sig and Real-Sig.

4.2.1. The Interface for Digital Signatures

The public interface I_{PKISig} of Ideal-Sig and Real-Sig (both have the same public interface) is as follows:

```
1  public final class Signer {
2    public Signer();
3    public byte[] sign(byte[] message);
4    public Verifier getVerifier();
5  }
6  public class Verifier {
7    public Verifier(byte[] verifKey);
8    public boolean verify(byte[] signature, byte[] message);
9    public byte[] getVerifKey();
10 }
11 public class RegisterSig {
12   public static void registerVerifier(int id, Verifier verifier,
13                       byte[] pki_domain) throws PKIError, NetworkError;
14   public static Verifier getVerifier(int id, byte[] pki_domain)
15                                      throws PKIError, NetworkError;
16 }
```

Typical usage. Similarly to public-key encryption, the intended way for an honest user with identifier `ID_A` to create and register her keys is the following:

```
17 Signer sig = new Signer();
18 Verifier ver = sig.getVerifier();
19 try {
20     SigEnc.registerVerifier(ID_A, ver, PKI_DOMAIN);
21 } catch (PKIError e) {}      // registration failed: id already claimed
22   catch (NetworkError e) {}  // network problems
```

Intuitively, an object of the class `Signer` encapsulates a verification/signing key pair, which is generated when the object is created (line 17). It allows a party who owns such an object to sign messages (this requires the signing key), using the method `sign` (of the class `Signer`). This party can also obtain a `Verifier` object (line 18), which encapsulates the related verification key and can be used (by other parties) to verify signatures via the method `verify`. Similarly to the case of

public-key encryption, such a verifier can be registered in the public-key infrastructure (line 20) in order to make the verification key available to other parties. Again, we do not require a proof of possession of the corresponding signing key.

After a verifier has been registered, it can be used by other parties to check whether a signature `signature` is valid for a message `message` w.r.t. the verification key of `(ID_A, PKI_DOMAIN)` encapsulated in `verifier`:

```
23 try {
24   Verifier verifier = RegisterSig.getVerifier(ID_A, PKI_DOMAIN);
25   verifier.verify(signature, message);
26 } catch(PKIError e) {}     // id has not been successfully registered
27   catch(NetworkError e) {} // network problems
```

Corruption. To model (static) corruption, analogously to the case of public-key encyrption we allow verifiers to be created directly, without creating associated signers, simply by providing an arbitrary bitstring `verif_key` as the public key:

```
28 Verifier ver = new Verifier(verif_key);
29 try {
30     RegisterSig.registerVerifier(ID, ver, PKI_DOMAIN);
31 } catch (PKIError | NetworkError e) {}
```

By this, a dishonest party (the adversary) can register any bitstring `verif_key` he wants as a verification key, including dishonestly generated keys. This key can then be used by any other party (honest and dishonest) to verify messages signed by the dishonest party, just like with verification keys of honest parties. Note that since we do not require PoPs, a dishonest party can register any verification key of another (possibly honest) party under his identity. A verifier created in such a way is called *corrupted*. A corresponding signing object is not necessary as the adversary can directly sign messages by himself using the matching signing key (if this key is known to the adversary). Note that, given a verifier object, other parties cannot tell a priori whether this verifier object is corrupted or not.

4.2.2. The Ideal Functionality for Digital Signatures

We now present the ideal functionality for digital signatures, Ideal-Sig. This functionality provides the interface I_{PKISig}, introduced above, to its users (parties, environment) with ideal implementations of the methods declared in I_{PKISig}.

The functionality is defined on top of the interface $I_{\mathsf{CryptoLibSig}}$ which contains methods for key generation, signing, and verification. Analogously to the interface $I_{\mathsf{CryptoLibEnc}}$ for public-key encryption, these methods are supposed to be provided by the environment, and hence, are completely untrusted. In particular, in the analysis of a system that uses Ideal-Sig, they do not have to be analyzed, which, again, greatly simplifies the analysis task.

Now, Ideal-Sig works as follows. On initialization of an object of class `Signer`, a verification/signing key pair is created by calling the key generation operation of the interface $I_{\mathsf{CryptoLibSig}}$. A signer object also creates an (initially empty) list of signed messages; this list will be shared with all associated verifiers (objects returned by `getVerifier`). When the method `sign` is called to sign a message m, the signing procedure of $I_{\mathsf{CryptoLibSig}}$ is called to sign m using the encapsulated

signing key. Before this signature is returned, the signed message m is added to the list of signed messages.

A verifier object returned by the method `getVerifier` belongs to the class `UncorruptedVerifier` (a subclass of the class `Verifier`) and it implements ideal verification as follows: the method `verify`, when called to verify a signature s on a message m, first uses the verification procedure of $I_{\text{CryptoLibSig}}$ to check if s is a valid signature on m w.r.t. the verification key encapsulated in the verifier object. If this is the case, it additionally checks if m is in the list of signed messages (this list, as mentioned before, is shared with the associated signer object). If this is true as well, the method returns 'true'. The idea behind this procedure is that, independently of how the signing and verification algorithms work, the verification of a signature on some message succeeds only if this message has been signed before (and hence, logged) using Ideal-Sig.

A (corrupted) verifier object created directly implements the verification procedure simply by calling the verification method of $I_{\text{CryptoLibSig}}$.

The methods for registering and obtaining verifiers in class `RegisterSig` are implemented in a straightforward way by Ideal-PKIEnc, using a list of registered verifiers along with associated identifiers and domains.

We refer the reader to Appendix D.3 as well as to [TSK13] for the code of Ideal-Sig and Real-Sig.

4.2.3. The Realization of Ideal-Sig

The classes `Verifier` and `Signer` of the realization Real-Sig of the ideal functionality Ideal-Sig are implemented in a straightforward way using a digital signature scheme: messages are simply signed/verified directly using such a scheme. Analogously to the methods in `EncPKI`, the methods `registerVerifier` and `getVerifier` of the class `RegisterSig` are based on the interface I_{PKI} introduced in Section 4.1.3.

The part of the code of Real-Sig for which we provide the realization result is listed in Appendix D.3, while the full library can be found in [TSK13].

4.2.4. Realization Result

We prove that Real-PKISig realizes Ideal-PKISig, provided that

i) the signature scheme used in the implementation of Real-PKISig is EUF-CMA-secure [GMR88] and

ii) the public-key infrastructure used by Real-PKISig realizes the ideal functionality Ideal-PKI (see Section 4.1.4).

Analogously to the case of public-key encryption, it has been shown that EUF-CMA-secure signature schemes are necessary to realize Ideal-PKIEnc (see, e.g., [KT08a]). EUF-CMA-secure schemes are also standard and widely used signature schemes (see Appendix A.2 for its formulation).

Theorem 4.2. *If* Real-PKISig *uses an EUF-CMA-secure signature scheme and* Real-PKI $\leq^{I_{PKI}}$ Ideal-PKI, *then* Real-PKISig $\cdot$ Real-PKI $\leq^{I_{PKIEnc}}$ Ideal-PKISig.

The proof of this theorem is again highly modular and leverages such properties of the realization relation as the composition theorem, reflexivity, and transitivity. The basic structure of the proof is analogous to the one for public-key encryption. We split Ideal-PKISig and Real-PKISig into two parts: i) signing and verification and ii) key registration and retrieving of verification keys. The most involved part is to show that the real component for signing and verification realizes the corresponding ideal component. Here we make use of an existing results in the cryptographic literature, in particular [KT08a], and reduce the statement to a corresponding statement in the Turing machine model. We refer to Appendix F.5 for details.

The EUF-CMA-secure signature scheme which we employ in [TSK13] is the `RSASSA-PSS` scheme which is part of the PKCS #1 v2.2 family of standards [KJRM16]. It is a signature scheme with appendix (SSA) based on the RSA algorithm [RSA78], where the signature of the message is listed next to the message itself. To verify the signature it is necessary to have the message itself. `RSASSA-PSS` is based on the *EMSA-PSS* encoding method which is based on the Bellare and Rogaway's Probabilistic Signature Scheme (PSS) [BR96].

Bellare and Rogaway also proved that the `RSASSA-PSS` signature scheme is EUF-CMA-secure [BR96, Mih98].

4.3. Private Symmetric Encryption

In this section, we present an ideal functionality for what we call *private symmetric encryption* and a realization of this functionality. Private symmetric encryption allows a user to encrypt messages (using a symmetric encryption scheme) just for herself. She does not share the symmetric key with other parties. This is useful, for example, to store confidential information on an untrusted medium. Since keys do not have to be shared between parties, the functionality can be kept quite simple.

The public interface I_{SymEnc} of this functionality and its realization consists of only one class `SymEnc` with two methods: `encrypt` and `decrypt`. These methods use a symmetric key generated when an object of this class is created.

In the ideal functionality Ideal-SymEnc for private symmetric encryption, encryption and decryption work analogously to the case of public-key encryption: a sequence of zeros is encrypted instead of the given plaintext and the ciphertext obtained in this way is logged along with the plaintext, which enables the functionality to recover this plaintext when the ciphertext is to be decrypted. The realization Real-SymEnc simply uses the encapsulated key to encrypt and decrypt messages using a symmetric encryption scheme. Clearly, there is no need to model (static) corruption here: a dishonest party can simply perform private symmetric encryption by himself. We refer the reader to Appendix D.4 as well as to [TSK13] for the code of Ideal-SymEnc and Real-SymEnc.

Based on the implementation of Real-SymEnc and Ideal-SymEnc, we obtain the following result.

Theorem 4.3. *If* Real-SymEnc *uses an IND-CCA2-secure symmetric encryption scheme, then* Real-SymEnc $\leq^{I_{PKIEnc}}$ Ideal-SymEnc.

We omit the proof here because it closely follows the one for public-key encryption only that it

is much simpler now, as we neither need to consider a public-key infrastructure nor corruption (see [KT09] for a corresponding result in a Turing machine model).

4.4. Nonce Generation

In this section, we propose an ideal functionality and its realization for nonce generation, formulated in Java (Jinja+). The property that the ideal functionality is supposed to provide is nonce freshness, i.e., nonces returned by the functionality should always be different to the nonce that have been returned so far (no collisions); unguessability of nonces is not intended to be modeled by this functionality.

The public interface I_{Nonce} for this functionality consists of one class `NonceGen` with one method `newNonce` only, which is supposed to return a fresh nonce.

The ideal functionality Ideal-Nonce for nonce generation works as follows. The functionality maintains an, initially empty, collection (formally, a static list) of nonces that have been returned so far. When the method `newNonce` is called, the environment/simulator is asked to provide a bitstring; more precisely, the method `CryptoLib.newNonce()`, which is supposed to be provided by the environment is called. Then, the method `newNonce` checks whether the returned bitstring is fresh, i.e., whether it does not already belong to the collection of returned nonces. If the nonce is indeed fresh, the nonce is added to the collection and returned to the caller of the method. Otherwise, the above process is repeated until a fresh nonce is returned by the environment/simulator. This guarantees that Ideal-Nonce always outputs a fresh nonce.

In the realization Real-Nonce of Ideal-Nonce, if the method `newNonce` is called, a bitstring of the length of the security parameter is picked uniformly at random and then returned to the caller. More precisely, we assume the method `CryptoLib.newNonce()` called by Real-Nonce to work in this way. We refer the reader to Appendix D.5 as well as to [TSK13] for the code of Ideal-Nonce and Real-Nonce.

Now, it is easy to prove that Real-Nonce realizes Ideal-Nonce.

Theorem 4.4. Real-Nonce $\leq^{I_{\mathsf{Nonce}}}$ Ideal-Nonce.

Proof. (Sketch) To prove this theorem, we let the simulator S work just like Real-Nonce, i.e., when asked to provide a new nonce by Ideal-Nonce, it picks a bitstring of the length of the security parameter uniformly at random and returns this bitstring to Ideal-Nonce. Now, Real-Nonce cannot be distinguished by any (polynomial bounded) environment from $S \cdot$ Ideal-Nonce unless Real-Nonce produces a collision, which, however, happens with negligible probability only. □

4.5. Joana, a Static Checker for proving Noninterference

Although the results of the CVJ framework are very general and not tailored to any specific tool, to show the applicability of the framework, in [KTG12a] the tool *Joana* has been applied to establish cryptographic privacy properties of a simple client-server architecture where messages are sent encrypted over an untrusted the network.

Among the tools for checking noninterference, Joana[12] [HS09a, GHM13] is a static checker for the fully automatic analysis of noninterference properties of Java programs. A user needs to only specify the high and low variables of a program.

Joana is based on the program analysis framework WALA (which stands for "The T. J. WAtson Libraries for Analysis"[13]) which provides static analysis capabilities for Java bytecode. It computes a conservative approximation of the information flow inside the program in form of a *Program Dependence Graph* (PDG). Then, the PDG is checked for illegal information flows using advanced dataflow analysis based on *slicing*, a program analysis technique which consists in the computation of the set of all program statements that, at some point of the execution, may affect the value of a specific variable.

If no illegal flow is found in the PDG, the program is guaranteed to be noninterferent. The correctness of this implication has been verified with a machine-checked proof [Was10] which includes formal specifications of PDGs and the slicing algorithm [WL10, WLS09].

The fully automatic analysis performed by Joana comes at the cost of potential false alarms due to overapproximation. Joana leverages sophisticated flow-, context-, field- and object-sensitive analysis techniques that help to reduce such false alarms, but it does not consider the actual values of variables. For example, whenever a high variable is used in an expression, the value of the expression is considered to contain high information—even if the value of the high variable does not actually influence the result. Therefore, the high level of automation of Joana precludes the possibility to check functional properties of the analyzed programs.

4.6. The Case Study: A Cloud Storage System

As a case study of the results presented in this chapter, we now describe the verification of a cloud storage system implemented in Java. This system illustrates how the ideal functionalities we have developed can be used to analyze an interesting and non-trivial Java program. As already mentioned at the beginning of this chapter, except for the work in [KTG12a], where only a much simpler Java program has been considered, there has been no other work on establishing cryptographic (indistinguishability) properties for Java programs.

In what follows, we first provide a brief description of the cloud storage system program. Then we state the (cryptographic) security property that we verify and, finally, report on the verification process carried out using the tool Joana [HS09b, GHM13], which, as already mentioned, allows for the fully automatic verification of noninterference properties of Java programs.

Design of the Cloud Storage System. We have implemented a cloud storage system that allows a user (through her client application) to store data on a remote server such that confidentiality of the data stored on the server is guaranteed even if the server is untrusted: data stored on the server is encrypted using a symmetric key known only to the client.

More specifically, data is stored (encrypted with the symmetric key of a user) on the server along with a label and a counter (a version number). When data is to be stored under some label, a new (higher) counter is chosen and the data is stored under the label and the new counter; old

[12] The sourcecode of Joana and additional information is available at `http://joana.ipd.kit.edu/`.
[13] `http://wala.sf.net/`.

Store

(1) $C \to S: \; Enc_S(userID, Sig_C[\texttt{STORE}, label, counter, SymEnc_k(message)])$

(2a) $S \to C: \; Enc_C(Sig_S[Sign_C, \texttt{STORE_OK}])$

(2b) $S \to C: \; Enc_C(Sig_S[Sign_C, \texttt{STORE_FAIL}, lastCounter])$

Retrieve

(3) $C \to S: \; Enc_S(userID, Sig_C[\texttt{RETRIEVE}, label, counter])$

(4a) $S \to C: \; Enc_C(Sig_S[Sign_C, \texttt{RETRIEVE_OK}, encryptedMsg, authToken])$

(4b) $S \to C: \; Enc_C(Sig_S[Sign_C, \texttt{RETRIEVE_FAIL}])$

Synchronization

(5) $C \to S: \; Enc_S(userID, Sig_C[\texttt{GET_COUNTER}, label, nonce])$

(6) $S \to C: \; Enc_C(Sig_S[Sign_C, \texttt{LAST_COUNTER}, serverCounter, nonce])$

Figure 4.1.: Messages exchanged between the client and the server, where $Enc_S(m)$ and $Enc_C(m)$ denote m encrypted under the public key of the server and the client, respectively. Analogously, $Sig_S[m]$ and $Sig_C[m]$ denote the signatures of the server and the client, respectively, on m, along with the message m itself. Finally, $SymEnc_k(m)$ denotes m encrypted under the symmetric key k. By $Sign_C$, we denote the signature (not the signed message) of C in the previous message. For example, in (2a) $Sign_C$ denotes C's signature in message (1).

data is still preserved (under smaller counters). Different users can have data repositories on one server. These repositories are strictly separated. The system can be used to securely store any kind of data. A user may use our cloud storage system, for example, to store her passwords remotely on a server such that she has access to them on different devices.

Communication between a client and a server is secured and authenticated using functionalities for public-key encryption and digital signatures. Moreover, the functionality for nonce generation is essential to prevent replay attacks (when the client and the server run a sub-protocol to synchronize counter values for labels).

Implementation of the Cloud Storage System. In our system, data is stored (encrypted) on the server along with a label and a counter (a version number). When data is to be stored under some label, a new (higher) counter is chosen and the data is stored under the label and the new counter; old data is still preserved (under smaller counters). Different users can have data repositories on one server. These repositories are strictly separated. The system can be used to securely store any kind of data. A user may use our cloud storage system, for example, to store her passwords remotely on a server such that she has access to them on different devices.

When created, client and server objects are provided with all necessary key material. In particular, a client object is provided with a user ID and the corresponding public and private encryption and signing keys as well as the symmetric key for encrypting data. The server obtains its public and private encryption and signing keys.

The client class of our system offers two methods: `store` (with parameters *message* (data) and *label*), to store data (*message*) under a chosen label (*label*), and `retrieve` (with the parameter

label), to retrieve data stored under a label (*label*). The client and the server internally maintain the current counter. A counter recorded on the client for a label may differ from the one recorded on the server since, for example, another instance of the client (with the same user ID) may have stored further data on the server meanwhile. Store and retrieve actions therefore always start with a synchronization step (see Figure 4.1, (5) and (6)) where the client asks the server for the current counter for the considered label. If this value is higher than the one stored locally by the client, the client updates its counter to this higher value. If the value is lower, the client throws an exception. The nonce in messages (5) and (6) is used to prevent replay attacks.

Now, when the method `store` is invoked with parameters *message* and *label*, the client object, after having synchronized the counter with the server (see above), sends message (1) in Figure 4.1 to the server, where *counter* is the current value of the counter for *label* obtained after synchronization and k is a private key of the client (not shared with any other party). The client's signature in (1) is stored by the server along with *label*, *counter*, and the ciphertext $SymEnc_k(message)$, and is used later as an authentication token (when retrieving the data). The server may reply with an error message (2b), indicating a counter error (some message has already been stored for the given combination of *counter* and *label*). Otherwise, the server acknowledges that the storage operation was successful (2a).

When the method `retrieve` is called with parameter *label*, the client sends, again after synchronization with the server, message (3) to the server, where *counter* is the current value of the counter for *label* after synchronization. The server can, again, respond with an error message (4b) (indicating that there is no message stored under the given combination *label*/*counter* for that user), or it responds with the message (4a), containing the encrypted data *encryptedMsg* stored under *label*/*counter* and an authentication token *authToken* (see above), which proves to the client that the response of the server is correct.

The code of the client can be found in Appendix E.1, while the full code of the system is available in [STG13].

The Security Property. As mentioned above, the most fundamental security property of the cloud storage system is confidentiality of the stored data. This property is supposed to be guaranteed even if the server and all clients of other users may be dishonest and cooperate with an active adversary.

To formulate this confidentiality property, we provide (besides the code of the client and the server) a setup class with the method `main`, which gets a `secret_bit` as input. This method models the interaction between the program of an honest client and the active adversary (the environment).

The setup program takes the parameter `secret_bit` of type `boolean` as its input. This program, first, creates a client (i.e. an object of class `Client`) and registers her public-key encryption and signing keys in the public-key infrastructure. If this registration process succeeds, the setup program enters its main loop where the adversary (the environment) determines, one by one, the actions to be taken by the system by sending instructions to `main`. Except for the first instruction, the following instructions can be sent by the adversary arbitrarily often.

– The adversary can decide to end the loop by sending a special end instruction.

– The adversary can register a corrupted encryptor and/or a corrupted verifier. In particular, he

can register such objects under the fixed identifier of the server. By this, the adversary is able to fully subsume (impersonate) the server: he can decrypt messages encrypted for the server and produce signatures of the server. Analogously, the adversary can register (and then subsume) dishonest clients. Note that the adversary cannot register keys under the ID of the honest client created at the beginning of the setup, because this ID is already taken.

– The adversary can pick an arbitrary label and an arbitrary message to be stored by the honest client on the server, by calling `client.store(label,message)`. More precisely, to do so, the adversary, besides the label, provides a pair (m_0, m_1) of messages of the same length (if the two messages do not have the same length, this step is aborted). Then, depending on the value of the `secret_bit`, m_0 or m_1 is picked as `message` (the message to be stored).

– The adversary can choose to have the honest client retrieve a message for a given label (again, determined by the adversary), using `client.retrieve(label)`. The value of the returned message is then ignored. However, notice that this step, according to the honest client program, triggers an exchange of messages between the client and the server (adversary), which includes the encrypted message.

By this, the adversary has full control over the network and over the actions taken by the honest client. Moreover, it subsumes both the server and all dishonest clients.

The security property now requires that no (probabilistic polynomial-time) adversary is able to determine the `secret_bit`, and hence, whether the data items in the first or in the second component of the item pairs provided by the adversary are sent by the client. This specifies a strong cryptographic privacy property, common in cryptography. Formally, this indistinguishability property is stated as follows:

$$\mathsf{CS}_R[\mathtt{false}] \approx^{\emptyset}_{\mathsf{comp}} \mathsf{CS}_R[\mathtt{true}] \tag{4.1}$$

where $\mathsf{CS}_R[\mathtt{secret_bit}]$ denotes the described system, consisting of the setup class and the client class (see Appendix E.1 for the full code). The index R indicates that in this system the cryptographic operations are carried out using the real cryptographic schemes (rather than ideal functionalities).

We note that the computational indistinguishability relation in (4.1) uses the empty interface $I = \emptyset$. This means that the adversary (environment) cannot directly call methods of the client object. As explained before, by the definition of the setup class, the environment can nonetheless determine which actions are taken and when. We also point out that CS_R is an open system which uses some classes not defined within CS_R, such as a network library. These classes are provided by the environment and, therefore, are untrusted. Thus, property (4.1) implies confidentiality of the stored messages no matter how such untrusted libraries are implemented.

Verification of the Security Property. In order to prove (4.1), by Theorem 2.2 it suffices to show that

$$\mathsf{CS}_I[\mathtt{secret_bit}] \text{ is } I\text{-noninterferent,} \tag{4.2}$$

where CS_I denotes the system which coincides with CS_R except that the real cryptographic schemes are replaced by their ideal counterparts (ideal functionalities), i.e., Ideal-PKEnc, Ideal-Sig,

Ideal-SymEnc, and Ideal-Nonce. Since, as can easily been seen, $\mathsf{CS}_I[\mathtt{secret_bit}]$ satisfies the conditions of Theorem 2.5, we can further reduce checking (4.2) to checking the following property:

$$\tilde{E}_{\vec{u}} \cdot \mathsf{CS}_I[\mathtt{secret_bit}] \quad \text{is noninterferent for all } \vec{u}, \tag{4.3}$$

where the family of systems $\tilde{E}_{\vec{u}}$, parameterized by a finite sequence of integers $\vec{u}$, is as described in Section 2.6. This system can be automatically generated from $\mathsf{CS}_I[\mathtt{secret_bit}]$. Also note that by "noninterference" we mean standard termination-insensitive noninterference (see Section 2.4). Altogether it suffices to prove (4.3) in order to obtain (4.1).

Joana is easily able to establish property (4.3). It takes about 17 seconds on a standard PC (Core i5 2.3GHz, 8GB RAM) to finish the analysis of the program (with a size of 950 LoC). Note that the actual running code of the distributed system is much bigger than what Joana needs to analyze, because the code of the distributed system includes untrusted libraries, such as the standard Java library for networking, which do not need to be analyzed, as already mentioned above.

5. Related Work and Discussion

This part of the thesis contributes to the effort to develop techniques and methodologies in order to prove security of implementations of cryptographic protocols.

Obtaining cryptographic guarantees for programs written in real-world programming languages is a challenging and quite recent research field. In fact, in most of the work on language-based analysis of cryptographic software the analysis is carried out based on a symbolic (Dolev-Yao) model, without computational/cryptographic guarantees (see, e.g., [GP05, CD09, BFG10]). The very few approaches aiming at cryptographic guarantees typically adopt one of the following:

1) They rely on symbolic analysis and then apply computational soundness results (see, e.g., [BMU10, AGJ11]).

2) They derive formal models from the source code and analyze these models using specialized tools for cryptographic verification, such as the tool CryptoVerif [Bla06] (see, e.g., [BJST08, AGJ12]).

3) They derive source code from formal specifications (see, e.g., [CB12, CB13]).

An approach similar to the approach taken by the CVJ framework is the work by Fournet et al. [FKS11] which aims at establishing computational indistinguishability properties for a fragment of F# [Don15], a functional-first programming language appositely designed for verification purposes. This approach is strongly based on type checking. Using this framework, Bhargavan et al. [BFK$^+$13] propose a *modular* implementation of the TLS protocol in F# and verify its computational security using the typechecker F7 [BBF$^+$08] for F#.

Other more recent works use a newly proposed programming language called F* [SHK$^+$16], a dialect of ML which bootstraps in both OCalm and F# and which is equipped with a verification system based on dependent and refinement types.[14] Security protocols such as TLS, multi-party sessions, web-browser extensions, and other cryptographic constructions written in F* can be mechanically proven secure using the typechecker and the SMT solver provided within this language (see, e.g., [CB13, BFK$^+$13, BFG$^+$14, DFK$^+$17] for some works using this approach to verify cryptographic implementations).

The CVJ framework, in contrast, aims at using existing program analysis tools and techniques to directly obtain cryptographic security guarantees for systems coded in Java, a programming language not specifically designed for verification purposes. It is the only approach for the cryptographic analysis of Java programs and it establishes general results for ideal functionalities and their realizations in the universal composability paradigm [Can00, PW01, Küs06, KT13]. Depending on the cryptographic application to be verified, one can instantiate the CVJ framework

[14] A dependent type is a type whose definition depends on a value derived from an algebraic or a logical formula. A refinement type is a type of the form $x{:}t\{\varphi\}$, where the refined type is the sub-type of t which is restricted to those expressions $e{:}t$ for which the logical formula $\varphi[e/x]$ is valid.

with the required cryptographic primitives, as we have, for example, done in Chapter 4. In this regard, we notice that universally composable functionalities for public-key encryption, digital signature, private symmetric encryption, and nonce generation have already been proposed in the literature (see, e.g., [KT08a, KT09]), but in a more generic Turing machine model rather than in a practical programming language.

Finally, we note that the CVJ framework is not tailored to any specific tool: static checkers, theorem provers, typesystems, or even manual proofs are, in principle, all valid possibilities for establishing noninterference properties. However, to check noninterference properties of our case studies, we primarily employ the static checker Joana. Among the tools for checking (standard) noninterference of sequential Java programs, Joana is based on program dependency graphs (PDGs) [HS09a]. Other tools, like, for instance, JIF [Mye99, MCN+01], perform information flow analyses based on type systems. Others, like, for instance, KeY [ABB+05, ABB+14, BHS07], are based on theorem proving and can check noninterference properties based on the technique of self-composition (see Section 9.3 for more details on the KeY tool).

Chapter 3 contributes to the field of establishing confidentiality properties for concurrent programs, extensively used, for instance, in operating systems and client-server applications.

In the context of language-based information-flow security, confidentiality properties are usually defined by noninterference polices, all referring to the notion of noninterference for confidentiality proposed by Goguen and Meseguer [GM82a] which requires the absence of information flowing from private (high) inputs to public (low) variables. This notion has been extended in various ways to address concurrent programs too (henceforth, we refer to noninterference for concurrent programs also as "concurrent noninterference").

The first major line of research formally defining concurrent noninterference lies on the notion of *possibilistic noninterference*. Roughly speaking, a concurrent program is possibilistic-noninterferent if in all its possible executions (i.e., executions where the spawned threads are interleaved differently by the scheduler) the values of the private (high) variables do not affect the values of the public (low) ones. Many definitions of possibilistic noninterference have been proposed in the literature along with different techniques and criteria to enforce them (see, e.g., [McL96, Man00, BC02, BDR04]). However, depending on what type of channels one accepts as capable of transmitting leaks, a program which is possibilistic noninterferent may still be vulnerable to:

1. *Probabilistic Attacks*, which may occur by running the program multiple times and by then gathering statistical information over the private data from the distribution of the public outcomes.

2. *Refinement Attacks*, which may occur by running the program under a predetermined scheduler (e.g., an uniform one) in order to exclude a range of possible (public) outcomes and then infer more precise information over the private data.

3. *Timing Attacks*, which may occur by running the program concurrently with an adversary able to infer information on private data by measuring the runtime of its computation.

4. *Termination Attacks*, which may occur by running the program concurrently with an adversary able to infer information on private data based on whether the computation of the program

terminates or not after a certain (fixed) number of steps. We note that termination attacks are quite often considered to be a special case of timing attacks.

To address these typologies of attacks, concurrent noninterference definitions have been lifted to be *timing/termination-sensitive* [VS97, Sab01], *probabilistic* [SV98, VS99, SS00], and *scheduler-independent* [ZM03, MS10, PHN13], where these different proposed definitions are quite often overlapping and usually coping with more than one typology of attack at a time. In a prominent systematization of knowledge work on language-based information-flow security, Sabelfeld and Myers [SM03] present a quite comprehensive overview of the variety of noninterference definitions, both for sequential and concurrent programs, discussing their relations and their limitations on the type of attack they are able or not to capture. More recent works, such as those by Popescu et al. [PHN12, PHN13], cast in a single framework the different notions of concurrent noninterference (possibilistic, probabilistic, and scheduler-independent) studying their relation and under which assumptions one implies another.

The notion which is more and more becoming the reference definition for concurrent noninterference is *probabilistic noninterference* [SV98, VS99, SS00, Sab03, Smi06]. This notion is typically stated by considering the concurrent program as a random variable over all its possible executions (traces) and by then assigning a probability on each trace caused by each possible input vector $\vec{x}$. In this respect, a program is considered to be probabilistic-noninterferent if the sums of all the probabilities on all the executions possibly generated by *each pair* of "low-equivalent input vectors" (i.e., vectors whose public values are equivalent) are the same. We refer, for instance, to [GS15] or to [BGH$^+$16] for such a formal definition. On the positive side, we note that this notion is intrinsically scheduler-independent, as it already considers all possible thread interleavings. On the negative side, criteria to establish probabilistic noninterference are usually too strict to be enforced on real-world programs. The most prominent criterion for establishing this property, namely the *Low-Security Observational Determinism* (LSOD) [RWW94, ZM03], has been implemented in the Joana verification tool [GS15]: However, even relaxed [GS15] and improved [BGH$^+$16] versions of this criterion are still too strict for being employed in the fully automatic verification of any realistic multi-threaded Java program (see, e.g., the case study in [BGH$^+$16]). Therefore, in some works the notion of probabilistic noninterference has been relaxed by providing so-called *scheduler-independent noninterference* definitions which, despite the name, consider only a suitable class of schedulers, as, for example, the class of "robust schedulers" [MS10] or of "noninterfering schedulers" [PHN13]. In this regard, we notice that the proposed classes of schedulers always try to capture as many well-known schedulers as possible, such as the uniform and the Round-Robin scheduler.

The variety of the different formalizations of concurrent noninterference suggests that there might not be a definition which is suitable for every purpose. Instead, it seems that the choice of such a definition depends on the particular application and/or on the specific runtime environment where the program is supposed to be executed. In this context, our proposed notion of concurrent noninterference for SyncJinja+ programs (Definition 3.21 in Section 3.6) is another way to express confidentiality at the language level in case of a (Java) program executing concurrently.

Compared to the aforementioned notions of concurrent noninterference, our proposed definition is:

i) a natural extension for multi-threaded programs of the noninterference notion already stated

for sequential programs which requires the absence of information flowing from private (high) inputs to public (low) outputs [GM82a];

ii) tailored to a specific programming language, namely Java (more precisely, to the fragment of Java falling into the SyncJinja+ language), and neither formulated for an abstract computational model as in [MS10, PHN13] nor for a relatively simple concurrent language as in [SS00, Sab03];

iii) made to capture both refinement and timing attacks (see items 2. and 3. above, respectively), since neither the deterministic scheduler $\mathscr{S}$ (see Definition 3.21) nor the deterministic adversary/enviroment E (see Definitions 3.22) has a predefined semantics; instances of $\mathscr{S}$ and E could therefore measure the runtime of the honest program P and/or execute P in such a way to infer information over its private data;

iv) less strict than the other considered notions because it does not impose any simulation, bisimulation, or equivalence among the possible executions (traces) of the concurrent program.

In particular, regarding the latter point, we notice that our proposed definition is less strict than the aforementioned low-security observational determinism (LSOD) criterion for checking probabilistc noninterference [ZM03]: While the definition of LSOD requires that, for a program which runs on two low-equivalent inputs, all possible traces are *stepwise* low-equivalent, our proposed definition is less strict in the sense that it only requires that, *at the end of the runs*, the value of the low variables are the same. On the positive side, this means that, compared to probabilistic noninterference, our proposed definition should, in principle, hold for a more reasonable class of multi-threaded programs. On the negative side, we neither capture termination attacks (see item 4. above), as our notion is termination-insensitive, nor probabilistic attacks (see item 1. above), as all the components involved, namely the honest program P, the scheduler $\mathscr{S}$, and the environment E, are considered to be deterministic.

Regarding termination attacks, we however note that a termination-insensitive notion of noninterference is necessary in case we want this property checked by automatic tools.

Regarding probabilistic attacks, we note that the aim behind such a definition has been to propose a suitable notion of concurrent noninterference to then formally link it to the notion of computational indistinguishability, which, as all the notions stated in cryptography, considers probabilistic systems.

Our proposed definition of a scheduler (Definition 3.1 in Section 3.2.2) also differs from most of the notions of schedulers proposed in the literature so far (see, e.g., [MS10, PHN13]). More precisely, our definition is more general in the sense that it neither imposes a specific semantics on the scheduler nor on the classes of schedulers it models (uniforms, round-robins, and so on). Insted, we restrict the scheduler as follows:

– The class of programs the scheduler belongs to is the class of probabilistic polinonomially bounded programs and its degree of parallelism is one. That is, we define the scheduler as a bounded, possibly randomized single-threaded Jinja+ program.

– The quantity of information exchanged with the scheduled program is limited to the set of threads which can execute the next step of computation (the so-called “active threads”, see Definition 3.2).

As in the case of our proposed concurrent noninterference definition, our definition of scheduler adds to the landscape of the different formalizations of runtime environment (i.e., the environment where the concurrent program is supposed to be executed) another formalization of what the scheduler is and of the quantity of information it exchanges with the scheduled program.

PART II

sElect, a Lightweight Verifiable Remote Voting System

6. E-voting Systems and their Security Properties

Systems for electronic voting (e-voting systems) have been used in many countries for national or municipal elections, for example, in the US, Estonia, India, Belgium, Switzerland, and Brazil, as well as for so-called low-stake elections, i.e., elections within associations, societies, and companies. There are two main categories of such systems. In the first category voters have to go to a polling station in order to cast their vote either by filling in a paper ballot, which then is scanned by an optical scan voting system, or by entering their vote directly to an electronic voting machine, usually referred to as Direct-Recording Electronic (DRE) voting machines. In the second category, called remote electronic voting, voters vote over the Internet using their own devices (e.g., desktop computers or smartphones). In addition, there are hybrid approaches, where voters, via an additional channel, such as mail, are provided with codes which they then use to vote (code voting).

In this part of the thesis we focus on remote electronic voting, i.e., on Internet-based voting systems. Compared to the other online services like e-banking or e-commerce, e-voting raises additional challenges in both designing such systems and guaranteeing their security requirements. This is mainly deriving from the following two issues:

– The need to verify the accuracy of the election while simultaneously providing the secrecy of the ballots.

– The practical infeasibility of designing remote e-voting systems which truly protect against voter coercion.

Unfortunately, most of the e-voting systems used in practice today do not provide a sufficient level of security w.r.t. these two issues. In particular, voters have no guarantee that their votes have been properly counted. Moreover, since e-voting systems are complex hardware/software systems, both programming errors and security vulnerabilities can hardly be avoided: As for any software/hardware systems, to guarantee the security of an e-voting system one has to defend the system against all possible attacks, while it is sufficient for an attacker to find out one specific vulnerability to manipulate an election undetectably. In addition, these systems might deliberately be tampered with when deployed in elections. That is, when using e-voting systems, voters generally do not have any guarantees that their votes were actually counted and that the published result is correct, i.e., reflects the actual voters' choices. In this respect, as also described in [KT14], numerous problems with e-voting systems employed in legally binding elections have been reported by the press. For instance, in [Com12] it has been described that in 2001, during the Democratic primary elections in New Jersey's Cumberland Country, a DRE voting machine attributed votes to the wrong candidates due to programming errors and ended up declaring the actual losers as winners of the election. Another example where attackers meddled in an high-stake election possibly interfering with its outcome is the more recent 2016 United States presidential election: several newspapers alluded to the actual possibility that Russian hackers

altered the results of this election so much that they have in fact determined the final winner (see, e.g., [New16] and [New17]).

These are just two examples of a long list of reports on miscounted votes or attacks on voting machines due to server misconfigurations, hardware or software problems, or programming errors. Moreover, the fact that voting systems can easily be manipulated by insiders and/or external attackers has been demonstrated on various systems, for example on the Estonian e-voting system [SFD+14, PCW14]. Again, such manipulations can often even allow attackers to undetectably change the overall election results.

To address these problems, in the last decade or so there has been an intensive research effort in formally defining the security properties e-voting systems should provide in order to then systematically analyze such systems.

Privacy. The first security property that e-voting systems should provide is *voters' privacy* which informally means that no one should be able to tell how each voter voted, at least with overwhelming probability (see, for instance, [KTV11] for a formal definition of privacy).

Verfiability. Besides voters' privacy, modern e-voting systems strive for what is called *verifiability*. This notion informally means that voters and possibly also external authorities should be able to check both that the votes were properly counted and the integrity of the overall election. Importantly, it should be possible to perform these checks even if one or more components of the e-voting system has/have programming errors, misconfigurations, security vulnerabilities, or is/are partially malicious.

While several (overlapping) definitions of verifiability have been proposed in the literature,[15] the connotation of this security property is usually determined by three different notions:

i) *Individual verifiability* [SK95] refers to the possibility for each voter to check whether or not her vote has been properly cast, collected, and counted.

ii) *Universal verifiability* [SK95] requires the possibility for any voter or interested auditor to verify the integrity of the overall election result, even if voting systems/authorities are (partially) untrusted.

iii) *End-to-end verifiability* [Jos87] more generically asserts that if some party, such as voting authorities, deviates from the (e-voting) protocol in a "serious way", then this deviation is noticed by honest participants with high probability.

Since end-to-end verifiability is the most generic among the three aforementioned notions, this property is usally the most desirable for e-voting systems to provide. Traditionally, the standard technique to obtain end-to-end verifiability is to establish both individual and universal verifiability, as it has been done in many prominent e-voting systems, such as Helios [Adi08] and Scantegrity [CCC+08]. However, in [KTV11] it has been demonstrated that the combination of individual and universal verifiability does not necessarily guarantee end-to-end verifiability. In particular, it is possible to build e-voting protocols which can achieve an high level of end-to-end

[15] In a recent systematization of knowledge paper [CGK+16], most of the formal definitions of verifiability proposed in the literature are reviewed, cast in a single framework, and detailed compared.

verifiability without universal verifiability, as it has been shown in [KMST16a] with the sElect e-voting system.

Accountability. Recent studies have shown that *accountability*, an even stronger property than verifiability, is desirable: Accountability not only requires that ballot manipulations is detected but also that the misbehaving parties are singled out and correctly blamed. As demonstrated in [KTV10b], verifiability is a weaker (and often too weak) form of accountability. For verifiability, one requires only that, if some goal of the e-voting protocol is not achieved (e.g., the election outcome does not correspond to how the voters actually voted), then at least one involved entity will not accept such a run (i.e., the instantiated election process), but it is not required to blame misbehaving parties. In contrast, accountability requires that misbehaving parties are also properly blamed. It is very important that an e-voting scheme provides accountability, not only verifiability. As verifiability is implied by accountability, it suffices to directly focus on accountability. In practice, accountability requires two conditions to be satisfied:

– *fairness*, the protocol participants who are honest are never blamed;

– *completeness*, if, during a run, some desired goal of the protocol is not met — due to the misbehavior of one or more protocol participants — then these participants who misbehaved are singled out and correctly blamed.

As in the case of verifiability, the desired goal for voting protocols is that the published result of the election corresponds to the actual votes cast by the voters. In this respect, the completeness condition guarantees that if in a run of the protocol the published result of the election does not correspond to the actual votes cast by the voters (a fact that must be due to the misbehavior of one or more protocol participants), then one or more participants are held accountable; the fairness condition guarantees that these participants are *rightly* held accountable.

Paper-based elections already include pragmatical mitigations to ensure some form of verifiability, ranging from external observers which can watch the polling station throughout the election process and scrutinize the count afterwards to rigorous auditing conducted transparently and under observation. In contrast, e-voting systems typically enforce verifiability and accountability in the following way: when voters cast their ballot, they are provided with some kind of receipt which they can then use in combination with additional data published by the system besides the election result to check that their votes were properly counted. However, since the use of receipts could in principle brake voters' privacy, further measures in designing such systems have to be taken into account to ensure that, even in the presence of receipts, other parties would not be able to tell how each voter voted, at least with some overwhelming probability.

Coercion-Resistance and Receipt-Freeness. Besides privacy and verifiability/accountability, some e-voting systems also try to prevent vote buying and voter coercion, aiming at providing another security property called *coercion-resistance* (see [KTV12a] for a formal definition of this property). Intuitively, an e-voting system achieves coercion resistance if a coercer cannot tell whether a voter cast her ballot following his instructions or not. In this respect, as also reported in [CCFG16], coercion-resistance can only be pragmatically enforced by giving the possibility for revoting, which is impractical in many scenarios.

Clearly, coercion-resistance can easily clash with the way verifiability and accountability is typically achieved: once the voter obtains some sort of receipt to check that her vote has been properly counted, the coercer could simply obtain this receipt from the voter to make sure the voter voted as requested. Given that designing remote voting systems which are truly coercion-resistant is practically impossible (a coercer can, in principle, always threaten a voter at the moment of using the voting device to cast her choice), some approaches aim instead at some form of *receipt-freeness* (see, e.g., [BT94, SK95, Oka97, MN06]). Receipt-freeness ensures that a voter cannot prove to anyone how she voted. In this respect, receipt-freeness can be seen as a weaker form of coercion-resistance.

Overall, the design of practical e-voting systems is very challenging as many aspects and different, somehow orthogonal security properties have to be taken into account. In addition, one has to find a good balance between simplicity, usability, and security. This in turn very much depends on various, possibly even conflicting requirements and constraints, such as: What kind of election is targeted? National political election or elections of much less importance and relevance, e.g., within clubs or associations? Should one expect targeted and sophisticated attacks against voter devices and/or servers, or are accidental programming errors the main threats to the integrity of the election? Is it likely that the voters are coerced, and hence, should the system defend against coercion? How heterogeneous are the computing platform of voters? Can voters be expected to have or to use a second (trusted) device and/or install software? Is a simple verification procedure important, e.g., for less technically inclined voters? Should the system be easy to implement and deploy, e.g., depending on the background of the programmers? Should authorities and/or voters be able to understand (to some extend) the inner workings of the system?

Since there does not seem to exist a "one size fits all" e-voting system, several different remote voting systems have been proposed in the literature, especially those designed for low-risk elections, such as elections within clubs and associations: for example, the prominent e-voting system Helios [Adi08], but also Prêt à Voter [RBH+10], STAR-Vote [BBB+13], and Remotegrity [ZCC+13], are designed to achieve vote privacy and verifiability, but do not offer concrete protection against vote buying and voter coercion. Some other systems, such as Civitas [CCM08] and Scantegrity [CCC+10], are designed to also achieve some level of coercion-resistance, i.e., they require that vote selling and voter coercion is prevented. Several of these systems have been used in legally binding elections (see, e.g., [AdMPQ09, CCC+10, CRST14]).

In this part of the thesis, we propose a new practical and verifiable e-voting system, called sElect (secure/simple elections). This system is meant for low-risk elections and it is designed to be particularly simple and lightweight in terms of its structure, the cryptography it uses, and the user experience.

7. The sElect E-voting System and its main features

sElect (secure and simple elections) is a remote e-voting system, which we implemented as a platform independent web application[16] and for which a detailed cryptographic security analysis has been performed in [KMST16a] with respect to privacy of the votes as well as verifiability and accountability. The system combines several concepts, such as verification codes (see, e.g., [DLM82]) and Chaumian mix nets [Cha81], in a novel way.

7.1. sElect in a nutshell

The main components of sElect are a *voting platform* (for which we propose an implementation as a static web-page), a *collecting server*, and a cascade of *mix severs*. Moreover, all the data outputted by these servers, such as the lists of voters, the intermediate, and the final results, are collected by a publicly available *bulletin board*.

In the voting phase, every voter prepares her ballots using the voting platform. A ballot contains the voter's choice (for example, the name of the candidate chosen by the voter) and a verification code, which is randomly generated by the system using cryptographic nonces. The voting platform encrypts the choice along with the verification code several times with the public key of each mix server, from the last to the first. The encrypted ballot is then submitted to the collecting server which authenticates the voter and, if the authentication succeeds, replies by sending back a digitally signed acknowledgment.

When the voting phase is over, the system enters the mixing phase. In this phase, the collecting server outputs the list of ciphertexts to the first mix server which decrypts the outer encryption layer, shuffles the inner ballots, and sends the signed result to the next mix server. Next, the bulletin board reads the list of (unencrypted) ballots produced by the last mix server. It then publishes the resulting list containing the voters' choices along with the verification codes, in alphabetical order and digitally signed. This list constitutes the official result of the election.

sElect is designed to provide privacy of the votes, as well as verifiability and accountability.

(i) *Privacy* is provided under the assumption that at least one of the mix servers is honest. The steps taken by an honest mix server, by design, hide the link between its input and output entries. Therefore, no one can associate the ballot of a given voter to her choice/identifier pair in the final output.

(ii) *Verifiability* is achieved in a very direct way: once the result has been published, every voter can simply check whether her verification code is included in the published election result, along with her choice. For this mechanism to work, one needs to make sure that the verification code is indeed randomly chosen and hence unique.

[16] A demo version of the system is available at `https://select.sec.uni-stuttgart.de`.

(iii) *Accountability* is provided by a fully automated verification procedure implemented within the voting platform which does not require any user interaction and is triggered as soon as the voter looks at the election result: cryptographic checks are performed and, if a problem is encountered, the specific misbehaving party is singled out and binding evidence of this misbehavior is produced to hold him accountable.

sElect is however not meant to defend against coercion and mostly tries to defend against untrusted or malicious authorities, including inadvertent programming errors or deliberate manipulation of servers, but excluding targeted and sophisticated attacks against voters' devices.

7.2. Main features of sElect

We now sketch the main features of sElect, including several novel and unique features and concepts which should be beneficial also for other systems. Besides the technical account of sElect provided in the following paragraphs, a general discussion on sElect, including its comparison with Helios, is provided in Chapter 10.

Fully automated verification. One of the important unique features of sElect is that it supports fully automated verification. This kind of verification is carried out by the voter's browser. It does not require any voter interaction and is triggered as soon as a voter looks at the election result. This is meant to increase verification rates and ease the user experience. As voters are typically interested in the election results, combining the (fully automated) verification process with the act of looking at the election result in fact appears to be an effective way to increase verification rates as indicated by two small mock elections we performed with sElect (see Section 8.2). In a user study carried out in [AKBW14] for various voting systems, automated verification was pointed out to be lacking in the studied systems, including, for example, Helios. It seems that our approach of automated verification should be applicable and can be very useful for other remote e-voting systems, such as Helios, as well. Another important aspect of the automated verification procedure of sElect is that it performs certain cryptographic checks and, if a problem is discovered, it singles out a specific misbehaving party and produces binding evidence of the misbehavior. This provides a high level of accountability and deters potentially dishonest voting authorities.

Voter-based verification (human verifiability). Besides fully automated verification, sElect also supports a very easy to understand manual verification procedure: a voter can check whether a verification code she has chosen herself when casting her vote appears in the election result along with her choice. As further discussed in Chapter 10, this simple procedure has several obvious benefits. For example, it reduces trust assumptions concerning the voter's computing platform (for fully automated verification the voter's computing platforms needs to be fully trusted). Also voter's can easily grasp the procedure and its purpose, essentially without any understanding of the rest of the system, which should help to increase user satisfaction and verification rates. On the negative side, such codes open the way for voter coercion (see also Chapter 10).

Simple cryptography and design. Unlike other modern remote voting systems, sElect uses only the most basic cryptographic operations, namely, public key encryption and digital signatures.

And, as can been seen from Section 8.1, the overall design and structure of sElect is simple as well. In particular, sElect does *not* rely on any more sophisticated cryptographic operations, such as zero-knowledge proofs, verifiable distributed decryption, universally verifiable mix nets, etc. Our motivation for this design choice is twofold.

Firstly, we wanted to investigate what level of security (privacy, verifiability, and accountability) can be obtained with only the most basic cryptographic primitives and a simple and user-friendly design, see also below.

Secondly, using only the most basic cryptographic primitives has several advantages (but also some disadvantages), as discussed in Chapter 10.

Design-level cryptographic security analysis. In [KMST16a], a rigorous cryptographic analysis of the voting protocol of sElect has been performed with respect to end-to-end verifiability, accountability, and privacy. Since quite rarely implementations of practical e-voting systems come with a rigorous cryptographic analysis, this is a valuable feature by itself.

More precisely, the sElect voting protocol has been formally modeled in [KMST16a] based on a general computational model which follows the one in [KTV10b, KTV11]. Then, based on generic notions of (end-to-end) verifiability, accountability, and vote privacy (the first two formally defined in [KTV10b], while the latter in [KTV11]) the level of security of the sElect voting protocol has been formally established w.r.t. these three security properties: the cryptographic analysis of sElect shows that its protocol enjoys a good level of security, given the very basic cryptographic primitives it uses.

Remarkably, the standard technique for achieving (some level of) end-to-end verifiability is to establish both individual and universal verifiability. In contrast, in [KTV11] it has been shown that the combination of individual and universal verifiability does not guarantee end-to-end verifiability. Instead, sElect demonstrates that one can achieve (a certain level of) end-to-end verifiability, as well as accountability, without universal verifiability. This is interesting from a conceptual point of view and may lead to further new applications and system designs.

Code-level analysis of privacy properties. Since confidentiality of the votes is the most important security property an voting system is supposed to provide, besides establishing this properties at the protocol level, one would prefer that e-voting systems come with privacy guarantees also on their implementation.

As we further describe in Chapter 9, we employed the CVJ framework introduced, instantiated, and extended in Part I in combination with a hybrid approach to prove noninterference (see Section 9.2 and [KTB+15]) to guarantee that the implementation of sElect provides strong cryptographic vote privacy. In order to establish noninterference properties directly on the implementation level, we integrate the high-level of automation provided by static checkers, such as the Joana tool [HS09a, GHM13], with the precision provided by theorem provers, such as the KeY verification system [ABB+05, ABB+14, BHS07].

sElect is then the first full-fledged remote voting system running a cryptographic component for which strong security guarantees have been formally established directly on *code-level* to ensure confidentiality of the cast votes.

8. Design, Implementation, and Deployment of the sElect E-voting System

In this chapter, we describe the components (Section 8.1), the development (Section 8.2), and the deployment (Section 8.3) of the sElect e-voting system. In particular, in the latter section we present two field tests to estimate the "verification ratio", i.e., the percentage of voters which seamlessly trigger the fully automated verification procedure introduced in Section 7.2.

8.1. Design of sElect

Our goal has been to design a particularly lightweight remote system which (still) achieves a good level of privacy, verifiability, and accountability. The system is supposed to be lightweight both from a voter's point of view and a design/complexity point of view. For example, we do not want to require the voter to install software or use a second device. Also, verification should be a very simple procedure for a voter or should even be completely transparent to the voter.

The system combines several concepts, such as verification codes (see, e.g., [DLM82]) and Chaumian mix nets [Cha81], in a novel way. One of the unique features of sElect is that it supports a *fully automated verification procedure* which does not require any user interaction and it is triggered as soon as a voter looks at the election result.

Cryptographic primitives. sElect uses only basic cryptographic operations: public-key encryption and digital signatures. More specifically, the security of sElect is guaranteed for any IND-CCA2-secure public-key encryption scheme and any EUF-CMA-secure signature scheme (see Appendices A.1 and A.2, respectively), and hence, very standard and basic cryptographic assumptions. Typically, the public-key encryption scheme will employ hybrid encryption[17] so that arbitrarily long messages and voter choices can be encrypted. We note that, for the privacy property of sElect, the only requirement regarding the cryptographic schemes is that for every public-key and for any two plaintexts of the same length, the public-key encryption scheme always yields ciphertexts of the same length. This property seems anyway to be satisfied by all practical schemes.

To simplify the protocol description, we use the following convention. First, whenever we say that a party produces a signature on some message m, this implicitly means that the signature is in fact computed on the tuple $(elid, tag, m)$, where $elid$ is an election identifier (different for different elections) and tag is a tag different for signatures with different purposes (for example,

[17] A hybrid encryption scheme merges the convenience of an asymmetric (public-key) encryption scheme with the efficiency and the flexibility of a symmetric encryption scheme: the public-key encryption scheme, computationally more demanding, is used to only encrypt a freshly generated symmetric key which is then used with the symmetric encryption scheme to encrypt the messages.

a signature on a list of voters uses a different tag than a signature on a list of ballots). Similarly, every message encrypted by a protocol participant contains the election identifier.

Set of participants. The set of participants of the protocol consists of an append-only *bulletin board B*, *n voters* $v_1, \ldots, v_n$ and their *voter supporting devices (VSDs)* $vsd_1, \ldots, vsd_n$, a *collecting server CS*, *m mix servers* $M_1, \ldots, M_m$, and a *voting authority VA*. For sElect, a VSD is simply the voter's browser (and the computing platform the browser runs on).

We assume that there are authenticated channels from each VSD to the collecting server *CS*. These channels allow the collecting server to ensure that only eligible voters are able to cast their ballots. By assuming such authenticated channels, we abstract away from the exact method the VSDs use to authenticate to the collecting server; in practice, several methods can be used, such as one-time codes, passwords, or external authentication services such as single sign-on systems (see Section 8.2 for a concrete instantiation).

We also assume that for each VSD there is one (mutual) authenticated and one anonymous channel to the bulletin board *B*. Depending on the phase, the VSD can decide which channel to use in order to post information on the bulletin board *B*. In particular, if something went wrong, the VSD might want to complain anonymously (e.g., via a proxy) by posting data on the bulletin board *B* that identifies the misbehaving party.

A protocol run consists of the following phases: the *setup phase* (where the parameters and public keys are fixed), the *voting phase* (where voters choose their candidate and let their VSDs create and submit the ballots), the *mixing phase* (where the mix servers shuffle and decrypt the election data), and the *verification phase* (where the voters verify that their ballots were counted correctly). These phases are now described in more detail.

Setup phase. In this phase, all the election parameters (the election identifier, list of candidates, list of eligible voters, opening and closing times, etc.) are fixed and posted on the bulletin board by VA.

Every server (i.e., every mix server and the collecting server) runs the key generation algorithm of the digital signature scheme to generate its public/private (verification/signing) keys. Also, every mix server M_j runs the key generation algorithm of the encryption scheme to generate its public/private (encryption/decryption) key pair (sk_j, pk_j). The public keys of the servers (both encryption and verification keys) are then posted on the bulletin board *B*; proofs of possession of the corresponding private keys are not required.

Voting phase. In this phase, every voter v_i can decide to abstain from voting or to vote for some candidate (or more generally, make a choice) m_i. In the latter case, the voter indicates her choice m_i to the VSD. In addition, for verification purposes, a *verification code* n_i is generated (see below), which the voter is supposed to write down/store. At the end of the election, the choice/verification code pairs of all voters who cast a vote are supposed to be published so that every voter can check that her choice/verification code pair appears in the final result, and hence, that her vote was actually counted. The verification code is a concatenation $n_i = n_i^{voter} \| n_i^{vsd}$ of two nonces. The first nonce, n_i^{voter}, which we call the *voter chosen nonce*, is provided by the voter herself, who is supposed to enter it into her VSD (in our implementation, see Section 8.2, this nonce is a nine character string chosen by the voter). It is not necessary that these nonces are chosen uniformly at random. What matters is only that it is sufficiently unlikely that different

voters choose the same nonce. The second nonce, n_i^{vsd}, is generated by the VSD itself, the *VSD generated nonce*. Now, when the verification code is determined, the VSD encrypts the voter's choice m_i and the verification code n_i, i.e., the choice/verification code pair $\alpha_m^i = (m_i, n_i)$, under the last mix server's public key pk_m using random coins r_m^i, resulting in the ciphertext $\alpha_{m-1}^i = \text{Enc}_{pk_m}^{r_m^i}((m_i, n_i))$. Then, the VSD encrypts α_{m-1}^i under pk_{m-1} using the random coins r_{m-1}^i, resulting in the ciphertext $\alpha_{m-2}^i = \text{Enc}_{pk_{m-1}}^{r_{m-1}^i}(\alpha_{m-1}^i)$, and so on. In the last step, it obtains

$$\alpha_0^i = \text{Enc}_{pk_1}^{r_1^i}(\ldots(\text{Enc}_{pk_m}^{r_m^i}(m_i, n_i))\ldots).$$

The VSD submits α_0^i as v_i's ballot to the collecting server *CS* on an authenticated channel. If the collecting server receives a ballot in the correct format, then *CS* responds with an acknowledgement consisting of a signature on the ballot α_0^i; otherwise, it does not output anything. If the voter/VSD tried to re-vote and *CS* already sent out an acknowledgement, then *CS* returns the old acknowledgement only and does not take into account the new vote.

If a VSD does not receive a correct acknowledgement from the collecting server *CS*, the VSD tries to re-vote, and, if this does not succeed, it files a complaint on the bulletin board using the authenticated channel. If such a complaint is posted, it is in general impossible to resolve the dispute and decide who is to be blamed: *CS* who might not have replied as expected (but claims, for instance, that the ballot was not cast) or the VSD who might not have cast a ballot but nevertheless claims that she has. Note that this is a very general problem which applies to virtually any remote voting protocol. In practice, the voter could ask the VA to resolve the problem.

When the voting phase is over, *CS* publishes two lists on the bulletin board, both in lexicographic order and without duplicates and both signed by the collecting server: the list C_0 containing all the cast valid ballots and the list *LN* containing the identifiers of all voters who cast a valid ballot. It is expected that the list *LN* is at least as long as C_0 (otherwise *CS* will be blamed for misbehavior). Note that, with high probability, a ballot of an honest voter will be different from the ballot of another honest voter, as their voter chosen nonces will be different (and if the VSD is honest, also the VSD generated nonces). Ballots of dishonest voters might coincide with those of other voters. Their ballots will then not be counted.

Mixing phase. The list of ciphertexts C_0 posted by the collecting server is the input to the first mix server M_1, which processes C_0, as described below, and posts its signed output C_1 on the bulletin board. This output is the input to the next mix server M_2, and so on. We will denote the input to the j-th mix server by C_{j-1} and its output by C_j. The output C_m of the last mix server M_m is the output of the mixing stage and, at the same time, the output of the election. It is supposed to contain the plaintexts $(m_1, n_1), \ldots, (m_n, n_n)$ (containing voters' choices along with their verification codes) in lexicographic order.

The steps taken by a mix server M_j are as follows:

1. *Input validation.* M_j checks whether C_{j-1} has the correct format, is correctly signed, arranged in lexicographic order, and does not contain any duplicates. If this is not the case, it sends a complaint to the bulletin board and stops its process (this in fact aborts the whole election process and the previous server is blamed for misbehaving). Otherwise, M_j continues with the second step.

2. *Processing.* M_j decrypts all entries of C_{j-1} under its private key sk_j, removes duplicates, and orders the result lexicographically. If an entry in C_{j-1} cannot be decrypted or is decrypted to a message in an unexpected format, then this entry is discarded and not further processed. The sequence of messages obtained in such a way is then signed by M_j and posted on the bulletin board as the output C_j.

Verification phase. After the final result C_m has been published on the bulletin board B, the verification phase starts. As mentioned in the introduction, a unique feature of sElect is that it supports the following two forms of verification, explained next: *(pure) voter-based verification*, and hence human verifiability, and *(fully automated) VSD-based verification*.

The first form is carried out by the voter herself and does not require any other party or any device, and in particular, it does not require any trust in any other party or device, except that the voter needs to be able to see the published result on the bulletin board. As we will see below, the verification procedure is very simple.

VSD-based verification is carried out fully automatically by the voter's VSD and triggered automatically as soon as the voter takes a look at the final result, as further explained in Section 8.2. It does not need any input from the voter. This is supposed to result in high verification rates and further ease the user experience, as verification is performed seamlessly from the voter's point of view and triggered automatically. Under the assumption that VSDs are honest, it yields verifiability, and even a high-level of accountability.

We now describe how these two forms of verification work in detail.

Voter-based verification. For voter-based verification, the voter simply checks whether her verification code, which in particular includes the voter chosen nonce n_i^{voter}, appears next to her choice in the final result list. That is, the voter should check that the final result list contains a choice/verification code pair of the form $(m_i, n_i^{voter} \| n_i^{vsd})$ where m_i is her choice and n_i^{vsd} is the VSD generated nonce, which the voter could ignore (the voter might not trust its VSD). If this is the case, the voter would be convinced that her vote was counted. A voter v_i who decided to abstain from voting may check the list *LN* to make sure that her name (identifier) is not listed there.[18] When checks fail, the voter would file a complaint.

VSD-based verification. For VSD-based verification, the voter's VSD performs the verification process fully automatically. In particular, this does not require any action or input from the user. In our implementation, as further explained in Section 8.2, the VSD-based verification process is triggered automatically whenever the voter goes to see the election result. Clearly, this kind of verification provides security guarantees only if the VSD is honest, and hence, for this kind of verification, the voter needs to trust her device. Making use of the information available to the VSD, the VSD can provide evidence if servers misbehaved, which can then be used to rightfully blame misbehaving parties. The VSD-based verification process works as follows. A VSD vsd_i checks whether the originally submitted plaintext (m_i, n_i) appears in C_m. If this is not the case, the VSD determines the misbehaving party, as described below. Recall that a VSD which did not obtain a valid acknowledgment from the collecting server was supposed to file a complaint

[18] Variants of the protocol are conceivable where a voter signs her ballot and the collecting server presents such a signature in case of a dispute. This solution is conceptually simple. On the pragmatic side, however, it is not always reasonable to expect that voters maintain keys and, therefore, here we consider the simpler variant without signatures. Note that this design choice was also made in several existing and prominent systems, such as Helios.

already in the voting phase. The following procedure is carried out by a VSD vsd_i which obtained such an acknowledgement and cannot find the plaintext (m_i, n_i) in C_m. First, the VSD vsd_i checks whether the ballot α_0^i is listed in the published result C_0 of the collecting server CS. If this is not the case, the VSD vsd_i anonymously publishes the acknowledgement obtained from CS on the bulletin board B which proves that CS misbehaved (recall that such an acknowledgement contains a signature of CS on the ballot α_0^i). Otherwise, i.e., if α_0^i is in C_0, the VSD checks whether α_1^i is listed in the published result C_1 of the first mix server M_1. If C_1 contains α_1^i, the VSD vsd_i checks whether α_2^i can be found in the published result C_2 of the second mix server M_2, and so on. As soon as the VSD vsd_i gets to the first mix server M_j which published a result C_j that does not contain α_j^i (such a mix server has to exist), the VSD anonymously sends (j, α_j^i, r_j^i) to the bulletin board B. This triple demonstrates that M_j misbehaved: the encryption of α_j^i under pk_j with randomness r_j^i yields α_{j-1}^i, and hence, since α_{j-1}^i is in the input to M_j, α_j^i should have been in M_j's output, which, however, is not the case.

We say that a voter v_i *accepts* the result of an election if neither the voter v_i nor her VSD vsd_i output a complaint. Otherwise, we say that v_i *rejects* the result.

Remark 8.1. Note that the procedures for ballot casting and mixing are very simple. In particular, a mix server needs to carry out only n decryptions. Using standard hybrid encryption based on RSA and AES, it amounts to n RSA decryption steps (n modular exponentiations) and n AES decryptions. This means that the mixing step is very efficient and the system is practical even for very big elections: mixing 100 000 ballots takes about 3 minutes and mixing one million ballots takes less than 30 minutes with 2 048-bit RSA keys on a standard computer/laptop.

8.2. Implementation of sElect

We have implemented sElect as a platform independent web application [SST16]. In order to vote, voters simply use a browser to vote, without the need to install other software, browser extensions, or plug-ins, and hence, they can vote on many platforms (desktop computers, smartphones, etc.). In particular, voters visit a web site which serves what we call a voting booth. Except for serving static files (HTML/CSS/JavaScript), the voting booth server does not play any role in the voting process. All the computations, including in particular ballot creation and verification of acknowledgements, are carried out locally on the voters' machine within the browser. Votes only leave the browser encrypted (as ballots), to be submitted to the authorization server. While for the mock elections, only one server serving the voting booth was set up, the idea would be that a voter can choose a voting booth of any organization/company she trusts or even set up her own voting booth server.

Voters merely need a browser to vote and to verify their votes. In order to vote, voters go to a web site that serves what we call a *voting booth*. More precisely, a voting booth is a collection of static HTML/CSS/JavaScript files served by a so-called voting booth server. Other than that, there is no interaction between the voter's browser and the voting booth server: ballot creation, casting, and verification are then performed within the browser, as explained below (of course for ballot casting, the voter's browser communicates with the collecting server). Ideally the voter can choose, among different voting booth servers, the voting booth server that she trusts and that is

independent of the election authority. Voting booths might be run by different organizations as a service and independently of a specific election (see also the discussion in Chapter 10). So what abstractly was called a VSD in the previous sections, in our implementation comprises the voter's computing platform, including her browser as well as some voting booth server which the voter picks and which serves the static HTML/CSS/JavaScript files to be executed. The JavaScript code performs the actual actions of the VSD described in Section 8.1 within the browser and without further interaction with the voting booth server. On a mobile device one could, for example, also provide an app to the voter which performs the task of the VSD; again there might be more apps from which the voter could choose. This of course assumes that the voter installs such an app on her device. Since the idea is that a voting booth can be used independently of a specific election, this is reasonable as well. In this case, the receipt with all the data required for the VSD-based verification process could be even more permanently stored within the mobile app. Furthermore, given that the typical voter usually owns a single mobile device for personal use, the voter would typically use the same device both for casting her ballot and for checking the election result, always triggering the fully automated verification procedure. This would consequently increase the verification ratio which, as reported in [KMST16a], is directly related to the level of verifiability and accountability provided by sElect.

In our implementation, we use only one authorization method: one time passwords. These are sent to the voters' e-mail addresses when voters initiate the voting process. However, it would not be a problem to support different authorization methods.

After the authorization phase, a voter enters her vote in the browser (on the voting booth's web site) and then ballot creation and verification of positive or negative acknowledgment (see below) are carried out locally within the voters' browser. Votes only leave the browser as encrypted ballots to be submitted to the collecting server. When submitting the ballot, the collecting server is supposed to reply with a signed acknowledgment, verified by the voting booth, communicating whether the server properly collected the cast ballot or not. In case the ballot is not correctly collected, the collecting server sends a negative acknowledgment indicating the reason:

(1) Election already closed or not yet opened.

(2) Malformed request.

(3) Wrong election ID.

(4) Voter not eligible for such an election.

(5) Double voting attempt.

Furthermore, the voting booth might also report a network error in case the connection is down or the collecting server is unreachable. Full receipts, i.e., all the information required for the VSD-based verification process, are saved using the browser's local storage (under the voting booth server's origin[19]): other web sites cannot access this information.

[19] In web applications, an *origin* is defined as a combination of the protocol used (HTTP or HTTPS), the domain name, and the port number. Web-pages loaded under the same protocol, the same domain, and the same port share the same origin. By an access control mechanism called *Same-Origin Policy* (SOP), browsers permit scripts contained in a first web-page to access data in a second web-page only if the two pages share the same origin.

We note that with this architecture the voting booth learns both the voter's credentials and her choice. To avoid these two pieces of information being learned by the same component of the system, we propose a more sophisticated version of the voter supporting device (VSD) in which another static page, the so-called *authenticator*, is downloaded in addition to the voting booth client. We refer the reader for the paragraph "Separate autentication" below for all the details.

When the election is over, the voter is prompted to go to her voting booth again in order to check the election result. When the voter opens the voting booth in this phase, it automatically fetches all the necessary data and carries out the automated verification procedure; if the voter's ballot has not been counted correctly, cryptographic evidence against a misbehaving server is produced, as described in Section 8.1. In addition to this fully automated check, the voter is given the opportunity to visit the bulletin board (another web-page), where she can see the result and manually check that her verification code is listed next to her choice.

User experience. We now describe and illustrate with screenshots the user experience of our implementation of sElect.

In the simpler implementation, i.e., in the implementation where the authentication and the casting of the votes are both operated by the voting booth, a user opens the voting booth (in a browser) where she is asked for her e-mail address (Figure 8.1a). The voting booth forwards this e-mail address to the collecting server which (if the voter is eligible) generates a one-time password for the voter and sends it to her e-mail address. The user is then supposed to enter this one-time password (copy and paste from her e-mail) in the voting booth web-page (Figure 8.1b). Then, the user is asked to provide a random code of nine characters, which will be used as part of the verification code (Figure 8.1c). Next, the user is prompted by the voting booth to make her choice (Figure 8.1d). Then, the voting booth, in the background, generates a random verification code, concatenates it with the code entered by the voter, and creates a ballot. This ballot is then, along with the one-time password, sent by the voting booth to the collecting server. The server is supposed to reply with an acknowledgement which is verified by the voting booth. After that, the verification code is displayed to the voter, who can then copy her verification code or save it as a picture (Figure 8.1e). Independently, the verification code along with the full receipt (the data necessary to blame misbehaving parties in case something should go wrong) is stored in the browser's *local storage*, an HTML5 feature for permanently storing data within the user's browser. We note that data is stored in the local storage by origin and then, by the Same Origin Policy (SOP), only JavaScript running under that origin can access this data. In our case, the idea is that a voting booth runs in its own (HTTPS) origin, and hence, only (the JavaScript loaded from) this voting booth can access the receipt stored in the user's browser.

We emphasize that we deliberately wanted to keep the user interface very simple. Therefore, only the verification code is shown to the voter (a concept voters should understand). The rest of the receipt, which is used for accountability purposes, is stored and checked only by the voting booth on the voter's browser.

When the election is over, the voter is prompted to open her voting booth again, namely the same web-page she used to vote. In our deployment, e-mails were sent to the voters informing them that the result of the election was ready and that the voter can see the result and check her verification code following a link to her voting booth. When the voter opens the voting booth in this phase, it fetches the information stored in the browser's local storage, which should

contain the full receipt, and the result of the election from the bulletin board, and then verifies signatures and makes sure that the verification code is listed in the final result along with the chosen candidate. If this is the case, the voter is informed that her vote has been counted correctly (Figure 8.1f). Otherwise, the evidence for blaming a (dishonest) party is generated and the voter is informed that the verification procedure failed. In particular, the complaint singles out the misbehaving party and provides evidence of the misbehavior. For instance, in Figure 8.1g the collecting server has been singled out as the misbehaving party. In addition to this fully automated check (carried out as soon the voter visits her voting booth), the voter is given the opportunity to visit the bulletin board, where she can see the result (Figure 8.1h) and manually check that her verification code is listed next to her choice (Figure 8.1i).

Separate authentication. In order to ensure that no component of sElect learns both the voters' identity and their choice, we propose a more sophisticated architecture and implementation of the web-pages used by the voters to take part in elections. In this architecture, which is meant to keep the user experience unchanged, the authentication of eligible voters is delegated to a separate component, called the *authenticator*. We now describe how, during the voting phase, this component interacts with the voting booth and the collecting server. We assume all these components running on different origins.

1. When a voter opens the voting booth in her browser, an HTML/Javascript document is retrieved from the collecting server's origin inside an `iframe`[20] of the voting booth web-page.

2. The voting booth sends the `election manifest` (the file containing all the election's attributes and hence uniquely defining each election) to the HTML/Javascript document embedded in the iframe. This document stores the election manifest in its *session storage*, a HTML5 feature for storing data within the user's browser until the page session ends (i.e., the page/tab of the browser is closed). As the voting booth and the collecting server run under different origins, the election manifest is exchanged using the `postMessage()`, a method of the `Window` object[21] allowing for cross-communication among origins.

3. The voting booth web-page is then automatically redirected to the authenticator web-page, which also contains an `iframe` retrieving the same HTML/Javascript document retrieved by the voting booth from the collecting server's origin.

4. The authenticator allows the voter to execute the authentication procedure in the exact same way as described in the previous paragraph (Figure 8.1a and 8.1b), with the only difference being that the voter's credentials are now sent to the page inside the iframe where they are also stored in its session storage.

5. The authenticator web-page is, in turn, redirected back to the voting booth web-page where the voter can now provide her part of the verification code (Figure 8.1c) and make her choice (Figure 8.1d).

[20] `iframe` is an HTML tag which is used to embed another document within the current one.

[21] The `Window` object represents an open window/tab in a browser containing a DOM (Document Object Model) document loaded in that window.

6. The (multiple encrypted) ballot created by the voting booth is then sent to the iframe where it is joined up with the voter's credentials and sent to the collecting server using a POST request.
7. The collecting server is supposed to send back a signed acknowledgement to the `iframe` which, in turn, forwards it via a `postMessage()` the voting booth web-page. The voting booth verifies it and, in case the ballot has been properly cast, invites the voter to save her verification code (Figure 8.1e).

We notice that data in the session storage are stored per origin (as for the local storage), but also per `Window` object: that is, only JavaScript running under that origin and within the same `Window` object can access both the voter's credentials and the ballot. Therefore, by the Same Origin Policy (SOP) neither the voting booth nor the authenticator can have access to the session storage of the page in their `iframe`, as it is loaded from the collecting server's origin. (The only possible way these components can exchange messages is through the `postMessage()` method of the `Window` object which, as already mentioned above, allows for cross-communication among different origins.) Moreover, it is also worth noticing that this architecture prevents the voters' credentials and their ballots stored in the session storage of the `iframe` to be accessed by other possible malicious web-pages loaded in a different tabs of the voter's browser, since they would be loaded on a different `Window` object.

However, this architecture leaves in principle open the possibility for another attack: an attacker might load the HTML/JavaScript document retrieved from the collecting server in an `iframe` of a different web-page, controlled by him. In this way, he could then forge the voting booth and/or the authenticator web-page intercepting the voter's credentials and/or her choice. In order to defend against this attack, following [SS13], we perform some checks in the JavaScript code allowing only web-pages which are loaded under a set of trusted origins (in our case, the voting booth's and authenticator's origins) to successfully post messages to the `iframe`. In addition, for browsers supporting the Content Security Policy (CSP),[22] we set the `Content-Security-Policy` HTTP header to instruct the browser to load pages within `iframes` only if their ancestors are also loaded from the specific domains: in this case, the domains where the voting booth and the authenticator reside.

Cryptographic schemes. The cryptographic operations used in the implementation of sElect are part of the PKCS #1 v2.2 family of standards [KJRM16]:

- The `RSAES-OAEP` public-key encryption scheme for symmetric and asymmetric encryption which, as already mentioned in Section 4.1, combines the RSA algorithm [RSA78] with the Optimal Asymmetric Encryption Padding (OAEP) method [BR95]. Hybrid encryption is employed in the sense that the asymmetric RSA encryption algorithm is used only to exchange a session key among the parties which consecutively encrypt the data using a more efficient symmetric encryption scheme: the AES (Advance Encryption Standard) scheme is used in GCM mode[23] with random initialization vector (IV) of 96 bit. For

[22] Content Security Policy (CSP) is a security measure defending against cross-site scripting (XSS), clickjacking, and other code injection attacks.

[23] Galois/Counter Mode (GCM) is an authenticated encryption mode of operation for symmetric key block ciphers with a block size of 128 bit. Besides confidentiality, it is designed to also provide data authenticity.

symmetric encryption with AES we use keys of 256 bits, while for asymmetric encryption we use keys of 1024 bits.

- The `RSASSA-PSS` scheme for digital signature, based on the Probabilistic Signature Scheme (PSS) proposed by Bellare and Rogaway [BR96]. As for public-key encryption, we use keys of 1024 bit.

In the voting booth web-page running in the voter's browser, we used the JavaScript `node-forge`[24] cryptographic library, whereas in the core of the mix server the Java `Bouncy Castle` cryptographic library.[25]

We refer the reader to [SST16] for our implementation of sElect.

(a) (b)

(c) (d)

[24] `https://digitalbazaar.com/forge/`
[25] `https://www.bouncycastle.org/`

Election closing time: Saturday, Nov 10th 2018, 17:00

Students' Union Election

Election Identifier: E873 2092 4353 0017

Your ballot has been accepted by the collecting server.

When the election is over, you can manually check that your ballot is in the final tally. If you want to do this, you need to

save/write down the following verification code

and look it up in the result of the election: it should appear next to your choice.

Your verification code: wk%m5=Q!FCCBCD98 Save as a picture

The first 8 characters are the code you entered, while the remaining part was generated randomly by the system.

Thank you!

Secure elections powered by *sElect*

(e)

Election closing time: Saturday, Nov 10th 2018, 17:00

Students' Union Election

Election Identifier: E873 2092 4353 0017

The election is closed and the result is ready and available.

To see the result and check your verification code, you can now

go to the result web page

Independently, an automatic verification procedure was just carried out to check that the ballot with the following verification code has in fact been counted:

wk%m5=Q!FCCBCD98

Verification successful ✓

Secure elections powered by *sElect*

(f)

Election closing time: Saturday, Nov 10th 2018, 17:00

Students' Union Election

Election Identifier: E873 2092 4353 0017

The election is closed and the result is ready and available.

To see the result and check your verification code, you can now

go to the result web page

Independently, an automatic verification procedure was just carried out to check that the ballot with the following verification code has in fact been counted:

wk%m5=Q!FCCBCD98

VERIFICATION FAILED: ballot with verification code wk%m5=Q!FCCBCD98 is missing!

Looking for the misbehaving party.

Ballot wk%m5=Q!FCCBCD98 has been dropped by the collecting server

The following data contains information necessary to hold the misbehaving party accountable. Please copy it and provide to the voting authorities.


```
{"electionID":"e873209243530017a31da4fc9c59649d2545da51996624438405f2fb863af2c9","signature":"
028ecc5c2861ebe9a8cc3a64e9354d0da9b3b1f23209f64d929b24fb00621a975051ae950704f914efc49d08c09339
1bbdcd7e042343f226ab263ca5752d764493c0c59a681c92aa27ee395293685ccb40d23e2f27a784d852e345ad4cf3
c5572dcff56d9471fb07f67beba2e0c5730896201408b3572f9a791838df20c97b99","ballot":"0000008084d376
9e3aa5a701d916d81c57e6d830c00b7133a717d912fc01ba6e09a469f7eede6598f01c6be42722208af289cef07d05
1066d18e788b3048efad991db874d471e0563fc6096f5bb55a1a9bd4e9e6c2f8fa0bb147a7489753e480d6e6cf022e
4026e216bd9041fe0d13e4bdae9541a3d3979e105ce9d2bfeae4e2b0b77ac8e23ef718a6e5a0b5f4e0299b89fa7121
c9e15628528a61dbaa57133d755dc6cbfb67fb11c71085db4004c70cf4b846bae71eddc39571b6d6d3fb3a489c77d2
ff399d26c104bf1dbe9e4df4938889ad43ad082adbbef6b2f5146509f790386d0678f21402afca4dfb581521db0f63
69318907522d2230e9e87188fbb476e12914508143174814b3e5011e18a442909b7da04e50aa4fcc66c71da819665b
57008ce07542cb2a8df8f295fafebdab99cc42b1dec699579c26d1833ea73a21e10ebf1cba85f8107282ad1142f1b5
26b866d7ff9d8553eca6d46f3051e82d30474318d671612a6c33d24b12460e48d87939f054aaea907c51e518d1a436
ce5f90db166b9ce00fa2d946107acfde0d7dec5643c08f5f49faa1eacbcaf0dedea1e03f6349a360463d4b57601b68
ffc8fa0d9fce43bdd3dffa8cf74a3f98fc052df3d1de60e4ce51df4740935938bc5521d8d488dbaaa320ddffbaebaf
5918c184afdd567baae2b2ef156bf155b1f3ed3bf3eb98f595fe724435c51cc1659e9351e61252e25f5302b9d6ad8f
7a40f8688b5990367495b7df23c02d12dd088bd3db0217e947b91a394fd4d54c3542e054e7b83e522e9b3e4cb7b67f
8e33fe9e023f2c3241f623172eb0a153c99abaea0655579b2db10408357483e0a6cdf85fa8055f27091bc20755b0ec
ad7f323dfda61b2e213953fc3a01c27e657086b390cba96b9ed61f7005220f44bde0e458c0d77d866befb9f2ffad7b
c41fc96e831449d7c67c932880a3bd6f167a59d6"}
```


Secure elections powered by *sElect*

(g)

Description

Election for the Chairperson of the Students' Union of the Computer Science Department.

Summary | Verification Codes | Additional Details

Result of the election

Choice	Number of votes
Sara Johnson	4
John Stevens	3
Lisa Wilson	2
Patrick Bernard	2
Mohammed Basha	2
Daniel Fischer	2
Lucy Romero	0
Chris Taylor	0

(h)

Description

Election for the Chairperson of the Students' Union of the Computer Science Department.

Summary | Verification Codes | Additional Details

List of Votes

Please check that your choice is listed next to your verification code.

verification code	choices
:o2_afv3D358594E	Sara Johnson | Daniel Fischer
a@lre}{03BFD5E1E	Sara Johnson | Patrick Bernard
b23*a_z20366233B	John Stevens | Sara Johnson | Patrick Bernard
bgb_1243F40ACC94	John Stevens | Mohammed Basha | Lisa Wilson
d_-as3}}AF3899A6	John Stevens | Daniel Fischer
wk%m5=Q!FCCBCD98	Sara Johnson | Mohammed Basha | Lisa Wilson

(i)

Secure elections powered by *sElect*

Election Manager

sElect secure election

▸ Click on the *entry* of the election you want to manage or create a new one.

Election ID	Election Title	Starting Time	Ending Time	Election State
bb29072	Students' Union Election	2017-03-20 11:42 AM	2017-03-24 11:42 AM	closed
b67b9aa	Students' Union Election	2017-04-10 4:01 PM	2017-04-14 4:01 PM	closed
47a7d06	Students' Union Election	2017-04-10 10:54 PM	2017-04-14 10:54 PM	closed
6d35a52	Students' Union Election	2017-05-23 12:05 PM	2017-05-27 12:05 PM	open

Create Custom Election

Set up a Mock Election

(for Mock Elections no real email addresses necessary)

Close Election

Remove Election

Invite Voters to Vote

Help

(j)

Figure 8.1.: Screenshots showing the user experience of our implementations of both sElect and its election manager.

8.3. Deployment of sElect

To obtain user feedback and, in particular, to get a first estimate of the verification ratio for the fully automated verification, we carried out two mock elections.

We used a slightly modified version of the voting booth which allowed us to gather statistics concerning the user behavior. We emphasize that these field tests were not meant to be full-fledged and systematic usability studies, which we leave for future work.

The participants in our first mock election were students of our department (who voted for the "Students' Union Election"): 52 voters cast their ballots and 30 (out of 52) verification codes/receipts were checked automatically by the voting booth. This gives a verification ratio of 57.7%.

The participants of our second mock election were researchers of a national computer science project (who voted on their favorite text editor). In this case, we recorded 22 cast ballots and 13 (out of 22) verification codes/receipts were checked automatically, which gives a verification ratio of 59.1%. Again, the overall verification ratio might be even higher considering possible voter-based verification.

As one can see, the verification ratio was quite high in both cases (57.5% and 59.1%). In fact, with such a high ratio, the dropping or manipulation of even a very small number of votes is detected with very high probability, according to the results reported in [KMST16a] and [KMST16b]. Moreover, in both these mock elections, we can expect that some number of verification codes were checked manually, so the overall verification ratio might be even higher. However, we do not have reliable data about voter-based verification.

We believe that for real elections one might obtain similar ratios: voters might be even more interested in the election outcome than in a mock election and, hence, they would tend to check the result and trigger the automated verification procedure.

To create a presentable demo version of the system and to ease its deployment, we have also implemented a web-based "Election Manager" allowing for the creation, the customization, and the removal of elections powered by sElect. The (static) web-interface is provided by a server also implemented in `node.js`, while the creation, the removal, and the resumption (in case of temporal dismission of the service) of elections are handled by `python` scripts. The NGINX web server is used to handle all the incoming and outgoing traffic to the seven components of sElect running for each election.

The user interface is also quite simple (see Figure 8.1j): an user can either set up a mock election where all the election's attributes (see item 1a) below) are already prestablished or set up a custom election where she has to specify the election's attributes using an HTML form. In any case, the web-page triggers, on the back-end, the python script responsible for spawning a new election. The script, receiving as input a JSON object containing all the attributes of the election to be created, roughly performs the following steps:

1. It determines all the settings related to the election, namely:
   a) the election's attributes, namely:
      – election lookup string (ELS), i.e., a string uniquely identifying each election,

- title of the election,
- description of the election,
- list of eligible voters,
- question of the election,
- further explanation of the question,
- list of choices,
- minimum number of choices per voter,
- maximum number of choices per voter,
- election starting time,
- election ending time,
- number of mix servers;

b) system and electoral flags, namely:

- `userChosenRandomness`:
  the voter is prompted to insert a part of her verification code,
- `showOtp`:
  the OTP is shown on the web-page to be directly inserted by the voter (only for the demo and for testing),
- `publishListOfVoters`:
  the list of voters who cast their ballot is published in the bulletin board,
- `mockElection`:
  the election is a mock election where some mock voters have already cast their ballot,
- `hidden`:
  the election does not appear on the web-page,
- `separateAuthentication`:
  the voters' authentication is delegated to the authenticator;

c) the cryptographic keys of each component;

d) the URL of each component;

e) the port each component will listen to;

f) the path to the file containing the list of confidential voters (in case all or part of the list of eligible voters does not have to be made public).

2. It generates the file `ElectionManifest.json` which contains all the election's attributes and therefore uniquely defines each election (its hash is indeed the election ID shown on the web-page).

3. In case of creation of a mock election, it sets up some ballots that mock voters are supposed to have already cast.

4. It starts all the `node.js` servers of the non-static components of sElect, namely the collecting server, the bulletin board, and the chain of mix servers. It then properly saves all the data related to these servers, i.e., their process IDs, the ports they listen to, and the information of the election (ELS, election ID, election description, starting time, and number of mix servers) they are related to.

5. It configures the NGINX web server with proxy servers for the collecting server, the bulletin board, and the chain of mix servers. Each proxy server binds each `node.js` server listening to a specific port to the external URL determined in step 1.; the voting booth and the authenticator (if present) are instead configured to be served as static web-pages.

We note that, during the creation of an election, additional care is taken to put each component on a different subdomain, i.e., on a different origin. This prevents the possibility for a component to access data stored by another component within the voter's browser: the access to the two HTML5 web storages employed, namely the session and the local storage, is regulated by the Same Origin Policy, which, as already explained in Section 8.2, allows scripts running in a first web-page to access data on a second web-page only if the two pages share the same origin.

When the user clicks on a particular election, the election manager web-page shows a link to the voting booth web-page which has to be forwarded to the eligible voters in order to make them cast their ballot. Once the election is over, the voters are invited to check the election result by visiting the same web-page they used to vote which then automatically triggers the verification procedure (see Section 8.2 for more details).

The election manager allows also for closing an election in advance and for dismissing it completely. In the latter case, another `python` script essentially shuts down the `node.js` servers and removes both the proxy servers and the modules for serving the static pages from the NGINX configuration file.

The sElect e-voting system has also been deployed for a more important election, where members of a program committee were asked to choose up to three candidates as the invited speaker of a security workshop. For this real-world election, we adopted the following deployment choices:

- With the aim of smoothing the user experience as much as possible, we did not prompt the voters to insert their part of the verification code (they had then to fully trust our voting booth);
- We set up for the first time the separate authentication mechanism to obtain feedback on its portability on different browsers.

Regarding the latter point, although we do not have reliable data about the portability of the separate authentication mechanism, we note that (1) the separate authentication mechanism has been implemented using only standard HTML5 components and features which are nowadays supported by every browser and that (2) nobody in the program committee reported any problem in interacting with the system.

We refer the reader to [SS16] for our implementation of the election manager.

9. Formal Verification of the sElect E-voting System

An important challenge related to systems of electronic voting is to formally analyze them in order to establish whether and to what degree they satisfy the security properties they were designed to guarantee (see the introduction of this part for the security properties).

Security vulnerabilities may occur at various levels.

(i) At the specification/design level: a system might be insecure by design, for example, because certain critical messages are not signed nor encrypted.

(ii) At the implementation level: confidential information might leak because of programming errors and security flaws (for instance, buffer overflow).

(iii) At the system level: certain software components might become compromised because of system level attacks (for instance, buffer overflow, code injection, security misconfigurations, insecure cryptographic storage, etc.).

(iv) At the hardware/physical level: confidential information might leak due to side channel attacks (for example, timing attacks on smart cards) or hardware components (for example, chips in a voting machine) might have been replaced by malicious ones.

As already reported in the introduction, most of the verification efforts for modern e-voting systems have concentrated on the design level, performing cryptographic analysis on the protocol of such systems (see, for example, [KTV10a, KTV10b, KTV10c, CS11, KTV11, KTV12b, ACW13, CEK$^+$15, KZZ15a, CCFG16, KZZ17, CW17]).

In this chapter, we present our efforts in extending such an analysis one step further, i.e., directly at the implementation level. More specifically, our aim is to establish strong cryptographic privacy of the votes the voters cast into the sElect voting system presented in Chapter 8. We note that performing implementation-level analysis of code containing cryptographic operations is far from being trivial: the only other similar line of research aims at providing securities guarantees at some properly tuned implementations of the latest versions of the TLS protocol (see, e.g., [BFK$^+$13, DFK$^+$17]).

9.1. Verification of the Mix Server

Since the component of the sElect voting system supposed to provide confidentiality of the cast votes is the Chaumian Mix-net (the original and simplest type of mix net introduced by David Chaum in [Cha81]), we analyze the cryptographic core of one of the mix servers[26] which, for

[26] We remind that a Chaumian Mix-net is defined as a chain of semantically equivalent mix servers each of whom, upon receiving a list of messages encrypted with its public key, decrypts all the ciphertexts, shuffles the decrypted messages, and outputs them signed.

verification purposes, has been indeed implemented in Java (the bulk of the mix server is written in `node.js`, instead).

Design and implementation of the Mix Server. The system consists in the cryptographic core of the mix server of sElect (see Chapter 8) and of a setup class with the method `main()` which reproduces, directly in Java, a cryptographic privacy game between the mix server and the environment/adversary:

1) the adversary/environment is asked to produce two vectors of messages, under the conditions that all messages are of the same length and the two vectors are permutations of the same set;
2) depending on a static boolean variable `secret` (the *high* value), one of the two vectors is picked;
3) the messages of the chosen vector are first encrypted individually with the mix server's public key, then the resulting cyphertexts are concatenated and signed with the sender's signing key;
4) the resulting string is then sent over the network to the mix server which, after checking its integrity, decrypts each message, checks for the absence of duplicates, shuffles (sorts) the decrypted messages, and finally outputs them in the proper format.

The code of the setup and of the core of the mix server of sElect can be found in Appendix E.3 and in [SHM17].

The Security Property. The most fundamental security property of the mix server is message anonymity, i.e, the impossibility to link its output to its input. To formulate this security property, we provide a setup class with the method `main()` which creates an instance of the mix server including its cryptographic functionalities (Public Key Encryption and Digital Signature), let the environment (adversary) determine two arrays of messages which must be a permutation of the same set and, depending on the static boolean variable `secret_bit`, one of the two arrays is picked. The concatenation of these messages, each of whom is encrypted with the mix server's public key, is then sent to the mix server which decrypts the messages, sorts, and outputs them in the proper format.

Message anonymity means that, independently from which one of the two arrays is picked, the attacker cannot tell the difference by observing the output of the mix server. Formally, this is expressed by the following computational indistinguishability property:

$$MS_{Real}[\texttt{false}] \approx^{\emptyset}_{\mathsf{comp}} MS_{Real}[\texttt{true}]. \tag{9.1}$$

That is, the two variants of the system are indistinguishable from the point of view of an adversary who implements the networking, but does not (directly) call methods of $MS_{Real}[\texttt{secret_bit}]$ with $\texttt{secret_bit} \in \{true, false\}$.

Verification Approach. Since the code falls into the Jinja+ fragment, by the results of the CVJ Framework, to prove (9.1), it is enough to show I-noninterference of the variant of

MS_{Ideal}[`secret_bit`], i.e., where the real functionalities has been replaced by the ideal ones and the `secret_bit` is considered to be the only high variable. That is:

$$\tilde{E}_{\vec{u}} \cdot MS_{Ideal}[\texttt{secret_bit}] \text{ is noninterferent for all } \vec{u}. \tag{9.2}$$

In principle, the automatic tool Joana presented in Section 4.5 is able to check properties such as (9.2). However, when applied to check (9.2) for our particular program, Joana reports an information flow from the high value `secret_bit` to the result of the mix server and from this result to the low output. The reason for this alert is the overapproximation that Joana employs. The result of the mix server does not actually depend on `secret_bit`, because `main()` in MS_{Ideal}[`secret_bit`] ensures that the two arrays of messages $\vec{m}_1$ and $\vec{m}_2$ are permutations of the same set. Hence, to avoid this false positive, an analysis tool has to establish that the result of the mix server corresponds to the sorted vector $\vec{m}_1$ (and, hence, to the sorted vector $\vec{m}_2$). This is a non-trivial functional property, which – not surprisingly – is beyond what Joana and all other fully automatic tools for checking noninterference can achieve.

In order to check (9.2), we therefore used the hybrid approach for proving noninterference proposed by Küsters et al. [KTB$^+$15]. We now briefly recall this hybrid approach in order to then apply it to the verification of the mix server.

9.2. A Hybrid Approach for Proving Noninterference of Java Programs

The problem of checking noninterference properties of programs has a long tradition in the field of computer security and, in particular, in language-based security [SM03]. Several tools and approaches exist in the literature for checking noninterference. Some approaches, such as type checking [VSI96, VS97], abstract interpretations [ACAE09], and program dependency graphs (PDGs) [HS09a], with tools including JIF [MCN$^+$01] and Joana (see Section 4.5) have an high degree of automation, but they overapproximate the actual information flow, and hence, may produce false positives. Other approaches—such as those based on theorem proving, with tools such as KeY [ABB$^+$05], Isabelle [Pau94], and Coq [BC04]—allow for precise analysis, but need human interaction, and hence, the analysis is often time-consuming (see, e.g., [AB04, BDR04, BNR08, SS12, Sch14]). Fully automatic tools are often preferable over interactive approaches since with such tools program analysis is typically less time-consuming and might require less expertise. However, if automatic tools fail due to false positives and the analysis cannot further be refined by these tools, because, for example, the tools do not allow this or run into scalability problems, the only option for proving noninterference so far is to drop the automatic tools altogether and instead turn to fine-grained but interactive, and hence, more time-consuming approaches, such as theorem proving. This "all or nothing" approach is unsatisfying and problematic in practice.

Therefore, in [KTB$^+$15], a tool independent hybrid approach has been proposed, which allows one to use (fully) automated verification tools for checking noninterference properties as much as possible and only resort to more fine-grained, but possibly interactive verification tools (typically theorem provers) at places in a program where necessary. The latter verification requires checking specific functional properties in (parts of) the program only, rather than checking the more

involved noninterference properties. In this way, the advantages of automated, but imprecise, and precise, but interactive tools are combined together to minimize the amount of work needed to prove the absence of illegal information flow.

The hybrid approach is also stated and proven for the language Jinja+ presented in Section 2.1. As also explained in the introduction, the idea underlying this approach is as follows: If the verification of noninterference of a program using an automatic tool fails due to (what we think are) false positives (i.e., the automatic tool falsely claims some illegal flow of information), then, following the rules of the approach, additional code is added to the program in order to make it more explicit and more clear for the automatic tool that there is no illegal information flow, and by this, avoid false positives. If the automatic tool now establishes that the extended program enjoys the desired noninterference property, it remains to show that the extended program is what it is called a *conservative extension* of the original program. Intuitively, this means that the additional code did not change the behavior of the original program in an essential way. Proving that an extension is conservative requires to prove *functional* properties of (parts of) the program and it is typically carried out by an (interactive) theorem prover. The central property shown for the hybrid approach is that if the extended program is noninterferent and is a conservative extension of the original program, then the original program is noninterferent as well.

Constructing a Conservative Extension. Given a program P for which an illegal information flow is reported by the automatic tool, we first provide an extension P' of P. We do this following the rule of the approach in such a way that

a) it is made (more) explicit for the automatic tool that there is no illegal information flow in P' and

b) P' extends P in what we call a *conservative* way.

In a nutshell, to construct a (conservative) extension of a program P, one adds an additional component M to the program P. This component is constructed in such a way that its state is isolated from the state of P. The component M is then used to collect some low data and explicitly "kill" potential illegal information flow paths, as explained below. More formally, an extension of a program P is defined in the following way.

Definition 9.1 (Extension [KTB+15]**).** *Let $P = P[\vec{x}]$ be a deterministic and closed (Jinja+) program. An* extension of *P is a program $P' = P'[\vec{x}]$ obtained from P in the following way.*

First, a new component M is added to P consisting of some number of classes with the following properties:

(i) the methods and fields of the classes in M are static,

(ii) the arguments and the results of the methods of M are of primitive types,

(iii) the methods of M do not refer to classes defined in P (in particular, no methods and fields of P are used in M),

(iv) all potential exceptions are caught inside M,

(v) all methods of M always terminate.

Second, P is extended by adding statements of the following form in arbitrary places within methods of P:

(a) Output to *M:*

$$C.f(e_1,\ldots,e_n) \tag{9.3}$$

where C is a class in M with a (static) method f and $e_1,\ldots,e_n$ are expressions without side effects.

(b) Input from *M:*

$$r = C.f(e_1,\ldots,e_n), \tag{9.4}$$

where C is a class in M, C.f is a (static) method with some (primitive) return type τ, $e_1,\ldots,e_n$ are expressions as above, and r is an expression that evaluates without side effects to a reference of type τ. (Such an expression can, for example, be a variable or an expression of the form o.x, where o is an object with field x.)

Now, one uses the automatic tool to verify that P' is noninterferent. If the automatic tool still fails to prove noninterference for P', because of another false positive, one can further extend P' in a conservative way, and so on. Once noninterference of P' is established, it remains to verify that P' is in fact a conservative extension of P'. Since a conservative extension is a *functional property*, to prove this property, the support of a more precise, but possibly interactive tool (e.g., a theorem prover) is needed. However, this should typically involve analyzing only a smaller fragment of the overall program. Being a functional property, this approach is more practical than to prove noninterference properties of the complete program.

If now noninterference and conservatism of P' is established, due to the way the extension is formally defined (see Definition 9.1) which implies (i) a *state separation* of the original program *P* and of the extension *M* and (ii) that the additional statements added to *P* do not change the state of *P*, we obtain noninterference of the original program *P*. More formally, in [KTB$^+$15] it has been stated and proven the following theorem.

Theorem 9.1. *([KTB$^+$15]) Let $P[\vec{x}]$ be a program with the variables $\vec{x}$ labeled as high and variables $\vec{y}$ labeled as low. Let $P'[\vec{x}]$ be a conservative extension of $P[\vec{x}]$ such that in $P'[\vec{x}]$ again the variables in $\vec{x}$ are labeled as high and those in $\vec{y}$ are labeled as low. Then, if $P'[\vec{x}]$ is noninterferent, then so is $P[\vec{x}]$.*

While the hybrid approach is widely applicable—it is not tailored to specific tools or specific applications, and the basic idea is quite independent of a specific programming language— in this thesis we combine it with the results of the CVJ framework (see Part I) to establish cryptographic privacy of the votes cast by sElect.

9.3. KeY, a Theorem Prover for sequential Java Programs

Although the hybrid approach is not tailored to any specific tools, to show its applicability, the authors of [KTB$^+$15] combined the fully automatic tool Joana (see Section 4.5) with the KeY interactive theorem prover.

KeY[27] [ABB$^+$05, ABB$^+$14, BHS07] is an integrated program verification system, which targets sequential Java. At its core lies an interactive theorem prover for first-order dynamic logic (JavaDL) [Bec00]. Program specifications can be given in the Java Modeling Language (JML) [LBR98]. KeY provides both a stand-alone graphical user interface, intended for interactive proofs, and an integration into the Eclipse platform, intended for push-button proofs hiding the underlying prover architecture [HKHB14].

In dynamic logic [FL79, Har84], programs π give rise to modal operators $[\pi]$ and $\langle\pi\rangle$. For instance, the formula $\varphi \rightarrow \langle\pi\rangle\,\psi$ intuitively means "if started in a state in which formula φ holds and program π terminates, then in the final state formula ψ holds". This means that the right-hand side of this implication is equivalent to the weakest precondition of π w.r.t. ψ. Replacing $\langle\cdot\rangle$ by $[\cdot]$ yields the weakest liberal precondition. Dynamic logic can be seen as a super-set of Hoare logic [Hoa69]. In contrast to Hoare triples, however, programs are an integral part of formulae. This allows one to write down more elaborate formulae, e.g., formulae with multiple programs or existential quantification ranging over program states.

In particular, this expressivity allows for relational properties, such as noninterference, to be expressed [SS12]. This technique has already been used to prove a simple e-voting system noninterferent [Sch14]. However, information flow analysis based on theorem proving is rather heavy-weight and requires a considerable amount of manual interaction, making its application on real-world programs cumbersome, tedious, and time consuming.[28]

The sequent calculus for JavaDL that is built into KeY precisely reflects the semantics of sequential Java, i.e., it does not use approximations. Thus, analysis techniques built on KeY are precise. They do not report any false positives. Proofs can be automatic to a certain degree, while the user can interact with the prover at any time. KeY can generate counter examples and unit tests from failed proof attempts. KeY is different from general purpose (first-order or higher-order) theorem provers (such as Isabelle [Pau94] or Coq [BC04]) in that KeY and its calculus are tailor-made for Java verification. The semantics of Java is 'built in.' The program to be analyzed is kept as part of formulas. Constructing a proof in KeY corresponds to *symbolic execution* [Kin76]. This helps to keep JavaDL formulas and proof goals human-readable and, thus, allows the user to understand the (sub-)proofs.

KeY supports *modular verification* based on the design by contract paradigm [Mey92]. Individual methods are verified w.r.t. their contract, i.e., independently from the environment. For every Java method, there is a separate proof. This ensures (preservation of) correctness of proofs even in case the implementation of some other method is unknown or changed. It is essential that contracts do not only contain pre- and post-condition pairs, which describe what an implementation is supposed to achieve, but also *frame conditions* [BMR93], which explicitly describe what an implementation does *at most* do, i.e., what it does beyond what is already specified in the post-conditions. Contracts are given in JML which integrates seamlessly into Java as it appears in Java comments and uses a superset of Java expressions. An example on how to annotate Java code with JML contracts can be found in Figure 9.1 of Section 9.4.

We now present the first Java system, namely an e-voting machine with auditing procedures,

[27] KeY is free software and can be downloaded from `http://key-project.org/`.

[28] The tediousness and inconvenience of using theorem provers to check noninterference properties was one of the reasons why the hybrid approach has been conceived by the authors of [KTB$^+$15].

which we implemented with the aim of combining the fully automatic tool Joana (see Section 4.5) and the KeY theorem prover to establish noninterference properties in the way proposed by the hybrid approach. We note that this system has also been used as case study in [KTB+15].

Design and implementation of the E-voting Machine. The system involves a *voting machine* which collects each voter's choice an sends, encrypted, each vote plus some additional data for auditing to an append-only *bulletin board*. Besides computing the election result and publishing it to the bulletin board, the voting machine can also be triggered to publish the complete list of internal (encrypted) log, also for auditing purposes. In fact, this voting machine maintains a vote counter increased every time a ballot is cast and an operation counter increased every time an operation is internally performed. The value of the latter counter is returned when a ballot is cast. In addition, the voting machine allows for vote canceling: auditors can submit a vote and then immediately delete it (clearly, the delete operation only increases the operation counter). In this way, auditors can check that an entry which carries the same operation counter as the one given to the auditors when they voted is indeed added to the bulletin board. By asking the machine to output the internal log, the auditors can also make sure that the voting machine properly logged this entry.

Besides the code of the voting machine and of the bulletin board, the program also contains a setup class with the method `main` which defines a cryptographic privacy game, similar to games in cryptography, except that this game is formulated in Java. To define this game, let ρ be a result function which takes a multiset (or a vector) of choices/candidates and returns a result vector $\vec{v}$, i.e., for every choice, $\vec{v}$ contains an entry with the number of occurrences of this choice in the given multiset. In the privacy game the environment (the adversary) can provide two vectors $\vec{c}_0$ and $\vec{c}_1$ of choices of (honest and dishonest) voters such that the two vectors yield the same result according to ρ, i.e., $\rho(\vec{c}_0) = \rho(\vec{c}_1)$; otherwise the game is stopped immediately. Now, the voters vote according to $\vec{c}_b$, where b is a secret bit. The environment tries to distinguish whether the voters voted according to $\vec{c}_0$ or to $\vec{c}_1$. In other words, it tries to determine the secret bit b. We denote the Java program describing this game by $\mathsf{EV}^{real}[\texttt{secret_bit}]$, with b being the only secret/high input.

The code of the setup, of the voting machine, and of the bulletin board can be found in Appendix E.2 and in [STB+14a].

The Security Property. The most fundamental security property each e-voting system is supposed to provide is of course privacy of the votes. More precisely, as reported in [KTB+15], the security property for which we designed this e-voting machine is cryptographic privacy of the votes of honest voters. Formally, this property is expressed by the following computational indistinguishability property (see Part I):

$$\mathsf{EV}^{real}[\texttt{false}] \approx^{\emptyset}_{\mathsf{comp}} \mathsf{EV}^{real}[\texttt{true}]. \tag{9.5}$$

That is, the two variants of the system are indistinguishable from the point of view of an adversary who implements the network, but does not call (directly) methods of $\mathsf{EV}^{real}[\texttt{secret_bit}]$. However, through the setup class, the adversary determines whether:

1. the next voter casts her ballot;

2. an auditor casts a vote and then cancels it;

3. the voting machine outputs its internal log;

4. a message coming from the network is delivered to the bulletin board;

5. the bulletin board outputs its internally stored entries.

Once all the voters cast their ballots, the voting machine outputs the election result (see Appendix E.2 for the specification of the setup).

By the result of the CVJ framework and, in particular, by Theorem 2.2, to prove (9.5) it is enough to show that

$$\mathsf{EV}^{ideal}[\texttt{secret_bit}] \text{ is } I\text{-noninterferent,} \tag{9.6}$$

where $I = \emptyset$ and EV^{ideal} denotes the system which coincides with EV^{real} except that the real cryptographic operations are replaced by their ideal counterparts. More specifically, the real cryptographic operations for public-key encryption and digital signatures are replaced by ideal functionalities for these primitives, as provided in Chapter 4. The realization result in Chapter 4 shows that the real cryptographic operations realize the ideal functionalities. Hence, these operations can indeed be replaced by their ideal counterparts.

Since, as can easily be seen, $\mathsf{EV}^{ideal}[\texttt{secret_bit}]$ satisfies the conditions of Theorem 2.4, we can further reduce checking (9.6) to checking the following property:

$$\tilde{E}_{\vec{u}} \cdot \mathsf{EV}^{ideal}[\texttt{secret_bit}] \quad \text{is noninterferent for all } \vec{u}, \tag{9.7}$$

where the family of systems $\tilde{E}_{\vec{u}}$, parameterized by a finite sequence of integers $\vec{u}$, is as introduced in Section 2.6. This system can be automatically generated from $\mathsf{EV}^{ideal}[\texttt{secret_bit}]$.

Verification of the Security Property. In order to establish cryptographic privacy properties for this Java system it suffices, according to the CVJ framework, to verify that $\tilde{E}_{\vec{u}} \cdot \mathsf{EV}^{ideal}[\texttt{secret_bit}]$ is noninterferent. To establish noninterference, the fully automatic tool Joana presented in Section 4.5 has been employed. However, due to the functional dependency between the secret voters' choices and the public outcome of the election, this tool produces a false positive. Since establishing noninterference for the case study requires indeed to prove a *functional property* of the e-voting machine, it is quite likely that all fully automatic tools would fail. The system has therefore been extended with a conservative extension to avoid the false positive. Joana can then easily prove that this extension indeed is noninterferent. Finally, to prove that the extension is conservative, the software verification system KeY presented in this section has been used. By the hybrid approach, this implies that the system (running with ideal functionalities) is noninterferent. The CVJ framework then immediately yields cryptographic privacy of the Java system when the ideal functionalities are replaced by the actual cryptographic implementations.

9.4. Applying the Hybrid Approach to Verify the Mix Server

Following the hybrid approach presented in Section 9.2 and the first case study where it has been applied (see Section 9.3), in order to prove the property (9.2) for the mix server of sElect, we provide an extension $MS^*_{Ideal}[\texttt{secret_bit}]$ of $MS_{Ideal}[\texttt{secret_bit}]$ which makes it more explicit for Joana that there is no information flow. In the extension $MS^*_{Ideal}[\texttt{secret_bit}]$, we explicitly state that what the mix server has to output is indeed the sorted array of messages $\vec{m}_1$. More specifically, we obtain $MS^*_{Ideal}[\texttt{secret_bit}]$ from $MS_{Ideal}[\texttt{secret_bit}]$ as follows: The array $\vec{m}_1$ is sorted and stored on a static variable `correctOutput`. After the point in the code of the mix server where the decrypted messages are sorted and stored in a local variable x, we add the assignment $x = \texttt{correctOutput}$. If the mix server is implemented correctly, then the result computed by the mix server (the list of decrypted and sorted messages) indeed is the same as the result stored in `correctOutput` and, therefore, the additional assignment does not change the state of the program. This is what in [KTB+15] is referred as a conservative extension.

Joana is now able to check noninterference of $\tilde{E}_{\vec{u}} \cdot MS^*_{Ideal}[\texttt{secret_bit}]$ for all $\vec{u}$. To show $\tilde{E}_{\vec{u}} \cdot MS_{Ideal}[\texttt{secret_bit}]$ noninterferent, what it is still remaining to be proven is that

$$\tilde{E}_{\vec{u}} \cdot MS^*_{Ideal}[\texttt{secret_bit}] \text{ is a conservative extension of } \tilde{E}_{\vec{u}} \cdot MS_{Ideal}[\texttt{secret_bit}] \text{ for all } \vec{u}. \tag{9.8}$$

We note that that this statement can be expressed with a *functional property* on $\tilde{E}_{\vec{u}} \cdot MS^*_{Ideal}[\texttt{secret_bit}]$: the syntactic extensions in $MS^*_{Ideal}[\texttt{secret_bit}]$ are redundant, i.e., they do not semantically change the system $MS^*_{Ideal}[\texttt{secret_bit}]$ respect to the system $MS_{Ideal}[\texttt{secret_bit}]$. We refer the reader to Appendix E.3 for the declaration of the system $MS^*_{Ideal}[\texttt{secret_bit}]$.

Following [KTB+15], we use the theorem prover KeY to prove the conservatism. Proving the aforementioned functional property with KeY actually consists of two separate parts:

(a) We have to show that the result of the mix server is indeed a permutation of the list of messages the mix server received, encrypted, as input.

(b) We have to show that certain invariants expressing well-formedness of the data structures of the ideal functionalities hold and that there exists a relation between the input of the mix server and the actual data structures from which the result of the mix server is computed.

The analysis with KeY of Part (a) is finished. This part involves 8 classes with 73 methods. These methods were annotated with 761 lines of Java Modeling Language specification. The complete KeY proof for Part (a) consists of 1 576 647 nodes in the proof trees (corresponding to rule applications), 6 402 of which were *interactive* (manual) steps required to construct these proofs. In addition, 6 new reductions rules have been added to the theorem prover. KeY needs about 1 hour for the automated parts of proof construction on a standard desktop PC.

Although the proofs for Part (b) are, in principle, less complex than those for Part (a), there are more sub-proofs to consider. Thus, due to scalability issues, the generation of the proofs for Part (b) requires more manual interaction than the generation of the proofs for Part (a): While

straightforward, the analysis becomes then more tedious. At the time of writing, the proofs for Part (b) are mostly finished but ongoing work.

```
/*@ public normal_behaviour
  @ requires \dl_array2seq(ballots) == \dl_arrConcat(0, \dl_array2seq2d(sorted));
  @ requires n == sorted.length;
  @ ensures  \dl_array2seq2d(\result) == \dl_array2seq2d(sorted);
  @ ensures  \fresh(\result);
  @ assignable \nothing;
  @*/
public byte[][] split(int n, byte [] ballots){
    byte[][] messages = newArray(n);
    byte[] bal  = ballots;
    int i = 0;
    /*@ loop_invariant 0 <= i && i <= n && n == messages.length
      @                && n == sorted.length && messages != sorted;
      @ loop_invariant \dl_array2seq(bal) ==
      @                \dl_arrConcat(i, \dl_array2seq2d(sorted)) && bal != null;
      @ loop_invariant messages != null && (\forall int k; 0 <= k
      @                && k < messages.length; messages[k] != null);
      @ loop_invariant \fresh(messages);
      @ loop_invariant (\forall int j; 0 <= j && j < i;
      @                \dl_array2seq(messages[j]) == \dl_array2seq(sorted[j]));
      @ assignable messages[*], bal;
      @ decreases n - i;
      @*/
    while(i < n){
        bal = split(messages, bal, i);
        i++;
    }
    return messages;
}
```

Figure 9.1.: JML annotated method to split the concatenated ballots (this method relies on a recursive method `split` not shown here). JML annotations appear as comments with the special delimiter pair `/*@ @*/`. Lines 1-6. (before the method signature) state the contract. Lines 2 and 3 impose a precondition, while lines 4 and 5 define a postcondition. Termination and absence of exceptions is included by default as stated by `normal_behavior`. Since the method contains an unbounded loop, we need to annotate that as well in order to prove the contract. A loop invariant is given in lines 12-20. To prove termination, we also have a variant clause in line 22.

To give a feel of the effort necessary to perform an analysis using KeY, Figure 9.1 displays one of the Java methods that had to be analyzed: this method splits the concatenated ballots the mix server receives as input. The code to be analyzed needs to be annotated with specifications, i.e., pre- and post-conditions as well as loop invariants, which then need to be proven by KeY. The latter, in general, might again require some human interaction if the prover cannot verify the

conditions automatically. The annotated program and the proofs already generated with KeY are available at [SHM17].

As mentioned above, for the conservative extension $\tilde{E}_{\vec{u}} \cdot MS^*_{Ideal}[\mathtt{secret_bit}]$, Joana easily establishes the noninterference property

$$\tilde{E}_{\vec{u}} \cdot MS^*_{Ideal}[\mathtt{secret_bit}] \text{ is noninterferent for all } \vec{u}. \tag{9.9}$$

Joana took about 5 seconds on a standard PC (Core i5 2.3GHz, 8GB RAM) to finish the analysis of the program (with a size of 726 LoC). PDG computation took 3 seconds and only 2 seconds were needed to detect the absence of illegal flow inside the PDG. Note that the actual running code of the mix server is much bigger than what Joana needed to analyze, because the code embedded in the mix server of sElect includes untrusted libraries, such as a Java library for cryptographic operation, which do not need to be analyzed, as already mentioned in Section 2.4.

Therefore, once the analysis with KeY is completed, by Theorem 9.1, we conclude that the property (9.2) holds true. Thus, by the result of the CVJ framework, the cryptographic privacy property (9.1) of the mix server follows as well.

Other possible approaches. KeY is able to prove noninterference properties itself, by using the technique of self-composition [BDR04]. In particular, KeY would in principle be able to establish the property (9.2) without the hybrid approach and therefore without the help of Joana or any other fully automatic tool. However, verifying the code of the mix server only with KeY would have eventually run in scalability problems due to the overwhelming amount of deductive steps to generate proofs. More generally, as already reported in [KTB$^+$15], checking noninterference properties directly using interactive theorem provers would be very complex and time consuming and, in fact, seems to be impractical for any realistic Java program.

There is another possible way of combining the high precision of deductive verification tools with the efficiency of fully automatic static checkers: while in realistic programs it can happen that the verification with interactive tools becomes infeasible, at the same time, parts of these programs tend to be irrelevant with respect to the functional property to be proven. To remedy this, a general technique called *spec slicing* has also been investigated in [KTB$^+$15]. The idea behind spec slicing is the following: If parts of the program do not influence the final state w.r.t. the proof obligation, they can be safely removed and function verification can be performed on the simpler program. Verification of this simpler program can then be performed without any loss of precision but with possibly much less effort. The identification and removal of irrelevant program parts can be performed by automatic tools based on program dependency graph (PDG) like Joana. As reported in [KTB$^+$15], this technique has already been applied to the e-voting machine described in Section 9.3: parts of its implementation perform mere logging, which does not affect the voting result and hence does not have any influence on the functional property which has to be verified. The program dependency graph computed by Joana is used to read off the separation of this component and mark it not relevant for the property to be verified. In this way, KeY is used to prove the functional property on a simplified version of the program which requires less reduction steps and it is therefore also less time consuming.

10. Related Work and Discussion

In this chapter, we first briefly mention further related work and then we provide a more detailed discussion of the main features of sElect, including the limitations of the system.

The basic idea of combining the choice of a voter with an individual verification code has already been mentioned in an educational voting protocol by Schneier [Sch96].

The F2FV boardroom voting protocol [ACW13] is based on the concept of verification codes too. In that protocol, it is assumed that all voters are located in the same room and use their devices in order to submit their vote-nonce pairs as plaintexts to the bulletin board. As pointed out in [ACW13], F2FV is mainly concerned with verifiability, but not with privacy.

Several remote e-voting protocols have been proposed in the literature (see, e.g., [RS07, Adi08, CCM08, RBH$^+$10, ZCC$^+$13, KZZ15a, GRCC15, RRI16]), with Helios [Adi08] being the most prominent one.

We now first briefly sketch how Helios works and then in each paragraph but the last we compare the features of sElect with those of Helios. In the last paragraph, we instead briefly discuss the other cryptographic analyses performed on e-voting systems.

The Helios e-voting system. A voter, using her browser, submits a ballot (along with specific zero-knowledge proofs) to a bulletin board. Afterwards, in the tallying phase, the ballots on the bulletin board are tallied in a universally verifiable way, using homomorphic tallying and verifiable distributed decryption. Helios uses so-called "Benaloh challenges" [Ben06] to ensure that browsers encrypt the actual voters' choices (cast-as-intended). For this, the browser, before submitting the ballot, asks whether the voter wants to audit or cast the ballot. In the former case, the browser reveals the randomness used to encrypt the voter's choice. After that, the voter should copy/paste this information to another (then trusted) device to check that the ballot actually contains the voter's choice. The voter is also supposed to check that her ballot appears on the bulletin board, which together with the homomorphic tallying and verifiable distributed (threshold) decryption then implies that the voter's vote has been properly counted.

The protocol of Helios follows the general ideas of Cramer, Gennaro, and Schoenmakers [CGS97] where the ElGamal cryptosystem [ElG85] is employed both for homomorphic tallying and distributed decryption. To provide a non-interactive zero-knowledge proof of decryption, Helios uses the Chaum-Pedersen protocol [CP92] in combination with the Fiat-Shamir heuristic [FS86]. Furthermore, in the latest version of Helios, namely Helios v4, the data structures of the system have been modified to support, besides homomorphic tallying, a more flexible mixnet-based tallying. While this feature has not been properly integrated into the Helios master branch yet, the mix net workflow and the employable cryptographic protocols have already been discussed and tested in [BGP11].

Fully automated verification. Fully automated verification is a main and unique feature of sElect, which would also be very useful for other systems, such as Helios. This kind of verification

is performed without any interaction required from the voter, and hence, is completely transparent to the user. In particular, the voter does not have to perform any cumbersome or complex task, which thus eases the voter's experience. This, and the fact that fully automated verification is triggered when the voter visits the voting booth again (to later look up the election result on the bulletin board), should also help to improve verification rates, as hinted at by the two small mock elections discussed in Section 8.3. Moreover, this kind of verification importantly also provides a high-level of accountability, as it is proven in [KMST16a, KMST16b].

Obviously, for fully automated verification we need to assume that (most of) the VSDs can be trusted. Recalling from Section 8.2 we note that in our implementation of sElect a VSD consists of the voter's computing platform (hardware, operating system, browser) and the voting booth (server), where the idea is that the voter can choose a voting booth she trusts among a set of voting booths.

As mentioned, we assume low-risk elections (e.g., elections in clubs and associations) where we do not expect targeted and sophisticated attacks against voters' computing platforms.[29]

Also, as mentioned in Section 8.2, the idea is that several voting booth services are available, possibly provided by different organizations and independently of specific elections, among which a voter can choose one she trusts. So, for low-risk elections it is reasonable to assume that VSDs are trusted. In addition, voter-based verification provides some mitigation for dishonest VSDs (see also the discussion below).

It seems that even for high-stake and high-risk elections some kind of fully automated verification might be better than completely relying on actions performed by the voter, as is the case for all other remote e-voting systems. So other systems should profit from this approach as well.[30]

Voter-based verification (human verifiability). The voter-based (manual) verification can be seen as an orthogonal mechanism as the fully automated verification procedure, which does not assume trust in other parties or devices (except that the voter needs to be able to look at the election outcome). This gives the voters direct understanding that their votes were actually counted.

The level of verifiability provided by voter-based verification (manual checking of voter-generated verification codes) has been analyzed in detail in [KMST16a, KMST16b].

On the positive side, voter-based verification provides a quite good level of verifiability, with the main problem being clashes. With voter-based verification the voter does not have to trust any device or party, except that she should be able to look up the *actual* election outcome on a bulletin board, in order to make sure that her vote was counted (see also below). In particular, she does not have to trust the voting booth (she chose) at all, which is one part of her VSD. Moreover, trust on the voter's computing platform (hardware, operating system, browser), which is the other part of her VSD, is reduced significantly with voter-based verification: in order to hide manipulations, the voter's computing platform would have to present a fake election outcome to the voter. As mentioned before, our underlying assumption is that (for low-risk elections) such targeted attacks

[29]For high-stake elections, such as national elections, untrusted VSD are certainly a real concern. This is in fact a highly non-trivial problem which has not satisfactorily been solved so far when both security and usability are taken into account (see, e.g., [GRCC15]).

[30]For high-risk elections one might have to take extra precautions for secretly storing the voter's receipt in the voter's browser or on her computer.

are not performed on the voter's computing platform. (Of course, voters also have the option to look up the election result using a different device.)

Voter-based verification is also very easy for the voter to carry out and the voter easily grasps its purpose. In particular, she can be convinced that her vote was actually counted without understanding details about the system, e.g., the meaning and workings of universally verifiable mix nets or verifiable distributed decryption. In other systems, such as Helios, voters must have trust in the system designers and cryptographic experts in the following sense: when their ballots appear on the bulletin board, then some universally verifiable tallying mechanism—which, however, a regular voter does not understand—guarantees that her vote is actually counted. Also, other systems require the voter to perform much more complex and cumbersome actions for verifiability and they typically assume a second trusted device in order to carry out the cryptographic checks, which altogether often discourages voters from performing these actions in the first place. For example, Helios demands voters

i) to perform Benaloh challenges [Ben06];

ii) to check whether their ballots appear on the bulletin board.

However, regular voters often have difficulties understanding these verification mechanisms and their purposes, as indicated by several usability studies (see, e.g., [AKBW14, KOKV11, KKO$^+$11, NORV14, OBV13, WH09]). Therefore, many voters are not motivated to perform the verification, and even if they attempt to verify, they often fail to do so. Furthermore, the verification process, in particular the Benaloh challenge, is quite cumbersome in that the voter has to copy/paste the ballot (a long randomly looking string) to another, *then trusted*, device in which cryptographic operations need to be performed. If this is done at all, it is often done merely in a different browser window (which assumes that the voter's platform and the JavaScript in the other window is trusted), instead of a different platform.

On the negative side, verification codes could be easily misused for coercion (see the "Mitigating coercion resistance" paragraph). A voter could (be forced to) provide a coercer with her verification code *before* the election result is published, and hence, once the result is published, a coercer can see how the voter voted.[31]

We note, however, that in any case, for most practical remote e-voting systems, including sElect and, for instance, Helios, there are also other simple, although perhaps not as simple, methods for coercion. Depending on the exact deployment of these systems, a coercer might, for example, ask for the credentials of voters, and hence, simply vote in their name. Also, voters might be asked/forced to cast their votes via a (malicious) web site provided by the coercer, or the coercer asks voters to run a specific software. So, altogether preventing coercion resistance is extremely hard to achieve in practice, and even more so if, in addition, the system should still be simple and usable. This is one reason that coercion-resistance was not a design goal for sElect.

Mitigating coercion resistance. The sElect voting system, just as Helios, was not designed to provide coercion resistance and, in fact, in the current implementation, vote selling and voter

[31] In a quite recent work, a mitigation for this problem has been considered [RRI16]. However, this approach assumes, among others, a public-key infrastructure for all the voters.

coercion is quite easy: a voter can simply forward her verification code to a coercer who can use this code to check which candidate the voter voted for.

Still, to mitigate this problem to some extent and to make coercion less easy, one can consider the following variants of sElect.

First, we can consider a variant where voter-based verification is dropped and the verification codes are not shown to the voter (but only stored internally in the browser's local storage). This, of course, means that the verification would be done only automatically by the voting booth; the voter could not carry out manual verification.

One can also consider a variant where the whole receipt, including the verification code (again the voter's part of the code would be dropped), is not computed and stored within the browser but on the server site of the voting booth. In this variant, as opposed to the implemented variant, the voting booth server plays an active role. In particular, it would perform the verification procedure itself (or delegate this to another party).

Note that without trusting the voting booth, coercion resistance and even privacy would be much harder to achieve. We emphasize that the voter is free to choose a voting booth she trusts, and as discussed before, for low-risk and low-coercion elections trusting the voting booth (and the client platform) will in many cases be reasonable.

Simple cryptography and design. Unlike other modern remote e-voting systems, sElect employs only the most basic and standard cryptographic operations, namely, public key encryption, digital signatures, and nonce generation, while all other verifiable remote e-voting systems use more sophisticated cryptographic operations, such as zero-knowledge proofs, verifiable distributed decryption, universally verifiable mix nets, etc. The overall design and structure of sElect is simple as well. The motivation for our design choices were twofold: Firstly, we wanted to investigate what level of security (privacy, verifiability, and accountability) can be achieved with only the most basic cryptographic primitives and a simple and user-friendly design. Secondly, using only the most basic cryptographic primitives has several advantages:

1. The implementation can use standard cryptographic libraries and does not need much expertise on the programmers side. In fact, simplicity of the design and of the implementation task is valuable in practice in order to avoid programming errors, as, for example, noted in [AKBW14].

2. The implementation of sElect is also quite efficient in terms of implementation and performances as already pointed out in Section 8.1.

3. sElect does not rely on setup assumptions. In particular, unlike other remote voting systems, we do not need to assume common reference strings (CRSs) or random oracles. We note that in [KZZ15a] and [KZZ15b] very complex non-remote voting systems were recently proposed to obtain security without such assumptions.

4. Post-quantum cryptography could easily be used with sElect, as cryptographic algorithms for public key encryption and digital signature that are secure against attacks of quantum computers are already available (see, for example, the NTRU algorithm [HPS98] and its Stehlé–Steinfeld version [SS11] for public-key encryption).

5. In sElect, the space of voters' choices can be arbitrarily complex since, if hybrid encryption is employed, arbitrary bit strings can be used to encode voters' choices; for systems that use homomorphic tallying (such as Helios) this is typically more tricky and requires to adjust the system (such as certain zero-knowledge proofs) to the specific requirements.

On the downside, with such a very simple design one does not achieve certain properties which could be obtained with more advanced constructions. For example, sElect, unlike for instance Helios, does not provide universal verifiability (by employing, for example, verifiable distributed decryption or universally verifiable mix nets). Universal verifiability can offer more robustness as it allows one to check (typically by verifying zero-knowledge proofs) that all ballots on the bulletin board are counted correctly. Every voter still has to check, of course, that her ballot appears on the bulletin board and that it actually contains her choice (cast-as-intended and individual verifiability).

Since sElect employs Chaumian mix nets, a single server could refuse to perform its task, and hence, block the tallying. Clearly, those servers who deny their service could be blamed, which in many practical situations should deter them from misbehaving. Therefore, for low-risk elections targeted in this work, we do not think that such a misbehavior of mix servers is a critical threat in practice. Other systems use different cryptographic constructions to avoid this problem, namely, threshold schemes for distributed decryption and (universally verifiable) reencryption mix nets.

Bulletin board. We finally note that in the implementation of sElect, we consider an (honest) bulletin board. This has been done for simplicity and is quite common; for example, the same is done in Helios.[32] The key property required is that every party has access to the bulletin board and that it provides the same view to everybody. This can be achieved in different ways, e.g., by distributed implementations, blockchain technologies, and/or observers comparing the (signed) content they obtained from bulletin boards (see, e.g., [CS14]); such approaches are orthogonal to the rest of the system, though.

Cryptographic analysis of e-voting systems. In Chapter 6, we have introduced the most fundamental security properties modern e-voting systems are supposed to achieve. As already mentioned in the introduction of the thesis, the formalization of these security properties has been used to perform cryptographic analyses on several e-voting systems finding attacks and revealing some misconceptions concerning their relations (see, e.g., [KTV10a, KTV10b, KTV10c, KTV11, CS11, KTV12b, ACW13, CEK+15, KZZ15a, KMST16a, CCFG16, KZZ17, CDD+17, CW17]).

However, such cryptographic analyses are typically performed on the design level, i.e., they do not analyze the actual implementation of the e-voting systems, but only their higher-level protocol. As already reported in Chapter 9, performing code-level verification on e-voting systems is far from being trivial since analyses typically depend upon the combination of different kinds of tools, some of whom requiring extensive human interaction for being properly employed.

The only analysis which has successfully been performed on the code of an e-voting system so far is the analysis on the e-voting machine which we have indeed implemented for this kind of

[32] To partially cope with dishonest bulletin boards, in [CGGI14] it has been proposed a modified version of Helios, called Helios-C, where a registration authority creates public/private key pairs for all voters. Voters sign their ballots in order to prevent ballot stuffing even if the bulletin board is dishonest.

cryptographic verification (see Section 9.3 and [KTB+15]). The verification of the mix server of sElect presented in Sections 9.1 and 9.4 shows that performing such analyses on (even parts of) real-world e-voting systems remains a tedious and almost impractical task at the current status of development of the employed tools.

11. Conclusion and Future Work

In this thesis, we have addressed the problem of establishing semantically sound cryptographic guarantees on the implementation of systems that are coded in Java and use cryptography.

We have extended and properly instantiated a framework for the cryptographic verification of Java programs, the CVJ framework, to make it applicable to a wider range of cryptographic software, i.e., software that uses cryptographic operations. We extended Jinja+, the Java-like formal language the framework is stated and proven for, with some not yet modeled features commonly used in Java applications: Java-interfaces, abstract classes, and the data type `String` in Chapter 2; two concurrency features of Java, namely thread creation and synchronized blocks, in Chapter 3. Interesting future work would consist in enlarging SyncJinja+ with the other three types of synchronization mechanisms, namely the `wait-notify` mechanism, thread joining, and interrupts, in order to be able to exhaustively model all the possible multi-threaded Java programs. However, modeling these other types of synchronization mechanisms would most likely not change any result of the CVJ framework restated for our proposed multi-threaded language.

In Chapter 3, we have formally linked the notion of computationally indistinguishability extended to multi-threaded SyncJinja+ systems to a novel noninterference definition also for multi-threaded SyncJinja+ systems. This notion substantially differs from all the other concurrent noninterference definitions in the literature, such as possibilistic and probabilistic noninterference. That is, we add yet another formalization of concurrent noninterference to the plethora of notions already stated in the literature (see, e.g., [PHN12, PHN13] for studies discussing and relating them). Although our impression is that our proposed notion of noninterference is less strict than those considered from the literature so far (which typically impose some form of simulation, bisimulation, or equivalence among all possible traces), it would be interesting to investigate how our notion formally relates with these other notions. For this purpose, one would have to first generalize our notion to a more general computational model than the specific Java-like language and then cast the different concurrent noninterference definitions in an unified and consistent framework, in a similar way as done, for example, in [PHN12].

The lack of a unique and commonly agreed definition of concurrent noninterference unfortunately also results in the scarcity of tools for checking this property in multi-threaded programs. Joana is the only fully automatic verification tool allowing one to check some noninterference properties in multi-threaded Java programs. In particular, Joana checks a specific criterion called Low-Security Observational Determinism [RWW94, ZM03] which then implies probabilistic noninterference for concurrent programs [VS99]. However, as already mentioned in Chapter 5, even relaxed [GS15] and improved [BGH+16] versions of this criterion are still too strict for being employed in the fully automatic verification of any realistic multi-threaded Java application (see, e.g., the case study in [BGH+16]). Establishing criteria to check our less strict concurrent noninterference definition and implementing them on verification tools for Java programs, such

as Joana, would make the CVJ framework more easily applicable to also multi-threaded Java applications.

In Chapter 4, we have instantiated the CVJ framework with the most common cryptographic operations used in security critical applications: digital signatures and public-key encryption, both also including a public-key infrastructure, private symmetric encryption, and nonce generation. For each real functionality implemented in Java, we have provided a corresponding ideal functionality also coded in Java as well as a formal proof that the real functionality realizes its ideal counterpart in the universal composability model and under strong cryptographic assumptions. Real and ideal functionalities for message authentication codes (MACs), different kinds of Diffie-Hellmann key exchanges with perfect forward secrecy, and key derivation have already been proposed in the universal composability framework (see, e.g., [KT11a, KT11b, KR17]), but in a more generic Turing machine model. Providing a formulation of these functionalities in the practical programming language Java, as well as their realization results, would be another interesting future work. Based on these cryptographic functionalities, one could then build higher-level protocols, such as secure and/or authenticated channel protocols, directly in Java. Thanks to the high modularity offered by the composition theorems (Theorems 2.1 and 3.1), to prove security properties of more complex protocols in the universal composability model, one can rely on the realization results proposed for the simpler cryptographic functionalities, without having to carry out every time reduction proofs from the high-level protocols down to the specific cryptographic primitives employed.

To illustrate the usefulness of this approach, two verification tools for proving noninterference of Java programs have been employed: the static checker Joana (see Section 4.5) and the theorem prover KeY (see Section 9.3). Certainly, fully automatic tools like Joana are preferable over interactive approaches, such as those based on theorem provers like KeY. However, if a fully automatic tool fails in proving noninterference due to false positives and the analysis cannot be further refined, instead of dropping this tool altogether, the hybrid approach proposed in [KTB+15] allows one to use the automatic tool as much as possible and only resort to a more precise (but also more time consuming) interactive tool at the point in the program where necessary. This approach has been used in combination with the CVJ framework to establish cryptographic vote privacy of an e-voting machine (see Section 9.3) and of the mix server of the sElect e-voting system (see Section 9.4). To properly verify these case studies, several practical improvements in the KeY verification system have been implemented and presented in [SS12] and [Sch14].

Nevertheless, previous experiments on larger and structurally more complex e-voting systems have shown that using the hybrid approach in combination with the CVJ framework to establish privacy properties of non-trivial cryptographic applications remains a challenging and tedious task, mainly because the employed tools are not mature enough to tame the intrinsic entanglement of such applications. The code of these experiments can be found in [STB+14b]. Further developing reduction rules and heuristics within KeY to more efficiently prove complex functional properties, such as trace properties involving the different components of the system under verification, would increase the applicability of the tool to more realistic Java programs as well. In the context of easing the verification process, another interesting future work would be to further enhance the synergy between theorem provers and automatic tools: On the one hand, theorem provers can be

used to eliminate the false positives reported by automatic tools. On the other hand, automatic tools could also assist theorem provers to recognize the parts of the code which are irrelevant w.r.t. the functional properties to be proven and which can hence be safely sliced off during the verification process.

Altogether, further extending and instantiating the CVJ framework as well as improving the tools for its application takes steps forward in providing implementations of cryptographic protocols which come with strong cryptographic security guarantees.

In Chapter 8, we have also proposed a new practical e-voting system, sElect, which starting from simply being a case study for the techniques developed within CVJ framework eventually evolved into a full-fledged and already deployed remote e-voting system. sElect, which is intended to be used in low-coercion environments, provides a number of novel features and, compared to other remote e-voting systems, it is designed to be particularly simple and lightweight in terms of its structure, the cryptography it uses, and the user experience. Some of the novel features of sElect, such as the fully automated procedure for the seamless verification of proper vote counting, can probably also be integrated into other systems or might inspire new design.

In the description of the proposed voting system (Section 8.1), we assume the existence of an anonymous channel from the voting platform to the bulletin board so that, in case a component misbehaves (removes or manipulates ballots), blaming evidence of this misbehavior is anonymously published on the bulletin board. In the implementation of the system, we do not however provide such a channel (see Section 8.2 and [SST16]). In case the voting platform is implemented as a static web-page, an anonymous channel could only be set up by providing a proxy server which receives the blaming evidences and forwards them to the bulletin board. App-based voting platforms could instead use the Tor anonymity network for this purpose. Another interesting future work related to the implementation of sElect concerns the extension of both the voting protocol and the related web interface to also support elections with multiple questions, different question types, and/or different "result functions" (algorithms to weigh the votes in the final result).

sElect can easily be deployed on a single server instance using an election manager that we have designed and implemented in [SS16]. However, to properly provide the security properties sElect is supposed to guarantee, the different components of the system have to be deployed on different machines, each of whom independent and not cooperating with the others. In this context, it would also be interesting to build a distributed, anonymous network consisting of several mix servers run by different, independent entities: when an election is closed, a routing algorithm in the collecting server would set up a pathway through the mix net to properly dispatch the result of the election to the bulletin board.

While we have already deployed sElect in three small field studies, namely two mock elections and one real-world election (see Section 8.3), another relevant future work would be to perform a systematic and broad usability study and to try out sElect in bigger and more relevant elections.

Finally, in Chapter 9, we have combined the techniques of the CVJ framework with the hybrid approach for proving noninterference to establish cryptographic vote privacy directly on the implementation of our proposed voting system. sElect is therefore the first full-fledged remote voting system for which strong cryptographic security guarantees have been formally established on the code level to ensure confidentiality of the cast votes.

PART III

Appendices

A. Security Notions for Cryptographic Schemes

In this chapter we present the cryptographic definitions to state the security of encryption and signatures schemes.

Beforehand, we briefly recall what it means for a function to be negligible, overwhelming, and bounded. As usual, a function f from the natural numbers to the interval $[0,1]$ is *negligible* if, for every $c > 0$, there exists ℓ_0 such that $f(\ell) \leq \frac{1}{\ell^c}$ for all $\ell > \ell_0$. The function f is *overwhelming* if the function $1 - f$ is negligible. A function f is λ*-bounded* if, for every $c > 0$ there exists ℓ_0 such that $f(\ell) \leq \lambda + \frac{1}{\ell^c}$ for all $\ell > \ell_0$.

A.1. IND-CCA2-secure Public-Key and Symmetric Encryption Schemes

After recalling the definition of a generic a public-key encryption scheme, we explain what it means for a public-key encryption scheme to be secure against adaptive chosen chiphertext attacks (IND-CCA2 security).

Definition A.1 (Public-Key Encryption schemes)**.** *A public-key encryption scheme consists of a triple of algorithms (Gen, Enc, Dec), where*

1. *Gen, the* key generation algorithm*, is a probabilistic algorithm that takes a security parameter ℓ and returns a pair (pk, sk) of matching public and secret keys.*
2. *Enc, the* encryption algorithm*, is a probabilistic algorithm that takes a public key pk and a message $x \in \{0,1\}^*$ to produce a ciphertext y.*
3. *Dec, the* decryption algorithm*, is a deterministic algorithm which takes a secret key sk and a ciphertext y to produce either a message $x \in \{0,1\}^*$ or a special symbol $\perp$ to indicate that the ciphertext was invalid.*

We require that for all (pk, sk) which can be output by $Gen(1^\ell)$, for all $x \in \{0,1\}^*$, and for all y that can be output by $Enc_{pk}(x)$, we have that $Dec_{sk}(y) = x$. We also require that Gen, Enc and Dec can be computed in polynomial time.

Definition A.2 (Encryption of vectors)**.** *Let (Gen, Enc, Dec) be a public-key encryption scheme. Let $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$ be vectors of entries in $\{0,1\}^*$. We write*

$$Enc_{pk}(x) = (Enc_{pk}(x_1), ..., Enc_{pk}(x_n))$$
$$Dec_{sk}(y) = (Dec_{sk}(y_1), ..., Dec_{sk}(y_n))$$

for every public key pk and every secret key sk.

We now present what it means for an encryption scheme to satisfy the "Indistinguishability under an Adaptive Chosen Chiphertext Attack" (IND-CCA2) property.

Definition A.3 (Challenger)**.** *Let* (Gen, Enc, Dec) *be a public-key encryption scheme. The* challenger C *is a probabilistic polynomial-time algorithm that takes a bit* b *as well as a key pair* (pk, sk) *and that serves two types of queries:*

1. *For a vector of messages* y*, the challenger returns the decryption of* y*, that is* $Dec_{sk}(y)$*.*

2. *For a pair of vectors of messages* (x_0, x_1) *where both vectors have the same size and all messages at the same position in the vectors have the same length, the challenger encrypts* x_b *under* pk *and returns the vector of ciphertexts, that is* $Enc_{pk}(x_b)$*.*

Definition A.4 (IND-CCA2-security [BDPR98])**.** *Let* (Gen, Enc, Dec) *be a public-key encryption scheme with security parameter* ℓ *and let* C *be the challenger. Then the encryption scheme* (Gen, Enc, Dec) *is* IND-CCA2-secure*, if for every polynomially bounded adversary* A *who never submits decryption queries for (parts of) a vector of messages* y *previously returned by a challenge query, we have that*

$$\begin{aligned} &\mathsf{Prob}\{(pk, sk) \leftarrow Gen(1^\ell); b' \leftarrow A^{C(1,pk,sk)}(1^\ell, pk); b' = 1\} \\ -&\mathsf{Prob}\{(pk, sk) \leftarrow Gen(1^\ell); b' \leftarrow A^{C(0,pk,sk)}(1^\ell, pk); b' = 0\} \end{aligned}$$

is a negligible function in ℓ*.*

A.2. EUF-CMA-secure Digital Signatures Schemes

After recalling the definition of a generic a digital signature scheme, we explain what it means for a digital signature scheme to be existentially unforgeable against adaptive chosen-message attacks (EUF-CMA security).

Definition A.5 (Digital Signature schemes)**.** *A digital signature scheme consists of a triple of algorithms* (Gen, Sig, Ver)*, where*

1. *Gen, the* key generation algorithm*, is a probabilistic algorithm that takes a security parameter* ℓ *and returns a pair* (sk, pk) *of matching secret and public keys.*

2. *Sig, the* signing algorithm*, is a (possibly) probabilistic algorithm that takes a private key* sk *and a message* $x \in \{0,1\}^*$ *to produce a signature* σ*.*

3. *Ver, the* verification algorithm*, is a deterministic algorithm which takes a public key* pk*, a message* $x \in \{0,1\}^*$ *and a signature* σ *to produce a boolean value.*

We require that for all (sk, pk) *which can be output by* $Gen(1^\ell)$*, for all* $x \in \{0,1\}^*$*, and for all* σ *that can be output by* $Sig_{sk}(x)$*, we have that* $Ver_{sk}(x, \sigma) = true$*. We also require that Gen, Sig and Ver can be computed in polynomial time.*

We now present what it means for a digital signature scheme to satisfy the "Existential Unforgeability under an Adaptive Chosen-Message Attack" (EUF-CMA) property.

Definition A.6 (EUF-CMA-security [GMR88]). *Let (Gen, Sig, Ver) be a signature scheme with security parameter ℓ. Then the signature scheme is* existentially unforgeable under adaptive chosen-message attacks (EUF-CMA-secure) *if for every probabilistic polynomial-time algorithm A who has access to a signing oracle and who never outputs tuples (x, σ) for which x has previously been signed by the oracle, we have that*

$$\mathsf{Prob}\{(sk, pk) \leftarrow Gen(1^\ell); (x, \sigma) \leftarrow A^{Sig_{sk}(\cdot)}(1^\ell, pk); Ver_{pk}(x, \sigma) = true\}$$

is negligible as a function in ℓ.

B. The Jinja+ and SyncJinja+ languages

At the basis of the formal results of the CVJ framework lies the language Jinja+ that extends Jinja [KN06] with:

– the primitive type `byte` with natural conversions from and to `int`;

– arrays;

– `abort` primitive;

– static fields (with the restriction that they can be initialized by literals only);

– static methods;

– access modifier for classes, fields, and methods (such as `private`, `protected`, and `public`);

– final classes (classes that cannot be extended);

– the `throws` clause of a method declaration (that declare which exceptions can be thrown by a method).

For the last three extensions—access modifiers, final classes, and `throws` clauses—we assume that they are provided by a compiler that, first, ensures that the policies expressed by access modifiers, the final modifier, and `throws` clauses are respected and then produces pure Jinja+ code (without access modifiers, the final modifier, and `throws` clauses). In the similar manner, we can deal with constructors: a program using constructors can be easily translated to one without constructors (where creation and initialisation of an object is split into two separate steps).

The remaining extensions are described below:

Primitive types. The Jinja language, as specified in [KN06], offers only boolean and integer primitive types. For our purpose, we find it useful to also include type `byte` with natural conversions from and to `int`. Also, the set of operators on primitive types is extended to include the standard Java operators (such as multiplication). This extensions can be done in very straightforward way and, thus, we skip its detailed description.

Arrays. We will consider only one-dimensional arrays (an extension to multi-dimensional arrays is then quite straightforward; moreover multi-dimensional arrays can be simulated by nested arrays). To extend the Jinja language with one-dimensional arrays, we adopt the approach of [NvO98].

First, we extend the set of types to include array types of the form τ`[]`, where τ is a type. Next, we extend the set of expressions by:

(i) creation of new array: `new` τ`[`e`]`, where e is an expression (that is supposed to evaluate to an integer denoting the size of the array) and τ is a type;

(ii) array access: e_1`[`e_2`]`;

(iii) array length access: e`.length`;

(iv) array assignment: e_1`[`e_2`] :=` e_3.

For this extension, following [NvO98], we redefine a *heap* to be a map from references to *objects*, where an *object* is either an *object instance*, as defined above, or an *array*. An *array* is a triple consisting of its component type, its length l, and a table mapping $\{0,\ldots,l-1\}$ to values.

Extending (small-step) semantic rules to deal with arrays is quite straightforward.

The `abort` primitive. Expression `abort`, when evaluated, causes the program to stop. (Technically this expression cannot be reduced and causes the program execution to get stuck.)

Static methods and fields. Fields and methods can be declared as static. However, as can be seen below, to keep the semantics of the language simple, we impose some restrictions on initializers of static fields.

A static method does not require an object to be invoked. The syntax of static method call is `C.f(`*args*`)`, where `C` is the name of a class that provides `f`.

Extending Jinja with with static methods is straightforward. The rule for static method invocation is very similar to the one for non-static method invocation: the difference is that the variable **`this`** is not added to the context (block) within which the method body is executed (a static method cannot reference non-static fields and methods).

We assume that static fields can be initialized only with literals (constants) of appropriate types. If there is no explicit initializer, then a static variable is initialized with the default value of its type. For example, while `static int x = 7` and `static int[] t` are valid declarations, the declaration `static A a = new A()` and `static int y = A.foo()` are not.

Dealing with more general static initializers is not difficult in principle, but it would require a precise—and quite complicated—model of the initialisation process, the complication we want to avoid.

Extending Jinja with static fields requires only a very little overhead: for a static field `f` declared in class `C` we introduce a global variable `C.f` (note that names of this form do not interfere with names of local variables and method parameters). These global variables are initialized before actual program (expression) is executed, as described in the definition of a run below.

Exceptions. A method declaration can contain a `throws` clause in which classes of exceptions that can be propagated by the method are listed. Such a clause can be omitted, in which case the above mentioned list is considered empty. When the meaning of `throws` clauses is considered, standard subtyping rules are applied (if class A is listed in such a clause, then the method can propagate exceptions of class A or any subclass of A).

As mentioned, we assume that the compiler (or a static verifier) statically checks whether the program complies with `throws` clauses.

Unlike in Java, however, we can assume without loss of generality that all exceptions must be declared in a `throws` clause if they are propagated by a method (in the Java terminology, we can say that all exceptions are checked). This will give us more control on the information which is passed between program components.

We consider the following hierarchy of standard (system) exceptions. In the root of this hierarchy we place (empty) class `Exception`. We require that only object of this class (and its subclasses) can be used as exceptions. Class `SystemException`, also empty, is a subclas of class `Exception`, and is a base class for the following system exceptions (exceptions which are not thrown explicitly, but may occur in result of some standard operations on expressions):

`ArrayStoreException` — trown to indicate an attempt to store an object of the wrong type into an array,

`IndexOutOfBoundsException` — thrown to indicate that an array has been indexed with an index being out of range,

`NegativeArraySizeException` — thrown to indicate an attempt to create an array with negative size,

`NullPointerException` — thrown if the `null` reference is used when an object is required,

`ClassCastException` — thrown to indicate an illegal cast.

We will assume that the above classes are predefined, and can be used in any program.

For completeness of the presentation, in this section we summarize all the rules of Jinja+. We start with rules of Jinja, following [KN06] (see this paper for the details on the used symbols). In particular, the syntactical convention used in these rules is that an application of a function f to an argument a is denoted by $f\ a$.

The rules assume a function *binop* that provides semantics for operations on atomic types. The exact definition of this function depends on the maximal size of integers that we consider (recall that we consider different variants of semantics for different size of integers given by $intsize(\eta)$ where η is the security parameter).

Rules of Jinja+. There are two points where the rules diverge from the ones of [KN06]. First, as we assume *unbounded memory*, we do not have rules which throw `OutOfMemoryError` (and we assume that (*new-Addr h*) is never *None*). Second, we added labels to rules. These labels allow us to count the number of steps performed within (by) a given class or subsystem.

A label D in a step

$$\langle e, s\rangle \xrightarrow{\mathscr{A}}_D \langle e', s'\rangle$$

means, informally, that the step was executed by the code of class D (the meaning of the symbol $\mathscr{A}$ on top of the arrow is explained in the next section). More precisely, the expression that was selected to be reduced by an elementary rule comes from a method of D. We use the label ε if the origin of the reduced expression is not known (because, at that point, the context of this expression is not known; typically this label is overwritten by a subexpression reduction rule for blocks, that is Rules (B.8)–(B.10)).

To define labeling of transitions, labels are also added to blocks that are obtained from the method call rules (B.24) and (B.61) (a block is labeled by the name of the class from which the body of the method comes). Then, the labels of transitions are, roughly speaking, inherited from the innermost block within which the reduction takes place.

Now, for the run of a program P with a subsystem S, we say that a step $\langle s_1, e_1\rangle \xrightarrow{\mathscr{A}}_D \langle s_2, e_2\rangle$ is performed by S and write $\langle s_1, e_1\rangle \xrightarrow{\mathscr{A}}_S \langle s_2, e_2\rangle$, if D is the name of a class defined in S.

Subexpression reduction rules (Figure B.0) describe the order in which subexpressions are evaluated. The relation $[\rightarrow]$ it the extension of $\rightarrow$ to expression list ($\cdot$ is the list constructor).

One particular type of expression is a *block expression* of the form $\{V : T;\ e\}_C$ or $\{V : T;\ V := \texttt{Val}\ v;\ e\}_C$, where V is a local variable (whose scope is this block) of type T and, in the second variant, with value `Val` v, e is an expression (e can access the local variable V), and C is a class name (denoting that the block originates from the code of the class C).

In a block $\{V : T;\ e\}_C$ we keep reducing e in a store where V is undefined (i.e, set to "None"), restoring the original binding of V after each step (Rule B.8). Once the store after the reduction step binds V to a value v, this binding is remembered by adding an assignment in front of the reduced expression, yielding $\{V : T;\ V := Val\ v;\ e\}_C$ (Rule B.9). The final rule (Rule B.10) reduces such block.

Expression reduction rules (Figure B.1) are applied when the subexpressions are sufficiently reduced. In rule (B.24) for method invocation, the required nested block structure is built with the help of the auxiliary function *blocks*:

$$\begin{aligned} blocks_C([],[],[],e) &= e \\ blocks_C(V \cdot Vs, T \cdot Ts, v \cdot vs, e) &= \\ &= \{V : T; V := v;\ blocks_C(Vs, Ts, vs, e)\}_C \end{aligned}$$

(where $\cdot$ is the list constructor and [] denotes the empty list).

Exceptional reduction and *exception propagation* rules (Figure B.2 and B.3) describe how exception are thrown and propagated.

Note that we do not have a rule reducing `abort`. That means that, if this expression is to be reduced, the execution gets stuck.

The rules given in Figure B.4 and B.5 are the additional rules of Jinja+ concerning static method invocation and arrays. Figure B.6 describes instead the extension of the (small-step) semantic rules for the `String` data type.

SyncJinja+, a multi-threaded and synchronized Java-like language. We extend Jinja+ in order to model runs of multi-threaded programs.

Following [Loc12], we extend the Jinja+ semantics with rules to reduce the constructs introduced in SyncJinja+, namely `start`, `sync`, and `insync` (Rules B.77-B.87, Figure B.7). These rules produce the thread actions $\mathscr{A} ::= \emptyset \mid Spawn(a) \mid Lock(a) \mid Unlock(a)$ which are then propagated to all the other subexpression reduction rules of Jinja+.

We then define the rules of the interleaving (multi-threaded) semantics for SyncJinja+ which models the run of a multi-threaded program P under a scheduler $\mathscr{S}$.

In order to model the multi-threaded execution we need to introduce a new kind of exception: `IllegalThreadState` is thrown to indicate an attempt to spawn a thread from an object of (subclass of) class *Thread* which was already used for spawning another thread (see Rule B.92).

B.1. Small-Step Semantics of Jinja, Jinja+, and SyncJinja+

B.1.1. Semantics Rules of Jinja

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \texttt{Cast}\ C\ e', s'\rangle} \tag{B.1}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle V := e, s\rangle \xrightarrow{\ell} \langle V := e', s'\rangle} \tag{B.2}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e.F\{D\}, s\rangle \xrightarrow{\ell} \langle e'.F\{D\}, s'\rangle} \tag{B.3}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e.F\{D\} := e_2, s\rangle \xrightarrow{\ell} \langle e'.F\{D\} := e_2, s'\rangle} \tag{B.4}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e, s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.F\{D\} := e', s'\rangle} \tag{B.5}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e \ll bop \gg e_2, s\rangle \xrightarrow{\ell} \langle e' \ll bop \gg e_2, s'\rangle} \tag{B.6}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg e, s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v_1 \ll bop \gg e', s'\rangle} \tag{B.7}$$

$$\frac{P \vdash \langle e, (h, l(V := None))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = None \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D, (h, l)\rangle \xrightarrow{g(\ell, D)} \langle \{V : T; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{B.8}$$

$$\frac{P \vdash \langle e, (h, l(V := None))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = v \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D, (h, l)\rangle \xrightarrow{g(\ell, D)} \langle \{V : T;\ V := \texttt{Val}\ v; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{B.9}$$

$$\frac{P \vdash \langle e, (h, l(V := v))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = v'}{P \vdash \langle \{V : T; V := \texttt{Val}\ v; e\}_D, (h, l)\rangle \xrightarrow{g(\ell, D)} \langle \{V : T;\ V := \texttt{Val}\ v'; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{B.10}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.M(es),s\rangle \xrightarrow{\ell} \langle e'.M(es),s'\rangle} \tag{B.11}$$

$$\frac{P \vdash \langle es,s\rangle \ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es),s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.M(es'),s'\rangle} \tag{B.12}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e;e_2,s\rangle \xrightarrow{\ell} \langle e';e_2,s'\rangle} \tag{B.13}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2,s\rangle \xrightarrow{\ell} \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2,s'\rangle} \tag{B.14}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \cdot es,s\rangle \ [\xrightarrow{\ell}]\ \langle e' \cdot es,s'\rangle} \qquad \frac{P \vdash \langle es,s\rangle \ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es,s\rangle \ [\xrightarrow{\ell}]\ \langle \texttt{Val}\ v \cdot es',s'\rangle} \tag{B.15}$$

Figure B.0.: Subexpression reduction rules for Jinja. We define $g(\ell,D) = D$, if $\ell = \varepsilon$; otherwise $g(\ell,D) = \ell$. The predicate *assigned V e* is defined as: *assigned V e* $\equiv \exists\ v\ e'.\ e =$ $\texttt{V} := \texttt{Val v};\ e'$ The relation $[\rightarrow]$ is the extension of $\rightarrow$ to expression lists.

$$\frac{\textit{new-Addr}\ h = a \quad P \vdash C\ \textit{has-fields}\ FDTs}{P \vdash \langle \texttt{new}\ C, (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \textit{addr}\ a, (h(a \mapsto (C,\ \textit{init-fields}\ FDTs)), l)\rangle} \tag{B.16}$$

$$\frac{\textit{hp}\ s\ a = (D, fs) \quad P \vdash D \preceq^{*} C}{P \vdash \langle \texttt{Cast}\ C\ (\textit{addr}\ a), s\rangle \xrightarrow{-} \langle \textit{addr}\ a, s\rangle} \tag{B.17}$$

$$P \vdash \langle \texttt{Cast}\ C\ \textit{null}, s\rangle \xrightarrow{-} \langle \textit{null}, s\rangle \tag{B.18}$$

$$\frac{\textit{lcl}\ s\ V = v}{P \vdash \langle \texttt{Var}\ V, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{B.19}$$

$$P \vdash \langle V := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h, l(V \mapsto v))\rangle \tag{B.20}$$

$$\frac{\textit{binop}\ (\textit{bop}, v_1, v_2) = v}{P \vdash \langle \texttt{Val}\ v_1 \ll \textit{bop} \gg \texttt{Val}\ v_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{B.21}$$

$$\frac{\textit{hp}\ s\ a = (C, fs) \quad fs(F,D) = v}{P \vdash \langle \textit{addr}\ a.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{B.22}$$

$$\frac{\textit{hp}\ a = (C, fs)}{P \vdash \langle \textit{addr}\ a.F\{D\} := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h(a \mapsto (C, fs((F,D) \mapsto v))), l)\rangle} \tag{B.23}$$

$$\frac{\textit{hp}\ s\ a = (C, fs) \quad P \vdash C\ \textit{sees}\ M:\ Ts \to T = (\textit{pns},\ \textit{body})\ \textit{in}\ D \quad |vs| = |\textit{pns}| \quad |Ts| = |\textit{pns}|}{P \vdash \langle \textit{addr}\ a.M(\textit{map}\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle \textit{blocks}_D(\textit{this} \cdot \textit{pns},\ \textit{Class}\ D \cdot Ts,\ \textit{addr}\ a \cdot vs,\ \textit{body}), s\rangle} \tag{B.24}$$

$$P \vdash \langle \{V : T;\ \texttt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val}\ u, s\rangle \tag{B.25}$$

$$P \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ \texttt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val}\ u, s\rangle \tag{B.26}$$

$$P \vdash \langle \texttt{Val}\ v;\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{B.27}$$

$$P \vdash \langle \texttt{if}(\textit{true})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_1, s\rangle \tag{B.28}$$

$$P \vdash \langle \texttt{if}(\textit{false})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{B.29}$$

$$P \vdash \langle \texttt{while}(b)\ c, s\rangle \xrightarrow{-} \langle \texttt{if}(b)\ (c;\ \texttt{while}(b)\ c)\ \texttt{else}\ \textit{unit}, s\rangle \tag{B.30}$$

Figure B.1.: Expression reduction rules for Jinja

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C(addr\ a), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW ClassCastException},\ s\rangle} \tag{B.31}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{B.32}$$

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{B.33}$$

$$P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{B.34}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{throw}\ e, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \texttt{throw}\ e', s'\rangle} \tag{B.35}$$

$$P \vdash \langle \texttt{throw}\ null, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NullPointerException}, s\rangle \tag{B.36}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle} \tag{B.37}$$

$$P \vdash \langle \texttt{try Val}\ v\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{Val}\ v, s\rangle \tag{B.38}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \{V : Class\ C;\ V := addr\ a;\ e_2\}, s\rangle} \tag{B.39}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{Throw}\ a, s\rangle} \tag{B.40}$$

Figure B.2.: Exceptional expression reduction for Jinja

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.41}$$

$$P \vdash \langle V := \texttt{throw}\ e, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.42}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.43}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.44}$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.45}$$

$$P \vdash \langle \texttt{throw}\ e\ \ll bop \gg\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.46}$$

$$P \vdash \langle \texttt{Val}\ v_1\ \ll bop \gg\ \texttt{throw}\ e, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.47}$$

$$P \vdash \langle \{V : T;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{B.48}$$

$$P \vdash \langle \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{B.49}$$

$$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.50}$$

$$P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.51}$$

$$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.52}$$

$$P \vdash \langle \texttt{if}(\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.53}$$

$$P \vdash \langle \texttt{throw}(\texttt{throw}\ e), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{throw}\ e, s\rangle \tag{B.54}$$

Figure B.3.: Exception propagation rules for Jinja.

B.1.2. Semantics Rules of the Jinja+ extension

$$\frac{P \vdash \langle es, s \rangle \; [\xrightarrow{\mathscr{A}}_{\ell}] \; \langle es', s' \rangle}{P \vdash \langle D.M(es), s \rangle \xrightarrow{\mathscr{A}}_{\ell} \langle D.M(es'), s' \rangle} \tag{B.55}$$

$$\frac{P \vdash \langle e, s \rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e', s' \rangle}{P \vdash \langle e[e_2], s \rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e'[e_2], s' \rangle} \tag{B.56}$$

$$\frac{P \vdash \langle e, s \rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e', s' \rangle}{P \vdash \langle (\mathtt{Val}\ v)[e], s \rangle \xrightarrow{\mathscr{A}}_{\ell} \langle (\mathtt{Val}\ v)[e'], s' \rangle} \tag{B.57}$$

$$P \vdash \langle D.M(map\ \mathtt{Val}\ vs\ @\ (\mathtt{throw}\, e\ \cdot\ es')), s \rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \mathtt{throw}\, e, s \rangle \tag{B.58}$$

$$P \vdash \langle (\mathtt{throw}\, e)[e'], s \rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \mathtt{throw}\, e, s \rangle \tag{B.59}$$

$$P \vdash \langle e'[\mathtt{throw}\, e], s \rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \mathtt{throw}\, e, s \rangle \tag{B.60}$$

Figure B.4.: Subexpression reduction and exception propagation rules for Jinja+.

$$\frac{P \vdash\ D\ \textit{has-static}\ M:\ Ts \to T = (pns,\ body) \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P \vdash \langle D.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle blocks_D(pns, Ts, vs,\ body), s\rangle} \tag{B.61}$$

$$\frac{n \geq 0,\ \textit{new-Addr}\ h = a}{P \vdash \langle \texttt{new}\ \ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle addr\ a, (h(a \mapsto initArr(\tau,n)), l)\rangle} \tag{B.62}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{B.63}$$

$$\frac{n < 0}{P \vdash \langle \texttt{new}\ \ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW NegativeArraySizeException},\ (h,l)\rangle} \tag{B.64}$$

$$\frac{h\ a = (\tau,m,t),\ 0 \leq n < m,\ h\ t(n) = v}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle Val\ v,\ (h,l)\rangle} \tag{B.65}$$

$$\frac{h\ a = (\tau,m,t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{B.66}$$

$$\frac{h\ a = (\tau,m,t),}{P \vdash \langle (addr\ a)\texttt{.lenght},\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle intg\ m,\ (h,l)\rangle} \tag{B.67}$$

$$\frac{h\ a = (\tau,m,t),\ 0 \leq n < m,\ isOfType(v,\tau),\ t' = arrayUpdate(t,n,v)}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle unit,\ (h(a \mapsto (\tau,m,t')), l)\rangle} \tag{B.68}$$

$$\frac{h\ a = (\tau,m,t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{B.69}$$

$$\frac{h\ a = (\tau,m,t),\ 0 \leq n < m,\ \neg isOfType(v,\tau),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW ArrayStoreException},\ (h,l)\rangle} \tag{B.70}$$

Figure B.5.: (Exceptional) expression reduction rules for Jinja+, where: Function $initArr(\tau,n)$ returns an array of length n with elements initialized to the default value of type τ. Expression $P \vdash D\ \textit{has-static}\ M : Ts \to T = (pbs, body)$ means that in program P, class D contains declaration of static method M with argument types Ts, return type T, formal arguments pbs, and the body $body$.

B.1.3. Semantics Rules for the data type `String`

$$\frac{\textit{new-Addr}\ h = a}{P \vdash \langle \textit{litS}\ str,\ \langle h, l\rangle\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \textit{addr}\ a, \langle h(a \mapsto \textit{initString}(str)), l\rangle\rangle} \tag{B.71}$$

$$\frac{h(a_1) = (ch_1, m_1),\ h(a_2) = (ch_2, m_2),\ \textit{new-Addr}\ h = a_3,\ ch_3 = \textit{concat}(ch_1, ch_2)}{P \vdash \langle(\textit{addr}\ a_1) + (\textit{addr}\ a_2), \langle h, l\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle(\textit{addr}\ a_3), \langle h(a_3 \mapsto (ch_3, m_1 + m_2)), l\rangle\rangle\rangle} \tag{B.72}$$

$$\frac{h(a) = (ch, m)}{P \vdash \langle(\textit{addr}\ a)\texttt{.lenght()},\ \langle h, l\rangle\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \textit{intg}\ m, \langle h, l\rangle\rangle} \tag{B.73}$$

$$\frac{h(a) = (ch, m),\ 0 \leq n < m,\ ch[n] = c}{P \vdash \langle(\textit{addr}\ a).\textit{charAt}(\textit{intg}\ n), \langle h, l\rangle\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \textit{char}\ c, \langle h, l\rangle\rangle} \tag{B.74}$$

$$\frac{h(a) = (ch, m),\ \neg(0 \leq n < m)}{P \vdash \langle(\textit{addr}\ a).\textit{charAt}(\textit{intg}\ n), \langle h, l\rangle\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \texttt{THROW IndexOutOfBoundsException}, \langle h, l\rangle\rangle} \tag{B.75}$$

$$\frac{h(a) = (ch, m),\ \textit{new-Addr}\ h = a'}{P \vdash \langle(\textit{addr}\ a).\textit{getBytes}(), \langle h, l\rangle\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \textit{addr}\ a', \langle h(a \mapsto \textit{encodeToByte}(ch, m), l\rangle\rangle} \tag{B.76}$$

Figure B.6.: Expression reduction rules for the `String` data type, where: Function $\textit{initString}(str)$ returns a pair of an array ch containing the charachers of the string literal str and its length m. Function $\textit{concat}(ch_1, ch_2)$ creates a new array of characters where the latter array ch_2 is concatenated to the former one ch_1. Function $\textit{encodeToByte}(ch, m)$ converts the string (ch, m) in a new array of bytes.

B.1.4. Semantics Rules of the SyncJinja+ extension

$$\frac{P \vdash \langle e,(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e',(h',l')\rangle}{P \vdash \langle \mathtt{start}(e),(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \mathtt{start}(e'),(h',l')\rangle} \tag{B.77}$$

$$\frac{P \vdash \langle e_1,(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e_1',(h',l')\rangle}{P \vdash \langle \mathtt{sync}(e_1)\{e_2\},(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \mathtt{sync}(e_1')\{e_2\},(h',l')\rangle} \tag{B.78}$$

$$\frac{P \vdash \langle e,(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle e',(h',l')\rangle}{P \vdash \langle \mathtt{insync}(a)\{e\},(h,l)\rangle \xrightarrow{\mathscr{A}}_{\ell} \langle \mathtt{insync}(a)\{e'\},(h',l')\rangle} \tag{B.79}$$

$$P \vdash \langle \mathtt{start}(\mathit{addr}\ a),(h,l)\rangle \xrightarrow{\mathit{Spawn}(a)}_{\varepsilon} \langle \mathit{unit},(h,l)\rangle \tag{B.80}$$

$$P \vdash \langle \mathtt{sync}(\mathit{addr}\ a)\{e\},(h,l)\rangle \xrightarrow{\mathit{Lock}(a)}_{\varepsilon} \langle \mathtt{insync}(a)\{e\},(h,l)\rangle \tag{B.81}$$

$$P \vdash \langle \mathtt{insync}(a)\{\mathtt{Val}\ v\},(h,l)\rangle \xrightarrow{\mathit{Unlock}(a)}_{\varepsilon} \langle \mathtt{Val}\ v,(h,l)\rangle \tag{B.82}$$

$$P \vdash \langle \mathtt{start}(\mathit{null}),(h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \mathtt{THROW\ NullPointerException},(h,l)\rangle \tag{B.83}$$

$$P \vdash \langle \mathtt{sync}(\mathit{null})\{e\},(h,l)\rangle \xrightarrow{\emptyset}_{\varepsilon} \langle \mathtt{THROW\ NullPointerException},(l,h)\rangle \tag{B.84}$$

$$P \vdash \langle \mathtt{start}(\mathtt{Throw}\ a),(h,l)\rangle \xrightarrow{\emptyset}_{\ell} \langle \mathtt{Throw}\ a,(l,h)\rangle \tag{B.85}$$

$$P \vdash \langle \mathtt{sync}(\mathtt{Throw}\ a)\{e\},(h,l)\rangle \xrightarrow{\emptyset}_{\ell} \langle \mathtt{Throw}\ a,(h,l)\rangle \tag{B.86}$$

$$P \vdash \langle \mathtt{insync}(a)\{\mathtt{Throw}\ a_1\},(h,l)\rangle \xrightarrow{\mathit{Unlock}(a)}_{\varepsilon} \langle \mathtt{Throw}\ a_1,(h,l)\rangle \tag{B.87}$$

Figure B.7.: Rules for the extension to SyncJinja+.

$$\frac{l_{\mathscr{S}}(\texttt{nextThread}) = null \quad P \vdash \langle e, (h, l(\texttt{actThreads} := \mathsf{Act}(\langle \Pi, h, lock \rangle))) \rangle_{\mathscr{S}} \xrightarrow{\emptyset}_{\ell} \langle e', (h', l') \rangle_{\mathscr{S}}}{P \vdash \langle \Pi, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \Longrightarrow_{\ell} \langle \Pi, h, lock \rangle_{\langle e', (l', h') \rangle_{\mathscr{S}}}} \tag{B.88}$$

$$\frac{l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e, (h, l) \rangle \xrightarrow{\emptyset}_{\ell} \langle e', (h', l') \rangle \quad \ell \notin \mathbb{A}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e, l \rangle\}, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \stackrel{tID_k}{\Longrightarrow}_{\ell} \langle \Pi \cup \{tID_k \mapsto \langle e', l' \rangle\}, h', lock \rangle_{\langle e, (l(\texttt{nextThread} := null), h) \rangle_{\mathscr{S}}}} \tag{B.89}$$

$$\frac{l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e, (h, l) \rangle \xrightarrow{\emptyset}_{\ell} \langle e', (h', l') \rangle \quad \ell \in \mathbb{A}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e, l \rangle\}, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \stackrel{tID_k}{\Longrightarrow}_{\ell} \langle \Pi \cup \{tID_k \mapsto \langle e', l' \rangle\}, h', lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}}} \tag{B.90}$$

$$\frac{\begin{array}{c} l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e, (h, l) \rangle \xrightarrow{Spawn(a)}_{\varepsilon} \langle e', (h', l') \rangle \\ h(a) = (C, fs) \quad P \vdash C \preceq^* Thread \quad tID_j \notin dom(\Pi) \cup \{tID_k\} \quad l'' := static(l) \end{array}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e, l \rangle\}, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \stackrel{tID_k}{\Longrightarrow}_{\varepsilon} \langle \Pi \cup \{tID_k \mapsto \langle e', l' \rangle\} \cup \{tID_j \mapsto \langle a.run(), l'' \rangle\}, h', lock \rangle_{\langle e, (l(\texttt{nextThread} := null), h) \rangle_{\mathscr{S}}}} \tag{B.91}$$

$$\frac{\begin{array}{c} l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e[\![\texttt{start}(a); e_0]\!], (h, l) \rangle \xrightarrow{Spawn(a)}_{\varepsilon} \langle e', (h', l') \rangle \\ h(a) = (C, fs) \quad P \vdash C \preceq^* Thread \quad tID_j \in dom(\Pi) \cup \{tID_k\} \end{array}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e[\![\texttt{start}(a); e_0]\!], l \rangle\}, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \stackrel{tID_k}{\Longrightarrow}_{\varepsilon} \langle \Pi \cup \{tID_k \mapsto \langle e[\![\texttt{THROW IllegalThreadState}; e_0]\!], l \rangle\}\}, h, lock \rangle_{\langle e, (l(\texttt{nextThread} := null), h) \rangle_{\mathscr{S}}}} \tag{B.92}$$

$$\frac{\begin{array}{l} l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e, (h, l) \rangle \xrightarrow{Lock(a)}_{\varepsilon} \langle e', (h', l') \rangle \\ a \notin dom(lock) \vee fst(lock(a)) = tID_k \quad lock' := \begin{cases} lock(a \mapsto (tID_k, 0)) & if\ a \notin dom(lock) \\ lock(a \mapsto (tID_k, snd(lock(a)) + 1) & else \end{cases} \end{array}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e, l \rangle\}, h, lock \rangle_{\langle e, (l,h) \rangle_{\mathscr{S}}} \stackrel{tID_k}{\Longrightarrow}_{\varepsilon} \langle \Pi \cup \{tID_k \mapsto \langle e', l' \rangle\}, h', lock' \rangle_{\langle e, (l(\texttt{nextThread} := null), h) \rangle_{\mathscr{S}}}} \tag{B.93}$$

$$\frac{l_{\mathscr{S}}(\texttt{nextThread}) = tID_k \quad P \vdash \langle e,(h,l)\rangle \xrightarrow{Unlock(a)}_{\varepsilon} \langle e',(h',l')\rangle \qquad lock' := \begin{cases} lock \setminus \{a \mapsto (tID_k, 0)\} & if\ snd(lock(a)) = 0 \\ lock(a \mapsto (tID_k, snd(lock(a)) - 1) & else \end{cases}}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle e,l\rangle\}, h, lock\rangle_{\langle e,(l,h)\rangle_{\mathscr{S}}} \overset{tID_k}{\Longrightarrow}_{\varepsilon} \langle \Pi \cup \{tID_k \mapsto \langle e',l'\rangle\}, h', lock'\rangle_{\langle e,(l(\texttt{nextThread}:=null),h)\rangle_{\mathscr{S}}}} \tag{B.94}$$

$$\frac{l_{\mathscr{S}}(\texttt{nextThread}) = tID_k}{P \vdash \langle \Pi \cup \{tID_k \mapsto \langle \texttt{abort}, l\rangle\}, h, lock\rangle_{\langle e,(l,h)\rangle_{\mathscr{S}}} \overset{tID_k}{\Longrightarrow}_{\varepsilon} \langle \Pi \cup \{tID_k \mapsto \langle unit, l\rangle\}, h, lock\rangle_{\langle unit,(l,h)\rangle_{\mathscr{S}}}} \tag{B.95}$$

Figure B.7.: Expression reduction rules for the multi-threaded semantics of SyncJinja+, where: Π is the current thread pool (with abuse of notation we also refer to Π as the thread pool minus the thread whose identifier is the bit string tID_k), while $\mathscr{A} ::= \emptyset \mid Spawn(a) \mid Lock(a) \mid Unlock(a)$ are the possible thread actions which the single-threaded semantics has to transmit to the multi-threaded one, Rules B.88-B.95. The only objects through which a new thread can be spawned are either objects of class $Thread$ or its subclasses: We assume this policy being statically enforced at compile time. Moreover, we assume that each object of class $Thread$ has an unique bit string (for instance, the object's hash code value) used as thread identifier. Expression $e[\![e_0]\!]$ makes explicit that e_0 is the expression in the innermost block of e, i.e., the (sub)expression of e which is going to be reduced in the next step. Functions $fst(a,b)$ and $snd(a,b)$ return the first and the second element of a pair, respectively. Function $static(l)$ returns the static fields in l, i.e., the subset of the map l containing those elements whose variable names are of type $D.f$, for some class D.

C. The Environment/Adversary

In all the ideal functionalities and in all the case studies for which we established noninterference properties, we make use of the class `Environment` which plays the role of the *active adversary* in the universal composability model: Besides controlling the network, the adversary also subsumes the not necessarily trusted external libraries (e.g., the libraries implementing the cryptographic primitives) and, depending on the scenario, other dishonest parties such as dishonest clients, voters, or servers. Therefore, in the analysis of our case studies what we identify as the environment/adversary is not only the class `Environment`, but also all the external libraries and entities interacting with the code under verification. The methods of this classes become then wrappers passing their arguments to the class `Environment` and returning the data returned, in turn, from the `Environment`. An example of how these methods are implemented is already given in Figures 2.3 and 2.4 of Section 2.6.1 and in Figure 2.6 of Section 2.6.3.

In what follow, we present the interface of the eight methods through which the communication with the `Environment` occurs.

```
1  // data passed to the Environment
2  public static void untrustedOutput(int x);
3  public static void untrustedOutputString(String s);
4  public static void untrustedOutputLong(long y);
5  public static void untrustedOutputMessage(byte[] t);
6
7  // data returned from the Environment
8  public static int untrustedInput();
9  public static String untrustedOutputString();
10 public static long untrustedInputLong();
11 public static byte[] untrustedInputMessage();
```

The methods `untrustedOutput`, `untrustedInput`, `untrustedOutputString`, and `untrutedInputString` are specified in Figure 2.2 of Section 2.6.1 and in Figure 2.5 of Section 2.6.3, since their implementation is part the proof technique for proving *I*-noninterference and it is essential to prove its results, namely Theorems 2.3, 2.4, and 2.5. Instead, the methods `untrustedOutputLong`, `untrustedInputLong`, `untrustedInputMessage`, and `untrustedOutputMessage` are not specified anywhere else, since they straightforwardly relying on the methods `untrustedInput` and `untrustedOutput`. In particular, `untrustedInputMessage` and `untrustedOutputMessage` are used for the communication through the *byte array* data type which is used in all the cryptographic operations and in all the exchange of data through the network. For the sake of completeness, we provide here their implementation.

```
1 public static byte[] untrustedInputMessage() {
2  int len = untrustedInput();
3  if (len<0) return null;
4   byte[] returnval = new byte[len];
5   for (int i = 0; i < len; i++)
```

```
6      returnval[i] = (byte) untrustedInput();
7     return returnval;
8  }
9
10 public static void untrustedOutputMessage(byte[] t) {
11  untrustedOutput(t.length);
12  for (int i = 0; i < t.length; i++) {
13   untrustedOutput(t[i]);
14  }
15 }
```

```
1  public static long untrustedInputLong(){
2   int x1=untrustedInput(); // first 32 bits
3   int x2=untrustedInput(); // last 32 bits
4   return (long) x2 << 32 | x1 & 0xFFFFFFFFL;
5  }
6
7  public static void untrustedOutputLong(long y){
8   untrustedOutput((int) y);       // first 32 bits
9   untrustedOutput((int) y >> 32); // last 32 bits
10 }
```

D. Real and Ideal Cryptographic Functionalities

We present here the implementation in Java of the cryptographic functionalities discussed in Chapter 4. In what follows, we note that we omit the declarations of:

– the method `getZeroMessage` that for a integer n returns a message of length n consisting of zeros;

– the method `copyOf` that returns a copy of the given message;

– the class `MessagePairList` that stores pairs of messages an offers the following methods: `add` (to add a message pair), `contains` (to check, for a message m, whether there is a pair (m',m) in the list), and `lookup` (which, for a message m, returns m' as above, if it exists).

– the class `CryptoLib` which, for the real functionalities, provides the actual implementation of the cryptographic operations, while for the ideal functionalities, is implemented as part of the environment (see Appendix C).

The implementation of these methods and classes as well as of all the cryptographic functionalities listed below can be found in [TSK13].

Finally, we notice that the code exhibited here falls into the fragment of Java captured by the extensions of the Jinja language presented in Appendix B.

D.1. The Public Key Infrastructure

Ideal Functionality.

```
1  public class IdealPKI {
2
3   static void register(int id, byte[] domain, byte[] key) throws PKIError, NetworkError {
4    if (Environment.untrustedInput()==0) throw new NetworkError();
5    if (registered(id, domain)) throw new PKIError();
6    entries.add(id, domain, key);
7   }
8   static byte[] getKey(int id, byte[] domain) throws PKIError, NetworkError {
9     if (Environment.untrustedInput()==0) throw new NetworkError();
10    byte[] key = entries.getKey(id, domain);
11    if (key == null) throw new PKIError();
12    return key;
13  }
14  static private boolean registered(int id, byte[] domain) {
15    return entries.getKey(id, domain) != null;
16  }
17  /// IMPLEMENTATION ///
18  private static class Entry {
19    final int id;
```

```
20      byte[] domain;
21      byte[] key;
22
23      Entry(int id, byte[] domain, byte[] key) {
24        this.id = id;
25        this.domain = domain;
26        this.key = key;
27      }
28    }
29    private static class EntryList {
30      private static class Node {
31        Entry entry;
32        Node next;
33        Node(Entry entry, Node next) {
34          this.entry = entry;
35          this.next = next;
36        }
37      }
38
39      private Node first = null;
40
41      void add(int id, byte[] domain, byte[] key) {
42       first = new Node(new Entry(id,domain,key), first);
43      }
44      byte[] getKey(int id, byte[] domain) {
45       for(Node node=first;  node!=null;  node = node.next)
46        if (node.entry.id==id &&  MessageTools.equal(node.entry.domain, domain))
47         return node.entry.key;
48        return null;
49      }
50    }
51    static private EntryList entries = new EntryList();
52  }
```

Real Functionality.

```
1   public class RealPKI {
2
3     public static class Error extends Exception {}
4
5     private static PKIServer pki = null;
6
7     public static void useRemoteMode() {
8      pki = new PKIServerRemote();
9      System.out.println("Working␣in␣remote␣mode");
10    }
11
12    public static void useLocalMode() {
13     pki = new PKIServerLocal();
14     System.out.println("Working␣in␣local␣mode");
15    }
16
17    public static void register(int id, byte[] domain, byte[] pubKey)
18                                          throws Error, NetworkError {
```

```
19     if (pki==null){
20       System.err.println("ERROR:␣PKI␣not␣initialized!\n"
21       + "Call␣‘RealPKI.useRemoteMode’␣or␣‘RealPKI.useLocalMode’␣first.");
22     }
23     pki.register(id, domain, pubKey);
24   }
25
26   public static byte[] getKey(int id, byte[] domain)
27                                        throws Error, NetworkError {
28     if (pki==null)
29      System.err.println("ERROR:␣PKI␣not␣initialized!");
30     return pki.getKey(id, domain);
31   }
32 }
```

For the class `RealPKI` we omit the declaration of the Java-interface `PKIServer` as well as of the classes `PKIServerRemote`, which allows one to register and retrieve a public key over the network, and `PKIServerLocal`, which instead maintains the `(id, publicKey)` pairs on a local database. The declaration of these classes and interface can be found in [TSK13].

D.2. PKIEnc: Public Key Encryption with a Public Key Infrastructure

Ideal Functionality.

```
1  public class Encryptor {
2    protected byte[] publicKey;
3    public Encryptor(byte[] publicKey) {
4      this.publicKey = publicKey;
5    }
6    public byte[] encrypt(byte[] message) {
7      return copyOf(CryptoLib.pke_encrypt(copyOf(message),
8                                          copyOf(publicKey)));
9    }
10   public byte[] getPublicKey() {
11     return copyOf(publicKey);
12   }
13   protected Encryptor copy() {
14     return new Encryptor(publicKey);
15   }
16 }
```

—

```
1  public final class UncorruptedEncryptor extends Encryptor {
2    private Decryptor.EncryptionLog log;
3
4    UncorruptedEncryptor(byte[] publicKey, Decryptor.EncryptionLog log) {
5      super(publicKey);
6      this.log = log;
7    }
8    public byte[] encrypt(byte[] message) {
9      byte[] randomCipher = null;
```

```
10      while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
11        randomCipher = copyOf(CryptoLib.pke_encrypt(getZeroMessage(message.length),
12                              copyOf(publicKey)));
13      }
14      log.add(copyOf(message), randomCipher);
15      return copyOf(randomCipher);
16    }
17    protected Encryptor copy() {
18      return new UncorruptedEncryptor(publicKey, log);
19    }
20  }
```

—

```
21  public class Decryptor {
22    private byte[] publicKey;
23    private byte[] privateKey;
24    private EncryptionLog log;
25
26    public Decryptor() {
27      KeyPair keypair = CryptoLib.pke_generateKeyPair();
28      this.privateKey = copyOf(keypair.privateKey);
29      this.publicKey = copyOf(keypair.publicKey);
30      this.log = new EncryptionLog();
31    }
32    public byte[] decrypt(byte[] message) {
33      byte[] messageCopy = copyOf(message);
34      if (!log.containsCiphertext(messageCopy)) {
35        return copyOf(CryptoLib.pke_decrypt(copyOf(privateKey), messageCopy));
36      } else {
37        return copyOf(log.lookup(messageCopy));
38      }
39    }
40    public Encryptor getEncryptor() {
41      return new UncorruptedEncryptor(publicKey, log);
42    }
43  }
```

—

```
44  public class RegisterEnc {
45    public static void registerEncryptor(Encryptor encryptor, int id,
46                       byte[] pki_domain) throws PKIError, NetworkError
47    {
48      if( RegisterEncSim.register(id, pki_domain, encryptor.getPublicKey()) )
49        throw new NetworkError();
50      if( registeredAgents.fetch(id, pki_domain) != null )
51        throw new PKIError();
52      registeredAgents.add(id, pki_domain, encryptor);
53    }
54    public static Encryptor getEncryptor(int id, byte[] pki_domain)
55                              throws PKIError, NetworkError
56    {
57      if( RegisterEncSim.getEncryptor(id, pki_domain) )
58          throw new NetworkError();
```

```
59        Encryptor enc = registeredAgents.fetch(id, pki_domain);
60        if (enc == null)
61          throw new PKIError();
62        return enc.copy();
63      }
64    
65      public static class PKIError extends Exception { }
66    
67      /// IMPLEMENTATION
68      private static class RegisteredAgents {
69        private static class EncryptorList {
70          final int id;
71          byte[] domain;
72          Encryptor encryptor;
73          EncryptorList next;
74          EncryptorList(int id, byte[] domain, Encryptor encryptor,
75                        EncryptorList next) {
76            this.id = id;
77            this.domain = domain;
78            this.encryptor= encryptor;
79            this.next = next;
80          }
81        }
82        private EncryptorList first = null;
83    
84        public void add(int id, byte[] domain, Encryptor encr) {
85          first = new EncryptorList(id, domain, encr, first);
86        }
87    
88        Encryptor fetch(int ID, byte[] domain) {
89          for( EncryptorList node = first;  node != null; node = node.next ) {
90            if( ID == node.id && MessageTools.equal(domain, node.domain) )
91              return node.encryptor;
92          }
93          return null;
94        }
95      }
96    
97      private static RegisteredAgents registeredAgents = new RegisteredAgents();
98    }
```

Real Functionality.

```
1     public class Encryptor {
2       private byte[] publicKey;
3     
4       public Encryptor(byte[] publicKey) {
5         this.publicKey = publicKey;
6       }
7       public byte[] encrypt(byte[] message) {
8         return copyOf(CryptoLib.pke_encrypt(copyOf(message),
9                                             copyOf(publicKey)));
10      }
11      public byte[] getPublicKey() {
```

```
        return copyOf(publicKey);
    }
}
```

```
public class Decryptor {
    byte[] publicKey;
    byte[] privateKey;

    public Decryptor() {
        KeyPair keypair = CryptoLib.pke_generateKeyPair();
        this.privateKey = copyOf(keypair.privateKey);
        this.publicKey = copyOf(keypair.publicKey);
    }
    Decryptor(byte[] pubk, byte[] prvkey) {
        this.publicKey = pubk;
        this.privateKey = prvkey;
    }
    public byte[] decrypt(byte[] message) {
        return copyOf(CryptoLib.pke_decrypt(copyOf(message),
                                            copyOf(privateKey)));
    }
    public Encryptor getEncryptor() {
        return new Encryptor(copyOf(publicKey));
    }
}
```

```
public class RegisterEnc {
    public static void registerEncryptor(Encryptor encryptor, int id,
                         byte[] pki_domain) throws PKIError, NetworkError
    {
        try {
            PKI.register(id, pki_domain, encryptor.getPublicKey());
        } catch (PKI.Error e) {
            throw new PKIError();
        }
    }
    public static Encryptor getEncryptor(int id, byte[] pki_domain)
                              throws PKIError, NetworkError
    {
        try {
            byte[] key = PKI.getKey(id, pki_domain);
            return new Encryptor(key);
        } catch (PKI.Error e) {
            throw new PKIError();
        }
    }

    public static class PKIError extends Exception { }
}
```

D.2.1. Ideal Functionality for Public Key Encryption without Corruption

```
public final class Encryptor {
  private Decryptor.EncryptionLog log;

  Encryptor(byte[] publicKey, Decryptor.EncryptionLog log) {
    super(publicKey);
    this.log = log;
  }

  public byte[] encrypt(byte[] message) {
    byte[] randomCipher = null;
    while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
      randomCipher = copyOf(CryptoLib.pke_encrypt(getZeroMessage(message.length),
                                  copyOf(publicKey)));
    }
    log.add(copyOf(message), randomCipher);
    return copyOf(randomCipher);
  }

  protected Encryptor copy() {
    return new Encryptor(publicKey, log);
  }
}
```

We notice that this code has already been presented in [KTG12a]. The class `Decryptor` is as in the functionality with corruption (see Appendix D.2).

D.3. PKISig: Digital Signature with a Public Key Infrastructure

Ideal Functionality.

```
public class Verifier {
  protected byte[] verifKey;

  public Verifier(byte[] verifKey) {
    this.verifKey = verifKey;
  }
  public boolean verify(byte[] signature, byte[] message) {
    return CryptoLib.verify(message, signature, verifKey);
  }
  public byte[] getVerifKey() {
    return copyOf(verifKey);
  }
  protected Verifier copy() {
    return new Verifier(verifKey);
  }
}
```

```
public final class UncorruptedVerifier extends Verifier {
  private Signer.Log log;

  UncorruptedVerifier(byte[] verifKey, Signer.Log log) {
```

```
21      super(verifKey);
22      this.log = log;
23    }
24    public boolean verify(byte[] signature, byte[] message) {
25      return CryptoLib.verify(message, signature, verifKey)
26              && log.contains(message);
27    }
28    protected Verifier copy() {
29      return new UncorruptedVerifier(verifKey, log);
30    }
31  }
```

```
32  final public class Signer {
33    private byte[] verifKey;
34    private byte[] signKey;
35    private Log log;
36
37    public Signer() {
38      KeyPair keypair = CryptoLib.generateSignatureKeyPair();
39      this.signKey = copyOf(keypair.privateKey);
40      this.verifKey = copyOf(keypair.publicKey);
41      this.log = new Log();
42    }
43    public byte[] sign(byte[] message) {
44      byte[] signature = CryptoLib.sign(copyOf(message), copyOf(signKey));
45      if (signature == null) return null;
46      if( !CryptoLib.verify(copyOf(message), copyOf(signature), copyOf(verifKey)) )
47        return null;
48      log.add(copyOf(message));
49      return copyOf(copyOf(signature));
50    }
51    public Verifier getVerifier() {
52      return new UncorruptedVerifier(verifKey, log);
53    }
54  }
```

```
55  public class RegisterSig {
56
57    public static void registerVerifier(Verifier verifier, int id,
58                          byte[] pki_domain) throws PKIError, NetworkError
59    {
60      if( RegisterSigSim.register(id, pki_domain, verifier.getVerifKey()) )
61        throw new NetworkError();
62      if( registeredAgents.fetch(id, pki_domain) != null )
63        throw new PKIError();
64      registeredAgents.add(id, pki_domain, verifier);
65    }
66    public static Verifier getVerifier(int id, byte[] pki_domain)
67                              throws PKIError, NetworkError
68    {
69      if( RegisterSigSim.getVerifier(id, pki_domain) ) throw new NetworkError();
70      Verifier verif = registeredAgents.fetch(id, pki_domain);
71      if (verif == null)
72        throw new PKIError();
```

```
73      return verif.copy();
74    }
75
76    public static class PKIError extends Exception { }
77
78    /// IMPLEMENTATION ///
79    private static class RegisteredAgents {
80      private static class VerifierList {
81        final int id;
82        byte[] domain;
83        Verifier verifier;
84        VerifierList  next;
85        VerifierList(int id, byte[] domain,  Verifier verifier,
86                     VerifierList next)
87        {
88          this.id = id;
89          this.domain = domain;
90          this.verifier = verifier;
91          this.next = next;
92        }
93      }
94
95      private VerifierList first = null;
96
97      public void add(int id, byte[] domain, Verifier verif) {
98        first = new VerifierList(id, domain, verif, first);
99      }
100     Verifier fetch(int ID, byte[] domain) {
101       for( VerifierList node = first;  node != null; node = node.next ) {
102         if( ID == node.id && MessageTools.equal(domain, node.domain) )
103           return node.verifier;
104       }
105       return null;
106     }
107   }
108
109   private static RegisteredAgents registeredAgents = new RegisteredAgents();
110 }
```

Real Functionality.

```
1  public class Verifier {
2    private byte[] verifKey;
3
4    public Verifier(byte[] verifKey) {
5      this.verifKey = verifKey;
6    }
7    public boolean verify(byte[] signature, byte[] message) {
8      return CryptoLib.verify(copyOf(message), copyOf(signature), copyOf(verifKey));
9    }
10   public byte[] getVerifKey() {
11     return copyOf(verifKey);
12   }
13 }
```

—

```
14 public class Signer {
15   byte[] verifKey;
16   byte[] signKey;
17 
18   public Signer() {
19     KeyPair keypair = CryptoLib.generateSignatureKeyPair();
20     this.signKey = copyOf(keypair.privateKey);
21     this.verifKey = copyOf(keypair.publicKey);
22   }
23   Signer(byte[] verifKey, byte[] signKey ) {
24     this.verifKey = verifKey;
25     this.signKey = signKey;
26   }
27   public byte[] sign(byte[] message) {
28     byte[] signature = CryptoLib.sign(copyOf(message), copyOf(signKey));
29     return copyOf(signature);
30   }
31   public Verifier getVerifier() {
32     return new Verifier(verifKey);
33   }
34 }
```

—

```
35 public class RegisterSig {
36   public static void registerVerifier(Verifier verifier, int id,
37                      byte[] pki_domain) throws PKIError, NetworkError
38   {
39     try {
40       PKI.register(id, pki_domain, verifier.getVerifKey());
41     } catch (PKI.Error e) {
42       throw new PKIError();
43     }
44   }
45   public static Verifier getVerifier(int id, byte[] pki_domain)
46                                         throws PKIError, NetworkError
47   {
48     try {
49       byte[] key = PKI.getKey(id, pki_domain);
50       return new Verifier(key);
51     } catch (PKI.Error e) {
52       throw new PKIError();
53     }
54   }
55 
56   public static class PKIError extends Exception { }
57 }
```

D.3.1. Ideal Functionality for Digital Signatures without Corruption

```
1 public final class Verifier {
```

```
2     private Signer.Log log;
3
4     Verifier(byte[] verifKey, Signer.Log log) {
5       super(verifKey);
6       this.log = log;
7     }
8     public boolean verify(byte[] signature, byte[] message) {
9       // verify both that the signature is correct
10      // and that the message has been logged as signed
11      return CryptoLib.verify(message, signature, verifKey)
12              && log.contains(message);
13    }
14    protected Verifier copy() {
15      return new Verifier(verifKey, log);
16    }
17  }
```

The class `Signer` is as in the functionality with corruption (Appendix D.3).

D.4. Private Symmetric Encryption

Ideal Functionality.

```
1   public class SymEnc {
2     private byte[] key;
3     private EncryptionLog log;
4
5     public SymEnc() {
6       key = CryptoLib.symkey_generateKey();
7     }
8     public byte[] encrypt(byte[] plaintext) {
9       byte[] randomCipher = null;
10      while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
11        randomCipher = copyOf(CryptoLib.symkey_encrypt(copyOf(key),
12                              getZeroMessage(plaintext.length)));
13      }
14      log.add(copyOf(plaintext), randomCipher);
15      return copyOf(randomCipher);
16    }
17    public byte[] decrypt(byte[] ciphertext) {
18      if (!log.containsCiphertext(ciphertext)) {
19        return copyOf( CryptoLib.symkey_decrypt(copyOf(key), copyOf(ciphertext)) );
20      } else {
21        return copyOf( log.lookup(ciphertext) );
22      }
23    }
24  }
```

Real Functionality.

```
1   public class SymEnc {
2     private byte[] key;
3
4     public SymEnc() {
```

```
5      key = CryptoLib.symkey_generateKey();
6    }
7    public byte[] encrypt(byte[] plaintext) {
8      return CryptoLib.symkey_encrypt(copyOf(key), copyOf(plaintext));
9    }
10   public byte[] decrypt(byte[] ciphertext) {
11     return CryptoLib.symkey_decrypt(copyOf(key), copyOf(ciphertext));
12   }
13 }
```

D.5. Nonce Generation

Ideal Functionality.

```
1  public class NonceGen {
2    public NonceGen() {
3    }
4    public byte[] newNonce() {
5      byte[] nonce = null;
6      // keep asking for a nonce until we get a fresh value
7      while( nonce==null || log.contains(nonce) ) {
8        nonce = CryptoLib.newNonce();
9      }
10     log.add(nonce); // log the nonce
11     return nonce;
12   }
13 }
```

Real Functionality.

```
1  public class NonceGen {
2    public NonceGen() {
3    }
4    public byte[] newNonce() {
5      return CryptoLib.newNonce();
6    }
7  }
```

E. Case Studies

In this section, we present the part of code of the case studies which have been proved being noninterferent either only by the fully automatic tool Joana or by combining Joana with the interactive theorem prover KeY.

As for the cryptographic functionalities, we notice that all the code exhibited here falls into the fragment of Java captured by the extensions of the Jinja language presented in Appendix B.

E.1. A Cloud Storage System

In this section, we report the full code of the setup and of the client, the two main classes which have been formally verified (along with the cryptographic functionalities proposed in Appendices D.2-D.5 and the class modeling the environment in Appendix C) by the Joana tool.

The `Setup` class.

```
public class Setup {
 public static final int HONEST_CLIENT_ID = 100;

 public static void main(String args[]) {
  setup(true);
 }

 public static void setup(boolean secret_bit) {
  // Create and register the client
  // (we consider one honest client; the remaining clients will be subsumed
  // by the adversary)
  SymEnc client_symenc = new SymEnc();
  Decryptor client_decryptor = new Decryptor();
  Signer client_signer = new Signer();
  Client client = null;
  try {
   RegisterEnc.registerEncryptor(client_decryptor.getEncryptor(),
     HONEST_CLIENT_ID, Params.PKI_ENC_DOMAIN);
   RegisterSig.registerVerifier(client_signer.getVerifier(),
     HONEST_CLIENT_ID, Params.PKI_DSIG_DOMAIN);
   client = new Client(HONEST_CLIENT_ID, client_symenc,
     client_decryptor, client_signer, new NetworkReal());
  } catch (RegisterEnc.PKIError e) { // encryptor registration failed -- id already registered
   return;
  } catch (RegisterSig.PKIError e) { // verifier registration failed -- id already registered
   return;
  } catch (NetworkError e) { // registration failed -- problems with the connection
   return;
  }

```

```
31    while( Environment.untrustedInput() != 0 ) {
32     // the adversary decides what to do:
33     int action = Environment.untrustedInput();
34     switch (action) {
35      case 0: // client.store
36       byte[] label = Environment.untrustedInputMessage();
37       byte[] msg1 = Environment.untrustedInputMessage();
38       byte[] msg2 = Environment.untrustedInputMessage();
39       if (msg1.length != msg2.length) break;
40       byte[] msg = new byte[msg1.length];
41       for (int i=0; i<msg1.length; ++i) {
42        try {
43         msg[i] = (secret_bit ? msg1[i] : msg2[i]);
44        } catch (Exception e) { }
45       }
46       try {
47        client.store(msg, label);
48       } catch(Exception e) {}
49      break;
50      case 1: // client.retrieve
51       label = Environment.untrustedInputMessage();
52       try {
53        client.retrieve(label);   // the result (the retrieved message) is ignored
54       }
55       catch(Exception e) {}
56      break;
57      case 2: // registering a corrupted encryptor
58       byte[] pub_key = Environment.untrustedInputMessage();
59       int enc_id = Environment.untrustedInput();
60       Encryptor corrupted_encryptor = new Encryptor(pub_key);
61       try {
62        RegisterEnc.registerEncryptor(corrupted_encryptor, enc_id, Params.PKI_ENC_DOMAIN);
63       } catch (Exception e) {}
64      break;
65      case 3: // registering a corrupted verifier
66       byte[] verif_key = Environment.untrustedInputMessage();
67       int verif_id = Environment.untrustedInput();
68       Verifier corrupted_verifier = new Verifier(verif_key);
69       try {
70        RegisterSig.registerVerifier(corrupted_verifier, verif_id, Params.PKI_DSIG_DOMAIN);
71       } catch (Exception e) {}
72      break;
73     }
74    }
75   }
76  }
```

The `Client` class.

```
77  public class Client {
78   private SymEnc symenc;
79   private Decryptor decryptor;
80   private Signer signer;
81   private Verifier verifier;
```

```
82  private Encryptor server_enc;
83  private Verifier server_ver;
84
85  private int userID;
86  private LabelList lastCounter;
87  private NonceGen nonceGen;
88
89  private NetworkInterface net;
90
91  public Client(int userID, SymEnc symenc, Decryptor decryptor, Signer signer, NetworkInterface net)
92    throws RegisterEnc.PKIError, RegisterSig.PKIError, NetworkError {
93   this.symenc = symenc;
94   this.decryptor = decryptor;
95   this.signer = signer;
96   this.verifier = signer.getVerifier();
97   this.server_enc = RegisterEnc.getEncryptor(Params.SERVER_ID, Params.PKI_ENC_DOMAIN);
98   this.server_ver = RegisterSig.getVerifier(Params.SERVER_ID, Params.PKI_DSIG_DOMAIN);
99   this.userID = userID;
100  this.net=net;
101  lastCounter = new LabelList(); // for each label maintains the last counter
102  nonceGen = new NonceGen();
103 }
104
105 public void store(byte[] msg, byte[] label) throws NetworkError, StorageError {
106  int serverLastCounter = getServerLastCounter(label);
107  int ourCounter = lastCounter.get(label);
108  // note that if 'label' has not been used yet, lastCounter.get(label) returns -1
109  if(serverLastCounter<ourCounter)
110   // the server is misbehaving (his counter is expected to be higher)
111   throw new IncorrectReply();
112  else if(serverLastCounter>ourCounter){
113   // we aren't up to date with the current counter stored in the server
114   lastCounter.put(label, serverLastCounter);
115   throw new CounterOutOfDate();
116  }
117  // otherwise they are the same!
118  int counter = ourCounter+1;
119
120  byte[] encrMsg = symenc.encrypt(msg);
121
122  // Encoding the message that has to be signed: (STORE, (label, (counter, encMsg)))
123  byte[] counter_msg = MessageTools.concatenate(MessageTools.intToByteArray(counter), encrMsg);
124  byte[] label_counter_msg = MessageTools.concatenate(label, counter_msg);
125  byte[] store_label_counter_msg = MessageTools.concatenate(Params.STORE, label_counter_msg);
126  /* HANDLE THE SERVER RESPONSE
127   * Expected server's responses (encrypted with the client's public key):
128   *    ((signClient, STORE_OK), signServer)            or
129   *    ((signClient, (STORE_FAIL, lastCounter)), signServer)
130   * where:
131   * - signServer: signature of all the previous tokens
132   * - signClient: signature of the message for which we are receiving the response
133   * - lastCounter: the higher value of the counter associated with label, as stored by the server
134   */
```

```
  ServerResponse response = sendPayloadToServer(store_label_counter_msg);
  // response.info is either (STORE_OK, {}) or (STORE_FAIL, lastCounter)
  if(Arrays.equals(response.tag, Params.STORE_OK)){  // message successfully stored
   // we can save the counter used to send the message
   lastCounter.put(label, counter);
   return;
  }
  else if(Arrays.equals(response.tag, Params.STORE_FAIL)){
   // the server hasn't accepted the request, because it claims
   byte[] serverCounter = response.info; // to have a higher counter for this label
   if(serverCounter.length!=4)
   // since lastCounter is supposed to be a integer, its length must be 4 bytes
    throw new IncorrectReply();
   serverLastCounter = MessageTools.byteArrayToInt(serverCounter);
   if (serverLastCounter<=counter)
   // the server is misbehaving (his counter is expected to be higher)
    throw new IncorrectReply();
   else if (serverLastCounter>counter){
   // we aren't up to date with the current counter stored in the server
    lastCounter.put(label, serverLastCounter);
    throw new CounterOutOfDate();
   }
  }
  else
   throw new IncorrectReply();
 }

 public byte[] retrieve(byte[] label) throws NetworkError, StorageError {
  int counter = getServerLastCounter(label);
  // pick the last the counter we stored
  int ourCounter = lastCounter.get(label);
  // note that if 'label' has not been used yet, lastCounter.get(label) returns -1
  if(counter<ourCounter)
   // the server is misbehaving (his counter is expected to be higher)
   throw new IncorrectReply();
  if(counter<0)
   // if counter<0 now we are sure that the server doesn't have anything under this label
   return null;
  // create the message to send
  byte[] label_counter = MessageTools.concatenate(label, MessageTools.intToByteArray(counter));
  byte[] retrieve_label_counter = MessageTools.concatenate(Params.RETRIEVE, label_counter);
  /* HANDLE THE SERVER RESPONSE
   * Expected server's responses (encrypted with the client's public key):
   *  ((signClient, (RETRIEVE_OK, (encMsg, signEncrMsg))), signServer)  or
   *  ((signClient, RETRIEVE_FAIL, {}), signServer)
   * where:
   * - signServer: signature of all the previous tokens
   * - signClient: signature of the message for which we are receiving the response
   * - signEncMsg: the signature of ((STORE, (label, (counter, encrMsg)))
   */
  ServerResponse response = sendPayloadToServer(retrieve_label_counter);
  // response.inf is either (RETRIEVE_OK, (encMsg, signEncrMsg)) or (RETRIEVE_FAIL,{})

```

```
188    // analyze the response tag
189    if(Arrays.equals(response.tag, Params.RETRIEVE_OK)){
190      byte[] encrMsg = MessageTools.first(response.info);
191      byte[] signMsg = MessageTools.second(response.info);
192      // check whether the signMsg is the signature for the STORE request with encrMsg
193      // which is of the form (STORE, (label, (counter, encrMsg)))
194      byte[] counter_msg = MessageTools.concatenate(MessageTools.intToByteArray(counter), encrMsg);
195      byte[] label_counter_msg = MessageTools.concatenate(label, counter_msg);
196      byte[] store_label_counter_msg = MessageTools.concatenate(Params.STORE, label_counter_msg);
197      if(!verifier.verify(signMsg, store_label_counter_msg))
198       // the server hasn't replied with the encrypted message we requested
199       throw new IncorrectReply();
200      // everything is ok; decrypt the message and return it
201      return symenc.decrypt(encrMsg);
202    }
203    else if(Arrays.equals(response.tag, Params.RETRIEVE_FAIL)){
204      // The server claims that it counldn't retrieve the message.
205      // But because the 'counter' is saved only after the server acknowledges
206      // that the message was successfully stored, it should not happen.
207      throw new IncorrectReply();
208    }
209    else
210     throw new MalformedMessage();
211   }
212
213   private class ServerResponse {
214    byte[] tag;
215    byte[] info;
216
217    ServerResponse(byte[] tag, byte[] info) {
218      this.tag = tag;
219      this.info = info;
220    }
221   }
222
223   /**
224    * Retrieve from the server the highest counter related to (clientID, label)
225    * If there isn't any counter related to this pair, return -1
226    */
227   private int getServerLastCounter(byte[] label) throws NetworkError, StorageError {
228    // pick a nonce
229    byte[] nonce = nonceGen.newNonce();
230    byte[] label_nonce=MessageTools.concatenate(label, nonce);
231    byte[] store_label_nonce=MessageTools.concatenate(Params.GET_COUNTER, label_nonce);
232    ServerResponse response = sendPayloadToServer(store_label_nonce);
233    // analyze the response tag
234    if(!Arrays.equals(response.tag, Params.LAST_COUNTER))
235     throw new MalformedMessage();
236    byte[] lastCounter_nonceResp = response.info;
237    byte[] lastCounter = MessageTools.first(lastCounter_nonceResp);
238    byte[] nonceResp = MessageTools.second(lastCounter_nonceResp);
239    if(!Arrays.equals(nonce, nonceResp))
240     throw new IncorrectReply();
```

```
241   if(lastCounter.length!=4)
242    throw new MalformedMessage();
243   return MessageTools.byteArrayToInt(lastCounter);
244  }
245
246  /**
247   * Sign the payload, add userID, encrypt with PKE and send everything to the server.
248   * Decrypt and validate the server response.
249   */
250  private ServerResponse sendPayloadToServer(byte[] payload)
251                                            throws MalformedMessage,
252 NetworkError {
253    // sign the message with the client private key
254    byte[] signClient = signer.sign(payload);
255    byte[] msgWithSignature = MessageTools.concatenate(payload, signClient);
256    // encrypt the (userID, ([payload], clientSign)) with the server public key
257    byte[] msgToSend = server_enc.encrypt(MessageTools.concatenate(
258                     MessageTools.intToByteArray(userID), msgWithSignature));
259    // Shape of msgToSend:
260    //           (userID, ([payload], signClient))
261    // where signClient is the signature of [payload]
262
263    // send the message to the server
264    byte[] encryptedSignedResp = net.sendRequest(msgToSend);
265    // Decrypt the validate the message in order to make sure
266    // that it is a response to the client's request.
267    return decryptValidateResp(encryptedSignedResp, signClient);
268  }
269
270  /**
271  * Decrypt the message, verify that it's a response of the server to our request
272  * (otherwise an exception is thrown).
273  */
274  private ServerResponse decryptValidateResp(byte[] encryptedSignedResponse,
275                                            byte[] signRequest) throws MalformedMessage {
276    // decrypt the message with the client private key and parse it
277    byte[] signedResponse = decryptor.decrypt(encryptedSignedResponse);
278    byte[] payload = MessageTools.first(signedResponse);
279    byte[] signServer = MessageTools.second(signedResponse);
280    // if the signature isn't correct, the message is malformed
281    // (note that the signature is incorrect even if one or both messages are empty)
282    if (!server_ver.verify(signServer, payload))
283     throw new MalformedMessage();
284    // check whether this is a response to the client's request as identified by signRequest
285    byte[] signatureClient = MessageTools.first(payload);
286    if(!Arrays.equals(signatureClient, signRequest))
287     throw new MalformedMessage();
288    byte[] response = MessageTools.second(payload);
289    // response should be of the form (tag, info), where info may be empty
290    return new ServerResponse(MessageTools.first(response), MessageTools.second(response));
291  }
292
293  public class StorageError extends Exception {}
```

```
294  /**
295  * Exception thrown when the response is invalid and demonstrates that the server
296  * has misbehaved (the server has be ill-implemented or malicious).
297  */
298  public class IncorrectReply extends StorageError {}
299  /**
300  * Exception thrown when the response of the server does not conform
301  * to an expected format (we get, for instance, a trash message or a response
302  * to a different request).
303  */
304  public class MalformedMessage extends StorageError {}
305  /**
306  * Exception thrown when the server is not able to store the message we sent to it, e.g.
307  * because it has always an higher counter related to our label.
308  */
309  public class StoreFailure extends StorageError {}
310  /**
311  * Exception thrown when the lastCounter provided by the server is higher than our counter.
312  * Before throwing this exception we should update our counter to the server one.
313  */
314  public class CounterOutOfDate extends StorageError{}
315  /**
316  * List of labels.
317  * For each 'label' maintains an counter representing
318  * how many times the label has been used.
319  */
320  static private class LabelList {
321    static class Node {
322      byte[] key;
323      int counter;
324      Node next;
325
326      public Node(byte[] key, int counter, Node next) {
327        this.key = key;
328        this.counter = counter;
329        this.next = next;
330      }
331    }
332    private Node firstElement = null;
333
334    public void put(byte[] key, int counter) {
335      for(Node tmp = firstElement; tmp != null; tmp=tmp.next)
336       if( Arrays.equals(key, tmp.key) ){
337         tmp.counter = counter;
338         return;
339       }
340      firstElement = new Node(key, counter, firstElement);
341    }
342
343    public int get(byte[] key) {
344      for(Node tmp = firstElement; tmp != null; tmp=tmp.next)
345      if( Arrays.equals(key, tmp.key)  )
346       return tmp.counter;
```

```
      return -1; // if the label is not present, return -1
    }
  }
}
```

E.2. An E-voting Machine with Auditing Procedures

In this section, we present the code of the setup, of the voting machine, and of the bulletin board, the three main classes which, along with the cryptographic functionalities proposed in Appendices D.2-D.5 and the class modeling the environment in Appendix C, have been formally verified by the combination of Joana and KeY. In what follows, we omit the declaration of the class `EntryQueue` which contains the following methods:

– `add` to add a byte array entry;

– `getEntries` to return a byte array containing the concatenation of all the entries added so far.

The full version of the code is available at [STB+14a].

The `Setup` class.

```
public class Setup {
 private static VotingMachine VM;
 private static BulletinBoard BB;
 // one secret bit
 private static boolean secret;
 // the correct result
 static int[] correctResult; // CONSERVATIVE EXTENSION

 private static int[] createChoices(int numberOfVoters, int numberOfCandidates) {
  final int[] choices = new int[numberOfVoters];
   for (int i=0; i<numberOfVoters; ++i) {
    choices[i] = Environment.untrustedInput();
   }
  return choices;
 }

 private static int[] computeResult (int[] choices, int numberOfCandidates) {
  int[] res = new int[numberOfCandidates];
  for (int i=0; i<choices.length; i++)
   ++res[choices[i]];
  return res;
 }

 private static boolean equalResult(int[] r1, int[] r2) {
  for (int j= 0; j<r1.length; j++)
   if (r1[j]!=r2[j]) return false;
  return true;
 }

 public static void main (String[] a) throws Throwable {
```

```
31  // Determine the number of candidates and the number of voters:
32  int numberOfCandidates = Environment.untrustedInput();
33  int numberOfVoters = Environment.untrustedInput();
34  if (numberOfVoters<=0 || numberOfCandidates<=0)
35   throw new Throwable();       // abort
36  // Create and register decryptor/encryptor of auditors:
37  Decryptor audit_decryptor = new Decryptor();
38  Encryptor audit_encryptor = audit_decryptor.getEncryptor();
39  RegisterEnc.registerEncryptor(audit_encryptor, Params.AUDITORS_ID, Params.ENC_DOMAIN);
40  // Create and register signer/verifier of the voting machine
41  Signer vm_signer = new Signer();
42  Verifier vm_verifier = vm_signer.getVerifier();
43  RegisterSig.registerVerifier(vm_verifier, Params.VOTING_MACHINE_ID, Params.SIG_DOMAIN);
44  // Create the voting machine and the bulletin board:
45  VM = new VotingMachine(numberOfCandidates, audit_encryptor, vm_signer);
46  BB = new BulletinBoard(vm_verifier);
47
48  // let the environment determine two vectors of choices
49  int[] choices0 = createChoices(numberOfVoters, numberOfCandidates);
50  int[] choices1 = createChoices(numberOfVoters, numberOfCandidates);
51  // check that those vectors give the same result
52  int[] r0 = computeResult(choices0, numberOfCandidates);
53  int[] r1 = computeResult(choices1, numberOfCandidates);
54  if (!equalResult(r0,r1))
55   throw new Throwable(); // abort if the two vectors do not yield the  same  result
56
57  // store correct result
58  correctResult = r1; // CONSERVATIVE EXTENSION
59
60  // the main loop
61  final int N = Environment.untrustedInput();
62  // the environment decides for how long the system runs
63  int voterNr = 0;
64  for(int i=0; i<N; ++i) {
65   int action = Environment.untrustedInput();
66   switch(action) {
67     // Importantly, the vote collection is done directly in the method collectBallot (without
68     // first sending the choice to any server).
69     case 0: // the next voter votes
70      if (voterNr<numberOfVoters) {
71        int choice = secret ? choices0[voterNr] : choices1[voterNr];
72        VM.collectBallot(choice);
73        ++voterNr;
74      }
75     break;
76     case 1: // make the voting machine publish the current (encrypted) log
77       VM.publishLog();
78     break;
79     case 2: // audit (this step altogether should not change the result)
80       int audit_choice = Environment.untrustedInput();
81       int sqnumber = VM.collectBallot(audit_choice);
82       Environment.untrustedOutput(sqnumber);
83       VM.publishLog();
```

```
84          VM.cancelLastBallot();
85        break;
86        case 3: // deliver a message to the bulletin board
87         byte[] request = Environment.untrustedInputMessage();
88         BB.onPost(request);
89        break;
90        case 4: // make the bulleting board send its content over the network
91         BB.onRequestContent();
92        break;
93       }
94      }
95     // make the voting machine publish the result (only if all the voters have voted)
96     if (voterNr == numberOfVoters)
97      VM.publishResult();
98    }
99   }
```

The `VotingMachine` class.

```
100  public class VotingMachine {
101   public class InnerBallot {
102    public final int votersChoice;
103    public final int voteCounter;
104    public final long timestamp;
105
106    public InnerBallot(int choice, int counter, long ts) {
107     votersChoice = choice;
108     voteCounter = counter;
109     timestamp = ts;
110    }
111   }
112
113   public class InvalidVote extends Exception {}
114   public class InvalidCancelation extends Exception {}
115
116   private final Encryptor bb_encryptor;
117   private final Signer signer;
118   private int numberOfCandidates;
119   private int[] votesForCandidates;
120   private int operationCounter, voteCounter;
121   private EntryQueue entryLog;
122   private InnerBallot lastBallot;
123
124   public VotingMachine(int numberOfCandidates, Encryptor bb_encryptor,
125                                             Signer signer) {
126    this.numberOfCandidates=numberOfCandidates;
127    this.bb_encryptor=bb_encryptor;
128    this.signer=signer;
129    votesForCandidates = new int[numberOfCandidates];
130    entryLog = new EntryQueue();
131    operationCounter=0;
132    voteCounter=0;
133    lastBallot=null;
134   }
```

```
135
136  public int collectBallot(int votersChoice) throws InvalidVote {
137   if (votersChoice < 0 || votersChoice >= numberOfCandidates)
138    throw new InvalidVote();
139   // increase the vote for the corresponding candidate
140   votesForCandidates[votersChoice]++;
141   // create a new inner ballot
142   lastBallot = new InnerBallot(votersChoice, ++voteCounter, Timestamp.get());
143   // log and send a new entry
144   logAndSendNewEntry(Params.VOTE);
145   return operationCounter;
146  }
147
148  public void cancelLastBallot() throws NetworkError, InvalidCancelation {
149   if(lastBallot==null)
150    throw new InvalidCancelation();
151   votesForCandidates[lastBallot.votersChoice]--;
152   logAndSendNewEntry(Params.CANCEL);
153   lastBallot = null;
154  }
155
156  public void publishResult() throws NetworkError {
157   signAndPost(Params.RESULTS, getResult(), signer);
158  }
159
160  public void publishLog() throws NetworkError {
161   signAndPost(Params.LOG, entryLog.getEntries(), signer);
162  }
163
164  private void logAndSendNewEntry(byte[] tag) {
165   byte[] entry = createEncryptedEntry(++operationCounter, tag, lastBallot, bb_encryptor, signer);
166   entryLog.add(copyOf(entry));
167   try {
168    signAndPost(Params.MACHINE_ENTRY, entry, signer);
169   } catch (Exception ex) {}
170    // this may cause an NetworkError, but even if we do not get any exception,
171    // there is no guarantee that the entry was indeed delivered to the bulletin board,
172    // so we ignore the problem
173  }
174  /**
175   * Create and return the new entry:
176   *  ( operationCounter, ENC_BB{ TAG, timestamp, voterChoice, voteCounter} )
177   */
178  private byte[] createEncryptedEntry(int operationCounter, byte[] tag,
179    InnerBallot inner_ballot, Encryptor encryptor, Signer signer) {
180    byte[] vote_voteCounter = concatenate(
181    intToByteArray(inner_ballot.votersChoice),
182    intToByteArray(inner_ballot.voteCounter));
183    byte[] ballot = concatenate(longToByteArray(inner_ballot.timestamp),
184                              vote_voteCounter);
185    byte[] tag_ballot= concatenate(tag, ballot);
186    byte[] encrMsg = encryptor.encrypt(tag_ballot);
187    byte[] entry = concatenate( intToByteArray(operationCounter), encrMsg);
```

```
188     return entry;
189   }
190   /**
191    * Sign_VM [ TAG, timestamp, message ]
192    */
193   private static void signAndPost(byte[] tag, byte[] message,
194                                   Signer signer) throws NetworkError {
195    long timestamp = Timestamp.get();
196    byte[] tag_timestamp = concatenate(tag, longToByteArray(timestamp));
197    byte[] payload = concatenate(tag_timestamp, message);
198    byte[] signature = signer.sign(payload);
199    byte[] signedPayload = concatenate(payload, signature);
200    NetworkClient.send(signedPayload, Params.DEFAULT_HOST_BBOARD,
201                                      Params.LISTEN_PORT_BBOARD);
202   }
203
204   private byte[] getResult() {
205    int[] _result = new int[numberOfCandidates];
206    for (int i=0; i<numberOfCandidates; ++i) {
207     int x = votesForCandidates[i];
208     // CONSERVATIVE EXTENSION:
209     // PROVE THAT THE FOLLOWING ASSINGMENT IS REDUNDANT
210     x = Setup.correctResult[i];
211     _result[i] = x;
212    }
213    return formatResult(_result);
214   }
215
216   private static byte[] formatResult(int[] _result) {
217    String s = "Result␣of␣the␣election:\n";
218    for( int i=0; i<_result.length; ++i ) {
219      s += "␣␣Number␣of␣votes␣for␣candidate␣" + i + ":␣" + _result[i] + "\n";
220    }
221    return s.getBytes();
222   }
223  }
```

The `BulletinBoard` class.

```
224  public class BulletinBoard {
225   Verifier verifier;
226   EntryQueue entryLog;
227
228   public BulletinBoard(Verifier verifier) throws NetworkError {
229    this.verifier=verifier;
230    entryLog= new EntryQueue();
231   }
232   /*
233    * Reads a message, checks if it comes from the voting machine, and,
234    * if this is the case, adds it to the maintained list of messages.
235    */
236   public void onPost(byte[] request) throws NetworkError {
237    byte[] message = first(request);
238    byte[] signature = second(request);
```

```
239
240     if(verifier.verify(signature, message))
241      entryLog.add(request);
242    }
243    /*
244     * Output its content, that is the concatenation of
245     * all the message in the maintained list of messages.
246     */
247    public byte[] onRequestContent() throws NetworkError {
248     return entryLog.getEntries();
249    }
250   }
```

E.3. The Mix Server of sElect

In this section, we present the Java implementation of the cryptographic core of the mix server of sElect and of the setup class which, along with the cryptographic functionalities proposed in Appendices D.2-D.5 and the class modeling the environment in Appendix C, are in the process of being formally verified by the combination of Joana and KeY.

The Setup class.

```
1   public final class Setup {
2    private static boolean setEquality(byte[][] arr1, byte[][] arr2) {
3     if(arr1.length!=arr2.length) return false;
4     byte[][] a1=MessageTools.copyOf(arr1);
5     byte[][] a2=MessageTools.copyOf(arr2);
6     Utils.sort(a1, 0, a1.length);
7     Utils.sort(a2, 0, a2.length);
8     for(int i=0;i<a1.length;i++)
9      if(!MessageTools.equal(a1[i],a2[i]))
10      return false;
11    return true;
12   }
13
14   private static boolean secret; // the HIGH value
15
16   public static void main (String[] a) throws Throwable {
17    Decryptor mixDecr = new Decryptor();
18    Encryptor mixEncr = mixDecr.getEncryptor();
19    Signer mixSign = new Signer();
20    Signer precServSign = new Signer();
21    Verifier precServVerif = precServSign.getVerifier();
22
23    byte[] electionID = Environment.untrustedInputMessage();
24    MixServer mixServ = new  MixServer(mixDecr, mixSign, precServVerif, electionID);
25
26    // let the adversary choose how many messages have to
27    // be sent to the mix server
28    int numberOfMessages = Environment.untrustedInput();
29    // let the adversary decide the length of the messages
30    // all the messages must have the same length:
```

```
31    int lengthOfTheMessages = Environment.untrustedInput();
32    // let the environment determine the two arrays of messages
33    byte[][] msg1 = new byte[numberOfMessages][];
34    byte[][] msg2 = new byte[numberOfMessages][];
35    for(int i=0; i<numberOfMessages; ++i){
36     msg1[i] = Environment.untrustedInputMessage();
37     msg2[i] = Environment.untrustedInputMessage();
38     if(msg1[i].length!=lengthOfTheMessages || msg2[i].length!=lengthOfTheMessages)
39      throw new Throwable();
40    }
41    // the two vectors must be two permutations of the same set
42    if(!setEquality(msg1, msg2))
43     throw new Throwable();
44
45    // CONSERVATIVE EXTENSION
46    ConservativeExtension.storeMessages(msg1);
47
48    // encrypt each message, along with the election ID as expected by the mix server
49    byte[][] encrMsg = new byte[numberOfMessages][];
50    for(int i=0; i<numberOfMessages; ++i){
51     byte[] msg = secret ? msg1[i] : msg2[i];
52     byte[] msg = new byte[lengthOfTheMessages];
53     for (int j=0; j<msg.length; j++) {
54      byte b1 = msg1[i][j];
55      byte b2 = msg2[i][j];
56      byte b = secret ? b1:b2;
57      msg[j] = b;
58     }
59     encrMsg[i] = mixEncr.encrypt(MessageTools.concatenate(electionID, msg));
60    }
61    Utils.sort(encrMsg, 0, encrMsg.length);
62
63    byte[] asAMessage=Utils.concatenateMessageArray(encrMsg, encrMsg.length);
64    // add election id, tag and sign
65    byte[] elID_ballots = MessageTools.concatenate(electionID, asAMessage);
66    byte[] input = MessageTools.concatenate(Tag.BALLOTS, elID_ballots);
67    byte[] signatureOnInput = precServSign.sign(input);
68    byte[] signedInput = MessageTools.concatenate(input, signatureOnInput);
69
70    // send the message over the network, controlled by the adversary
71    Environment.untrustedOutputMessage(signedInput);
72    // retrieve the message from the network
73    byte[] mixServerInput=Environment.untrustedInputMessage();
74
75    // let the mix server process the ballots
76    byte[] mixServerOutput=mixServ.processBallots(mixServerInput);
77
78    // send the output of the mix server over the network
79    Environment.untrustedOutputMessage(mixServerOutput);
80   }
81  }
```

The `MixServer` class.

```
82  public class MixServer {
83
84   private final Signer signer;
85   private final Decryptor decryptor;
86   private final Verifier precServVerif;
87   private final byte[] electionID;
88
89   public static class MalformedData extends Exception {
90    public int errCode;
91    public String description;
92    public MalformedData(int errCode, String description) {
93     this.errCode = errCode;
94     this.description = description;
95    }
96    public String toString() {
97     return "Mix␣Server␣Error:␣" + description;
98    }
99   }
100
101  public static class ServerMisbehavior extends Exception {
102   public int errCode;
103   public String description;
104
105   public ServerMisbehavior(int errCode, String description) {
106    this.errCode = errCode;
107    this.description = description;
108   }
109
110   public String toString() {
111    return "Previous␣Server␣Misbehavior:␣" + description;
112   }
113  }
114
115  public MixServer(Decryptor decryptor, Signer signer,
116                   Verifier precServVerif, byte[] electionID) {
117   this.signer = signer;
118   this.decryptor = decryptor;
119   this.electionID = electionID;
120   this.precServVerif = precServVerif;
121  }
122
123  public byte[] processBallots(byte[] data) throws MalformedData, ServerMisbehavior {
124   byte[] tagged_payload = MessageTools.first(data);
125   byte[] signature = MessageTools.second(data);
126   if (!precServVerif.verify(signature, tagged_payload))
127    throw new MalformedData(1, "Wrong␣signature");
128   byte[] tag = MessageTools.first(tagged_payload);
129   if (!MessageTools.equal(tag, Tag.BALLOTS))
130    throw new MalformedData(2, "Wrong␣tag");
131   byte[] payload = MessageTools.second(tagged_payload);
132   byte[] el_id = MessageTools.first(payload);
133   if (!MessageTools.equal(el_id, electionID))
134    throw new MalformedData(3, "Wrong␣election␣ID");
```

```
135    byte[] ballotsAsAMessage = MessageTools.second(payload);
136
137    ArrayList<byte[]> entries = new ArrayList<byte[]>();
138    byte[] last = null;
139    int numberOfEntries = 0;
140    for( MessageSplitIter iter = new MessageSplitIter(ballotsAsAMessage);
141          iter.notEmpty(); iter.next() ) {
142     byte[] current = iter.current();
143     if (last!=null && Utils.compare(last, current)>0)
144      throw new ServerMisbehavior(-2, "Ballots␣not␣sorted");
145     if (last!=null && Utils.compare(last, current)==0)
146      throw new ServerMisbehavior(-3, "Duplicate␣ballots");
147     last = current;
148     byte[] decryptedBallot = decryptor.decrypt(current);
149     if (decryptedBallot == null){
150      System.out.println("[MixServer.java]␣Decryption␣failed␣for␣ballot␣#" + numberOfEntries);
151      continue;
152     }
153     byte[] elID = MessageTools.first(decryptedBallot);
154     if (elID!=null || MessageTools.equal(elID, electionID))
155      entries.add(MessageTools.second(decryptedBallot));
156     else
157      System.out.println("[MixServer.java]␣Ballot␣#" + numberOfEntries + "␣invalid");
158     ++numberOfEntries;
159    }
160    byte[][] entr_arr = new byte[entries.size()][];
161    entries.toArray(entr_arr);
162    Utils.sort(entr_arr, 0, numberOfEntries);
163
164    /** CONSERVATIVE EXTENSION:
165     *   PROVE THAT THE FOLLOWING ASSINGMENT IS REDUNDANT
166     */
167    entr_arr = ConservativeExtension.retrieveSortedMessages();
168
169    // format entries as one message, and the proper tags, and sign the result
170    byte[] entriesAsAMessage = Utils.concatenateMessageArray(entr_arr, numberOfEntries);
171    byte[] elID_entriesAsAMessage = MessageTools.concatenate(electionID, entriesAsAMessage);
172    byte[] result = MessageTools.concatenate(Tag.BALLOTS, elID_entriesAsAMessage);
173    byte[] signatureOnResult = signer.sign(result);
174    byte[] signedResult = MessageTools.concatenate(result, signatureOnResult);
175
176    return signedResult;
177   }
178  }
```

The `ConservativeExtension` class.

```
179  public class ConservativeExtension{
180   private static byte[][] messages;
181
182   public static void storeMessages(byte[][] msg){
183    messages=MessageTools.copyOf(msg);
184   }
185   public static byte[][] retrieveSortedMessages(){
```

```
186     sort(messages, 0, messages.length);
187     return messages;
188   }
189 
190   private static void sort(byte[][] byteArrays, int fromIndex, int toIndex) {
191     if(fromIndex>=0 && toIndex<=byteArrays.length && fromIndex<toIndex){
192       for(int sorted=fromIndex+1; sorted<toIndex; ++sorted){
193         byte[] key=byteArrays[sorted]; // the item to be inserted
194         // insert key into the sorted sequence A[fomIndex, ..., sorted-1]
195         int i;
196         for(i=sorted-1; i>=fromIndex && Utils.compare(byteArrays[i], key)>0; --i)
197           byteArrays[i+1]=byteArrays[i];
198         byteArrays[i+1]=key;
199       }
200     }
201   }
202 }
```

F. Formal Proofs

F.1. Proof of Theorem 2.5

The proof of Theorem 2.5 is quite similar to the proof of Theorem 2.3 which can be found in [KTG12b]. The main difference is in the communication through strings, where indeed string references are exchanged, leading E and S to share part of their state. However, since strings are *immutable objects*, i.e., their internal state cannot be modified after their creation, the part of the shared state of a system cannot be modified by the other one.

We note that the six assumptions $E.1$-$E.3$ and $S.1$-$S.3$ (presented in Section 2.6.2 and extended to deal with strings in Section 2.6.3) on the interface I_E and on the system S respectively guarantee that—even if, technically, some references are exchanged between E and S—the communication between them is, effectively, as if only pure values were exchanged.

We first introduce the notation in order to highlight how the communication between S and E occurs when, besides primitive types, also strings, simple objects, arrays, and exception are involved. Let I_E and S be as in the Theorem 2.5. Let $\emptyset \vdash E : I_E$ be an environment for S. We represent the run ρ of $E \cdot S$ as:

$$\rho = A_1[s_1,x_1]B_1[t_1,y_1]A_2[s_2,x_2]\dots B_{n-1}[t_{n-1},y_{n-1}]A_n[s_n,x_n],$$

where the square brackets contain all the information that is passed between the two systems.

Thanks to the restrictions $S.1$ and $S.2$ on simple objects and arrays, we can assume that, except for producing a fresh copy of them, E and S only share string references. We can then define the components of ρ in the following way:

- s_i is the state of the run at the end of the segment A_i, i.e., where the code of S is executed.
- $x_i = (C_i, m_i, \vec{a}_i)$ is a tuple denoting the call of the method m_i of the class C_i defined in E with arguments $\vec{a}_i$. The arguments vector $\vec{a}_i$ records primitive types, strings, *values* of arrays of primitive types (not the references of these arrays though), or *values* of simple objects (but, again, not their references), i.e., collections of all values of the fields of objects whose classes are defined in I_E.
- t_i is the state of the run at the end of the B_i segment, i.e., where the code of E is executed.
- y_i is the value returned by the method $C_i.m_i$ of E to S. Again, y_i may be a primitive value, a string, or a *value* either of an array (of primitive types), or of a simple object.

Furthermore, given a state s, let $s|_S$ be the part of s the system S can access and, similarly, let $s|_E$ be the part of s accessible by the environment E.

The following Lemma is an extension of a result, so called, state separation of E and S introduced in [KTG12a]: intuitively, for a representation of a run as above, B_i does not change

the part of the state that can be reached from S and, similarly, A_i does not change the part of the state that can be reached from E.

Lemma F.1 (State separation). *Let S and E be two systems like in Theorem 2.5 and whose run ρ is defined as above. Then, for each $i \in \{1,\dots,n-1\}$ we have:*

1. $s_i|_S = t_i|_S$,
2. $t_i|_E = s_{i+1}|_E$.

Proof. (Sketch) The first statement asserts that the part of the state accessible by S is, at the end of each segment A_i of the run, equal to the part of the state accessible by S at the end of the following segment B_i. Similarly, the second statement asserts that the part of the state accessible by E is, at the end of each segment B_i of the run, equal to the part of the state accessible by E at the end of the following segment A_{i+1}.

Although simple objects, arrays, and exceptions are passed between the system S and the environment E, in Sections 2.6.2 and 2.6.3 we imposed some restrictions which guarantee that the communication between S and E actually resembles exchange of only primitive values and string references. In particular, imposing that only fresh copies of references (different from strings) are passed to the enviorment and that every reference (different from string) the environment returns to the system is immediately cloned and not used afterwards allows us to conclude that E and S do not share any of this kind of references (except S to clone them when it receives them). Without loss of generality, we can therefore assume that the systems E and S exchange only primitive values or string references.

Since in Jinja+ (as in Java) data is passed *by value*, both primitive values and string references are copied when they are exchanged between these systems. However, while if only primitive values are exchanged, the part of the state accessible by E is disjoint from the part of the state accesible by S, in case of communication through strings, the system receiving references of data type string would, in theory, be able to access the part of the state accessible from the other system too. Nevertheless, since in Jinja+ (and in Java) string objects are immutable and without any field, the system which receives their references cannot modify them. Therefore, in any case, at the end of each segment of the run, the part of the state accessible by the system which is going to execute the next segment has remained unchanged as it was at the beginning the last segment. □

Because of this state separation, we can obtain two lemmas asserting that the part of the state accessible by each system and the values this system, at the end of each segment of its run, passes to the other one depend solely on the values exchanged between them so far.

Let us first introduce some notation and definitions. Let ρ and $\hat{\rho}$ be the runs of two systems, for example $E \cdot S$ and $E \cdot \hat{S}$. As already introduced before, $x_i/\hat{x}_i$ and $y_i/\hat{y}_i$ denote, respectively, the calls of the method m_i of the class C_i defined in E with arguments $\vec{a}_i/\vec{a}'_i$ and the values returned by them. We remember that they both record either primitive values or string references.

Let $f : \hat{R} \to R$ be a bijection from $\hat{R}$ to R, where $\hat{R}$ and R are subsets of the set of all string references, such that for each $\hat{r}$ and r such that $f(\hat{r}) = r$, their string values are the same. We denote it as $r \sim_f \hat{r}$. We extend the domain of f to expressions, states, configurations, and tuples

$x_i/\hat{x}_i$. We do it, in the standard way, by structural isomorphism. That is, for example, $e \sim_f \hat{e}$ holds if and only if e and $\hat{e}$ are (syntactically) equal, up to references occurring as their corresponding subexpressions which need to be in the relation $\sim_f$, too. For primitive values $v/\hat{v}$, we write $v \sim_f \hat{v}$, if simply $v = \hat{v}$.

Let $f\colon \hat{R}_f \to R_f$ and $f'\colon \hat{R}_{f'} \to R_{f'}$ be two congruences. We say that f' is *compatible* with f if for each $r \in \hat{R}_f \cap \hat{R}_{f'}$, we have $f(r) = f'(r)$. Moreover, we say that f' is an *extension* of f (or, alternatively, f is a restriction of f') if f' is compatible with f, and, in particular, $\hat{R}_f \subseteq \hat{R}_{f'}$.

Lemma F.2. *Let S_1, S_2 be two systems like the system S in Theorem 2.5 and let E be an I-environment for both S_1 and S_2. Let ρ be the run of $E \cdot S_1$ and $\hat{\rho}$ be the run of $E \cdot S_2$. If there exists a bijection $f\colon \hat{R}_f \to R_f$, where $\hat{R}_f$ and R_f denote the sets of all references in $\{\hat{x}_1, \dots, \hat{x}_k, \hat{y}_1, \dots, \hat{y}_{k-1}\}$ and in $\{x_1, \dots, x_k, y_1, \dots, y_{k-1}\}$ respectively, such that*

$$x_1 \sim_f \hat{x}_1, \dots, x_k \sim_f \hat{x}_k \quad \text{and} \quad y_1 \sim_f \hat{y}_1, \dots, y_{k-1} \sim_f \hat{y}_{k-1},$$

then, there exists a bijection $f'\colon \hat{R}_{f'} \to R_{f'}$, where $\hat{R}_{f'}$ and $R_{f'}$ denote the sets of references in the domains of the heaps of $\hat{t}_k|_E$ and $t_k|_E$ respectively, such that

$$t_k|_E \sim_{f'} \hat{t}_k|_E \quad \text{and} \quad y_k \sim_{f'} \hat{y}_k,$$

where f' is compatible with f.

Proof. By the premise of the lemma we know that there exists a bijection f such that for each $i \in \{1, \dots, k\}$ we have $x_i \sim_f \hat{x}_i$ and for each $j \in \{1, \dots, k-1\}$ we have $y_j \sim_f \hat{y}_j$. By the inductive hypothesis, we can assert that if there exists a bijection $g\colon \hat{R}_g \to R_g$ such that for each $i \in \{1, \dots, k-1\}$ we have $x_i \sim_g \hat{x}_i$ and for each $j \in \{1, \dots, k-2\}$ we have $y_j \sim_g \hat{y}_j$, then there exists a bijection $g'\colon \hat{R}_{g'} \to R_{g'}$, compatible with g, such that $t_{k-1}|_E \sim_{g'} \hat{t}_{k-1}|_E$ and $y_{k-1} \sim_{g'} \hat{y}_{k-1}$. We define the function g as a restriction of f, where $\hat{R}_g$ and R_g denote all references in $\{\hat{x}_1, \dots, \hat{x}_{k-1}, \hat{y}_1, \dots, \hat{y}_{k-2}\}$ and $\{x_1, \dots, x_{k-1}, y_1, \dots, y_{k-2}\}$, respectively. Then, we also have the function g' as above where $\hat{R}_{g'}$ and $R_{g'}$ denote the sets of references in the domains of the heaps of $\hat{t}_k|_E$ and $t_k|_E$, respectively. Moreover, as by Lemma F.1 $t_{k-1}|_E = s_k|_E$ and $\hat{t}_{k-1}|_E = \hat{s}_k|_E$, we can assert $s_k|_E \sim_{g'} \hat{s}_k|_E$.

We want to show that g' is compatible with f. We will prove that for each $\hat{a} \in \hat{R}_{g'} \cap \hat{R}_f$, $\hat{a}$ is in $\hat{R}_g$ too. Then, since f is an extension of g, we have $f(\hat{a}) = g(\hat{a})$ and, since g is compatible with g', $f(\hat{a}) = g(\hat{a}) = g'(\hat{a})$, i.e., f and g' are compatible. The only references which may be in $\hat{R}_f \cap \hat{R}_{g'}$ but not in $\hat{R}_g$ are those that are in $\hat{x}_k$. Let $\hat{a}$ be a reference of the method call denoted by $\hat{x}_k$ which is also in $\hat{R}_{g'}$. Then, by the defintion of $\hat{R}_{g'}$, $\hat{a}$ is in the domain of the heap of $\hat{t}_{k-1}|_E = \hat{s}_k|_E$. In particular, if $\hat{a}$ is $\hat{y}_{k-1}$, by the definition of f and g', we have $f(\hat{y}_{k-1}) = y_{k-1} = g'(\hat{y}_{k-1})$. Otherwise we notice that, as $\hat{a}$ is part of $\hat{x}_k$, $\hat{a}$ is also in the domain of the heap of $\hat{s}_k|_S$. Since all references in the domain of the heap of $\hat{s}_k$ were uniquely created in a (previous) point of the run $\hat{\rho}$ and since in the domains of the heap of $\hat{s}_k|_E$ and $\hat{s}_k|_S$ there are only those references that can be reached by E and S respectively, $\hat{a}$ must have been created by one of the two systems and then passed to the other one either in a previous method call $\hat{x}_1, \dots, \hat{x}_{k-1}$ or in a value $\hat{y}_1, \dots, \hat{y}_{k-2}$ previously returned. That is, by the definition of g, $\hat{a}$ is in $\hat{R}_g$ too.

The segments B_k and $\hat{B}_k$ start with $(e_z[\{e'_k\}_{C_k}], s_k)$ and $(\hat{e}_z[\{\hat{e}'_k\}_{C_k}], \hat{s}_k)$ respectively, where $\{e_k\}_{C_k}/\{\hat{e}'_k\}_{C_k}$ are the blocks obtained by the static method call rule (Rule B.61 of Jinja+)

applied to the call described by $x_k/\hat{x}_k$. In particular, $\{e'_k\}_{C_k} = \{V_1\colon T_1,\ldots,V_n\colon T_n; V_1 := a_1,\ldots,V_n := a_n; \underline{e}\}_{C_k}$ and $\{\hat{e}'_k\}_{C_k} = \{V_1\colon T_1,\ldots,V_n\colon T_n; V_1 := \hat{a}_1,\ldots,V_n := \hat{a}_n; \underline{e}\}_{C_k}$, where for each $j \in \{1,\ldots,l\}$ we have $a_j \sim_f \hat{a}_j$ and $\underline{e}$ is the body of the method m_k of E. Then, since f and g' are compatible, we can assert $(\{e'_k\}_{C_k}, s_k|_E) \sim_h (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$ for a bijection $h\colon \{\hat{a}_1,\ldots,\hat{a}_n\} \cup \hat{R}_{g'} \to \{a_1,\ldots,a_n\} \cup R_{g'}$ defined in the following way: (i) $h(r) = f(r)$ if $r \in \{\hat{a}_1,\ldots,\hat{a}_n\}$; (ii) $h(r) = g'(r)$ if $r \in \hat{R}_{g'} \setminus \{\hat{a}_1,\ldots,\hat{a}_n\}$. That is, h maps references in $\hat{x}_k$ to references in x_k and references in the heap of $\hat{s}_k|_E$ to references in the heap of $s_k|_E$. We notice that h is compatible with both f and g', and therefore also with g.

As applicability of no Jinja+ rule depends on the particular value of references and as the blocks $\{e'_k\}_{C_k}/\{\hat{e}'_k\}_{C_k}$ are syntactically the same up to the references in their initial assignments, the rules applied in these blocks depend solely on $\underline{e}$. Let

$$(e_0,u_0) \to (e_1,u_1) \to \cdots \to (e_l,u_l),$$

be the segment B_k where its expressions are cut off from the set of classes belonging to S and where only the part of the state that E can modify is taken into account. In particular, $(e_0,u_0) = (\{e'_k\}_{C_k}, s_k|_E)$ and $(e_l,u_l) = (\{y_k\}_{C_k}, t_k|_E)$. Let

$$(\hat{e}_0,\hat{u}_0) \to (\hat{e}_1,\hat{u}_1) \to \cdots \to (\hat{e}_l,\hat{u}_l),$$

be the segment $\hat{B}_k$ pruned in the same way as B_k, where $(\hat{e}_0,\hat{u}_0) = (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$ and $(\hat{e}_l,\hat{u}_l) = (\{\hat{y}_k\}_{C_k}, \hat{t}_k|_E)$.

We show that for each pair of configurations $q_j/\hat{q}_j$ in the two (sub-)runs defined above, there exists a bijection $h_j\colon \hat{R}_j \to R_j$ such that $q_i = (e_j,s_j) \sim_{h_j} (\hat{e}_j,\hat{s}_j) = \hat{q}_j$ and such that h_j is a extension of the bijection h_{j-1} among (e_{j-1},s_{j-1}) and $(\hat{e}_{j-1},\hat{s}_{j-1})$. We do it by induction on the number of steps of execution.

`Base case`: $j = 0$. As we know $(\{e'_k\}_{C_k}, s_k|_E) \sim_h (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$, we have $h_0 = h$.

`Inductive Step`: Let us assume, by the inductive hypothesis, that there exists a bijection $h_j\colon \hat{R}_j \to R_j$ for $0 < j < n$ such that $(e_j,u_j) \sim_{h_j} (\hat{e}_j,\hat{u}_j)$ and such that h_j is an extension of h_{j-1}. Since $(e_j,u_j) \to (e_{j+1},u_{j+1})$ and $(\hat{e}_j,\hat{u}_j) \to (\hat{e}_{j+1},\hat{u}_{j+1})$, we prove that there exists a bijection $h_{j+1}\colon \hat{R}_{j+1} \to R_{j+1}$ such that $(e_{j+1},u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1},\hat{u}_{j+1})$ and $\hat{R}_j \subseteq \hat{R}_{j+1}$. We distinguish four cases depending on the behavior of the Jinja+ rule applied in the j-th step of computation:

a) The effect of the rule on the state is directly inherited from the reduction step of a subexpression in the hypotheses the rule (this case holds for all the subexpression reduction rules of Jinja+). In this case, there exist two subexpressions e and $\hat{e}$ of the expressions e_j and $\hat{e}_j$ respectively, whose reduction step directly determines the states $u_{j+1}/\hat{u}_{j+1}$. Since h_j is defined by structural isomorphism, (e,u_j) and $(\hat{e},\hat{u}_j)$ are under the bijection h_j too. By the inductive hyphothesis, the configurations resulting from the reduction step of the subexpressions $e/\hat{e}$ are under a bijection h_{j+1} which extends h_j. Hence, since h_{j+1} is also defined by structural isomorphism, we can assert that the configurations (e_{j+1},u_{j+1}) and $(\hat{e}_{j+1},\hat{u}_{j+1})$ are under the bijection h_{j+1}, too.

b) The rule creates a new reference in the state of computations (Rules B.16, B.62, B.71, B.72, B.76 of Jinja+). In this case, in (e_{j+1},u_{j+1}) and $(\hat{e}_{j+1},\hat{u}_{j+1})$ occur two fresh references a and

$\hat{a}$ unused in u_j and $\hat{u}_j$, respectively. Since, by the inductive hypothesis, h_j is an extension of $h_0 = h$, and, by the definition of h, $\mathtt{dom}(h)$ is the union of references in $\hat{x}_k$ with references in the heap of $\hat{u}_0 = \hat{s}_k|_E$, then $\mathtt{dom}(h_j)$ differs from $\mathtt{dom}(h_0)$ solely in the set of references which have been created in the segment $\hat{B}_k$ so far, i.e., which are in the heap of $\hat{u}_j$ but not in the heap of $\hat{u}_0$. Therefore, if $\hat{a}$ is not in the heap of $\hat{u}_j$, then it is not even in $\mathtt{dom}(h_j) = \hat{R}_j$. We can then extend the bijection h_j to another bijection h_{j+1} in the following way: (i) $\hat{R}_{j+1} = \hat{R}_j \cup \{\hat{a}\}$; (ii) for each $r \in \hat{R}_j$, we have $h_j(r) = h_{j+1}(r)$; (iii) $a = h_{j+1}(\hat{a})$. Since all the other references remain unchanged, we can conclude $(e_{j+1}, u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1}, \hat{u}_{j+1})$.

c) The rules update an entry in the state of computations (Rule B.20 of Jinja+ performs an update on the store, whereas Rules B.23 and B.68 perform an update on the heap). In these (three) rules, the expressions e_j and $\hat{e}_j$ are reduced to e_{j+1} and $\hat{e}_{j+1}$ by trimming the updating value off. Therefore, since no new references occur in $e_{j+1}/\hat{e}_{j+1}$, we have $e_{j+1} \sim_{h_j} \hat{e}_{j+1}$. Furthermore, the states $u_j/\hat{u}_j$ are updated with values which are in $e_j/\hat{e}_j$ and hence, in case these values are references, already under the bijection h_j. Therefore, the entries updated remain under the same bijection h_j, by which we can conclude $(e_{j+1}, u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1}, \hat{u}_{j+1})$, where $h_{j+1} = h_j$.

d) The rule leaves the state of the configurations unchanged: this case holds for all the expression reduction rules and all the exception propagation rules of Jinja+ not mentioned in the previous two items. Here, if in $e_{j+1}/\hat{e}_{j+1}$ occur new references, they must have been in $u_j/\hat{u}_j$ and therefore under the bijection h_j. (This case could for example happen in the field access rule, Rule B.22, if the field is not of primitive type.) We can then assert $e_{j+1} \sim_{h_j} \hat{e}_{j+1}$. Moreover, since their states remain unchanged, we can conclude $(e_{j+1}, u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1}, \hat{u}_{j+1})$, where $h_{j+1} = h_j$.

Therefore, since in each step j of these two runs the configurations are under a bijection h_j extending the initial bijection $h_0 = h$, at the end of the segments $B_k/\hat{B}_k$ there exists a bijection h_l, which extends h, such that $(\{y_k\}_{C_k}, t_k|_E) \sim_{h_l} (\{\hat{y}_k\}_{C_k}, \hat{t}_k|_E)$. We can then set $f' = h_l$ to obtain $t_k|_E \sim_{f'} \hat{t}_k|_E$ and $y_k \sim_{f'} \hat{y}_k$.

In order to conclude the proof, we have to show that f' is compatible with f. The bijection f' is an extension of h which is compatible with f. Hence, f' is also compatible with f.

□

Lemma F.3. *Let S be like in Theorem 2.5 and let E_1 and E_2 be two I-environments for S. Let ρ be the run of $E_1 \cdot S$ and $\hat{\rho}$ be the run of $E_2 \cdot S$. If there exists a bijection $f\colon \hat{R}_f \to R_f$, where $\hat{R}_f$ and R_f denote the sets of all references in $\{\hat{x}_1, \ldots, \hat{x}_k, \hat{y}_1, \ldots, \hat{y}_k\}$ and in $\{x_1, \ldots, x_k, y_1, \ldots, y_k\}$ respectively, such that*

$$x_1 \sim_f \hat{x}_1, \ldots, x_k \sim_f \hat{x}_k \quad \text{and} \quad y_1 \sim_f \hat{y}_1, \ldots, y_k \sim_f \hat{y}_k,$$

then, there exists a bijection $f'\colon \hat{R}_{f'} \to R_{f'}$, where $\hat{R}_{f'}$ and $R_{f'}$ denote the sets of references in the domains of the heaps of $\hat{s}_{k+1}|_S$ and $s_{k+1}|_S$ respectively, such that

$$s_{k+1}|_S \sim_{f'} \hat{s}_{k+1}|_S \quad \text{and} \quad x_{k+1} \sim_{f'} \hat{x}_{k+1},$$

where f' is compatible with f.

Proof. (Sketch) The proof of this Lemma is somewhat complementary to the proof of Lemma F.2: the segments $A_{k+1}/\hat{A}_{k+1}$ we are considering now are two sequences of n configurations of S pairwise equal, up to those references which have been exchanged with E_1 and E_2 so far and which are, by the hypothesis of the Lemma, under the bijection f.

By following the same inductive reasoning as in the proof of the Lemma F.2, there exist two bijections g and g' such that, respectively, g is defined as a restriction of f to the domain containing all references in $\{\hat{x}_1, \ldots, \hat{x}_{k+1}, \hat{y}_1, \ldots, \hat{y}_{k+1}\}$ and g' maps references in the domain of the heap of $\hat{s}_k|_S = \hat{t}_k|_S$ to references in the domain of the heap of $s_k|_S = t_k|_E$. Therefore, we can assert that the two initial configurations $(e_z, t_k)/(\hat{e}_z, \hat{t}_k)$ of the segments $A_{k+1}/\hat{A}_{k+1}$, pruned from the part of the state not accessible from S, are under a bijection h defined, as in the other lemma, piecewise by f and g'. (Bijections f and g' are compatible since all references in $\hat{R}_f \cap \hat{R}_{g'}$ are in the domain of g too; see the corresponding reasoning in the proof of Lemma F.2 for details.)

As applicability of no Jinja+ rule depends on the particular value of references, the same rule is applied in each step of A_{k+1} and $\hat{A}_{k+1}$. Therefore, by induction on the number of steps we can assert that, at the end of the two segments, there exists a bijection f', compatible with f, among their final (pruned) configurations, $(e_{z+n}[C_{k+1}.m_{k+1}(\vec{a}_{k+1})], s_{k+1}|_S)$ and $(\hat{e}_{z+n}[C_{k+1}.m_{k+1}(\vec{a}'_{k+1})], \hat{s}_{k+1}|_S)$, where $(C_{k+1}, m_{k+1}, \vec{a}_{k+1}) = x_{k+1}$ and $(C_{k+1}, m_{k+1}, \vec{a}'_{k+1}) = \hat{x}_{k+1}$, respectively. (Again, see the final part of the proof of Lemma F.2 for details.) □

We can now prove the main theorem of our extension. Thanks to the assumptions on the components of the run ρ, we remind that only primitives values and string references are exchanged between S and E.

Proof of Theorem 2.5. Implication from left to right is obvious. So let us assume that, I-noninterference *does not hold* for S. It means that there exists an I-environment E for S such that noninterference does not hold for $E \cdot S$, i.e., there are valid $\vec{a}_1$ and $\vec{a}_2$ such that the run ρ of $E \cdot S[\vec{a}_1]$ and the run $\hat{\rho}$ of $E \cdot S[\vec{a}_2]$ give different results.

We can define a sequence $\vec{u}$ such that the system $\tilde{E}_{\vec{u}}$ behaves exactly like E and it is therefore able to distinguish between $S[\vec{a}_1]$ and $S[\vec{a}_2]$. Since the value of `result` in a state s is part of $s|_E$, we conclude from Lemma F.2, that there exists an index k such that x_k^{ρ} and $x_k^{\hat{\rho}}$ are not under a bijection. Let k be the first such index. Note that, from Lemma F.2, we also know $x_i^{\rho} \sim_f x_i^{\hat{\rho}}$ and $y_i^{\rho} \sim_f y_i^{\hat{\rho}}$ for $i \in \{1, \ldots, k-1\}$ and for a bijection f. Let us assume that what breaks the bijection are the first arguments z^{ρ} and $z^{\hat{\rho}}$ in the calls described by x_k^{ρ} and $x_k^{\hat{\rho}}$, respectively (for the other cases the proof is very similar).

We can now define the sequence $\vec{u}$ which determines the behavior of $\tilde{E}_{\vec{u}}$ as a sequence containing only zeros with the following exceptions:

- For each primitive value v_i that E returns to $S[\vec{a}_1]$, let l be the index such that u_l is the element of $\vec{u}$ determining the i-th primitive value that $\tilde{E}_{\vec{u}}$ returns to $S[\vec{a}_1]$. We then set $u_l = v_i$.

- Each string reference r_i that E returns to $S[\vec{a}_1]$ can be either a reference already exchanged between E and $S[\vec{a}_1]$ before or a freshly exchanged one. In the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$, let p be the integer such that u_p decides whether to return a freshly created string or a string stored in `stringList` (see listing in Figure 2.5).

If r_i is a freshly exchanged reference, we set u_p to 1, u_{p+1} to the length of r_i and each u_j for $j \in \{p+2,\ldots,p+|r_i|+1\}$ to the integer corresponding to the $(j-p-2)$-th character of r_i. Otherwise, if r_i was already exchanged before the last segment of execution of E, let assume this reference is the j-th element of `stringList`. We set u_p to 2 and u_{p+j} to 1.

- The arguments z^{ρ} and $z^{\hat{\rho}}$ brake the bijection either because their values are different or because they point to different references in $s|_E$. In the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$, if z^{ρ} and $z^{\hat{\rho}}$ are primitive values, then let q be the integer such that u_q decides whether to test these arguments (they are then compared to u_{q+1} to determine the result). We set u_q to 1 and u_{q+1} to z^{ρ}.

  Contrariwise, if z^{ρ} and $z^{\hat{\rho}}$ are string references, we distinguish the following three cases:

  1) If the two string values have different lengths, let l be the integer such that u_l decides whether to test their lengths. We then set u_l to 1 and u_{l+1} to the length of z^{ρ}.
  2) Otherwise, if the two string values differ from at least one character, let p be the integer such that u_p decides whether to test the j-th character of z^{ρ}, the first one which differs from the j-th character of $z^{\hat{\rho}}$. We set u_p to 1 and u_{p+1} to the integer corresponding to the j-th character of z^{ρ}.
  3) Finally, if there exists (at least) an element of `stringList` whose reference comparison with z^{ρ} gives a different result to the reference comparison with $z^{\hat{\rho}}$, let q be the integer such that u_q decides whether to test the result of the comparison between $z^{\rho}/z^{\hat{\rho}}$ and the aforementioned element of `stringList`. We set both u_q and u_{q+1} to 1.

- As mentioned, for all remaining i we set $u_i = 0$.

In order to complete the proof, it is enough to show that $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$ give different results.

Let σ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and $\hat{\sigma}$ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$. Therefore, by the definition of $\vec{u}$, we have that there exist two bijections g and h such that $x_i^{\rho} \sim_g x_i^{\sigma}$, $y_i^{\rho} \sim_g y_i^{\sigma}$ and $x_i^{\hat{\rho}} \sim_h x_i^{\hat{\sigma}}$, $y_i^{\hat{\rho}} \sim_h y_i^{\hat{\sigma}}$ for $i \in \{1,\ldots,k-1\}$. Then, by Lemma F.3, there exist two bijections g' and h', compatible with g and h respectively, such that $x_k^{\rho} \sim_{g'} x_k^{\sigma}$ and $x_k^{\hat{\rho}} \sim_{h'} x_k^{\hat{\sigma}}$: In particular, $z^{\rho} = g'(z^{\sigma})$ and $z^{\hat{\rho}} = h'(z^{\hat{\sigma}})$. Since, by the construction of $\tilde{E}_{\vec{u}}$, we know that there not exists a bijection f such that $z^{\rho} = f(z^{\hat{\rho}})$, then there neither exists a bijection f' such that $z^{\sigma} = f'(z^{\hat{\sigma}})$. In fact, if the bijection f' existed, it should be defined as $f' = (g')^{-1} \circ f \circ h'$, i.e., the composition of the inverse of g' with (the nonexistent) f, composed with h'. We can then conclude that z^{σ} and $z^{\hat{\sigma}}$ are not under a bijection either. Therefore, by the definition of $\vec{u}$, the variable `result` is set to `true` in $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and to `false` in $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$, after which both systems terminate (the `abort` is executed immediately after the assignment). Hence, we show that these systems give different results which completes the proof. □

F.2. Proof of Equivalence Relation of $\equiv_{\text{comp}}$

The proof of the equivalence relation of $\equiv_{\text{comp}}$ (as defined in Definition 3.18) is completely trivial, except for the transitive relation. The intuition for the transitive relation is as follows: We

have three probability measures on the outcomes of three different SyncJinja+ systems such that both the difference between the first and the second and the difference between the second and the third are negligible, i.e., asymptotically close to zero. Then, the difference between the first and the third is negligible too.

We anyway provide the proof of the equivalence relation of $\equiv_{\mathsf{comp}}$ via the following three lemmas.

Lemma F.4 (Reflexivity of Computational Equivalence). *Let P_1 be a (complete, possibly probabilistic and multi-threaded) programs with security parameter η. Then, $P_1 \equiv_{\mathsf{comp}} P_1$, i.e., P_1 is computational equivalent to itself.*

Proof. We have $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}| = 0$. Then, according to the definition of a negligible function, for every $c > 0$, there exists $\eta_0 = 0$ such that $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}| = 0 \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$. □

Lemma F.5 (Symmetry of Computational Equivalence). *Let P_1 and P_2 be (complete, possibly probabilistic and multi-threaded) programs with security parameter η. If $P_1 \equiv_{\mathsf{comp}} P_2$, then $P_2 \equiv_{\mathsf{comp}} P_1$.*

Proof. Since $P_1 \equiv_{\mathsf{comp}} P_2$, according to the definition of a negligible function, we have that for every $c > 0$, there exists η_0 such that $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}| \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$. Since both $\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}$ and $\mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}$ are two probability measures and hence two real numbers in the interval $[0,1]$, we have

$$
\begin{aligned}
&|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}| \\
&\qquad = |\mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}|.
\end{aligned}
$$

Then for every $c > 0$, η_0 is also such that $|\mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}| \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$. Then, by Definition 3.18, $P_2 \equiv_{\mathsf{comp}} P_1$. □

Lemma F.6 (Transitivity of Computational Equivalence). *Let P_1, P_2, and P_3 be (complete, possibly probabilistic and multi-threaded) programs with security parameter η. If $P_1 \equiv_{\mathsf{comp}} P_2$ and $P_2 \equiv_{\mathsf{comp}} P_3$, then $P_1 \equiv_{\mathsf{comp}} P_3$.*

Proof. To simplify the notation, let us firstly define the probability measure on the outcome of a system P_i with security parameter η as follows:

$$
\mathsf{Prob}\{P_i(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} = x_i(\eta).
$$

As, by the premise of the lemma, we know $P_1 \equiv_{\mathsf{comp}} P_2$ and $P_2 \equiv_{\mathsf{comp}} P_3$, according to the definition of negligible functions, we have that for every $c > 0$, there exist η_0 and η_1 such that $|x_1(\eta) - x_2(\eta)| \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$ and $|x_2(\eta) - x_3(\eta)| \leq \frac{1}{\eta^c}$ for all $\eta > \eta_1$. To assert $P_1 \equiv_{\mathsf{comp}} P_3$, according to Definition 3.18, we have to show that for every $c > 0$, there exists η_2 such that $|x_1(\eta) - x_3(\eta)| \leq \frac{1}{\eta^c}$ for all $\eta > \eta_2$.

Let $c > 0$. We define $\hat{\eta} = \max(\eta_0, \eta_1)$. Since we have

$$|x_1(\eta) - x_3(\eta)| \leq |x_1(\eta) - x_2(\eta)| + |x_2(\eta) - x_3(\eta)| \leq \frac{2}{\eta^c} \text{ for all } \eta > \hat{\eta},$$

we can identify η_2 to be the largest natural number less than or equal to $\hat{\eta} \cdot \sqrt[c]{2}$.
It directly follows that for each $c > 0$ there exists η_2, where $\eta_2 = \lfloor \hat{\eta} \cdot \sqrt[c]{2} \rfloor$, such that

$$|x_1(\eta) - x_3(\eta)| \leq \frac{1}{\eta^c} \text{ for all } \eta > \eta_2.$$

□

F.3. Proof of Theorem 3.6

In order to prove the Theorem 3.6, we have to show that for each atomic (hence Jinja+) environmentally I-bounded system S, for each bounded SyncJinja+ I-environment E^{MT} for S, and for each bounded scheduler $\mathscr{S}$, there exists a bounded Jinja+ I-environment E^{ST}, such that

$$\{E^{MT} \cdot S\}_{\mathscr{S}} \equiv_{\mathsf{comp}} E^{ST} \cdot S. \tag{F.1}$$

Then, since by the premises of the theorem we know that for each single-threaded Jinja+ I-environment E^{ST} for S_1/S_2 we have

$$E^{ST} \cdot S_1 \equiv_{\mathsf{comp}} E^{ST} \cdot S_2, \tag{F.2}$$

by combining (F.1) and (F.2), by transitivity of the computational equivalence relation, we obtain

$$\begin{aligned} \{E^{MT} \cdot S_1\}_{\mathscr{S}} &\overset{(F.1)}{\equiv_{\mathsf{comp}}} E^{ST} \cdot S_1 \\ &\overset{(F.2)}{\equiv_{\mathsf{comp}}} E^{ST} \cdot S_2 \\ &\overset{(F.1)}{\equiv_{\mathsf{comp}}} \{E^{MT} \cdot S_2\}_{\mathscr{S}}. \end{aligned} \tag{F.3}$$

Given a multi-threaded (bounded) environment E^{MT} running under a scheduler $\mathscr{S}$, we now show the relation (F.1) by defining a single-threaded (bounded) environment E^{ST} which simulates the run of *both* E^{MT} and $\mathscr{S}$. We then represents the runs of S composed either with E^{MT} and $\mathscr{S}$ or with E^{ST} and, finally, we correlate two possible runs of these two (complete) programs to then assert that they output the same result with the same probability, up to some negligible function.

Let S be an atomic environmentally I-bounded system, E^{MT} a bounded SyncJinja+ I-environment, and $\mathscr{S}$ a bounded scheduler. Since the runs of S, E^{MT}, and $\mathscr{S}$ are all bounded by a polynomial p, we can fix the finite bit strings r_S, $r_{E^{MT}}$, and $r_{\mathscr{S}}$ so that S_{r_s}, $E^{MT}_{r_{MT}}$, and $\mathscr{S}_{r_{\mathscr{S}}}$ (to simplify the notation from now on we indicate the systems as S_r, E^{MT}_r, and $\mathscr{S}_r$, respectively) denote the deterministic systems obtained from S, E^{MT}, and $\mathscr{S}$, respectively, by fixing their randomness with r_S, $r_{E^{MT}}$, and $r_{\mathscr{S}}$ in the following way: The primitive randomBit() is replaced by a method (along with a new static field) declared within S_r, E^{MT}_r, and $\mathscr{S}_r$, respectively, such that

the first $|r_S|$, $|r_{E^{MT}}|$, and $|r_{\mathscr{S}}|$ bits are chosen according to r_s, $r_{E^{MT}}$, and $r_{\mathscr{S}}$, respectively; all the remaining bits returned by this method are 0. That is, the deterministic program $\{E_r^{MT} \cdot S_r\}_{\mathscr{S}_r}$ denotes exactly one run.

Since, by the premises of the theorem, S is considered to be an atomic system, in the run of $\{E_r^{MT} \cdot S_r\}_{\mathscr{S}_r}$ all the code of S is executed within the same thread, i.e., without interleaving of the run neither of any other thread spawned by the environment nor of the scheduler. Therefore, with abuse of notation, we can assert that the run of $\{E_r^{MT} \cdot S_r\}_{\mathscr{S}_r}$ is equivalent to the run of $\{E_r^{MT}\}_{\mathscr{S}_r} \cdot S_r$.

Simulating E^{MT} and $\mathscr{S}$. We now define a bounded Jinja+ I-environment E_r^{ST} whose composition with S_r reproduces the run of $\{E_r^{MT}\}_{\mathscr{S}_r} \cdot S_r$, for a finite bit string $r_{E^{ST}}$.

In what follows, we assume the method `main` defined in S. The case where `main` is defined in E^{MT} is similar. E^{ST} is defined in such a way that it also contains no `main` and it encodes the multi-threaded SyncJinja+ configuration $\langle \Pi, h, lock\rangle_{\langle e,(l,h)\rangle}$ of E^{MT} and $\mathscr{S}$ in its single-threaded configuration $\langle e,(l,h)\rangle$. At the beginning of the computation, the only available thread starts executing the code of S_r without interleaving of the scheduler. Whenever S_r calls an external method in the code of E_r^M, the same method of E_r^{ST} is also called though the interface I, since both E_r^{MT} and E_r^{ST} are defined to be I-environments for S. Then, for each reduction step taken, either by $\mathscr{S}$ or by a thread of E^{MT}, according to the multi-threaded semantics rules of SyncJinja+ (Rules B.88-B.95), the corresponding expression (either of the scheduler or of the corresponding thread) encoded in the configuration of E_r^{ST} is also reduced according to the single-threaded semantics of Jinja+ (Rules B.1-B.87). The multi-threaded state encoded in E^{ST} is then updated accordingly. That is, E_r^{ST} simulates each reduction step performed either by E_r^{MT} or by $\mathscr{S}$, by applying the *same* semantic rule at each step of the computation. The only single-threaded rules of SyncJinja+ which do not have any counterpart in Jinja+ are the rules which produce the thread actions (rules B.77-B.87, Figure B.7).

a) Whenever the language construct `start`(e) is to be reduced in a thread of E_r^{MT}, E_r^{ST} first reduces the expression e to the reference `addr` a. Then, another thread is added to the thread pool of the multi-threaded configuration encoded in the single-threaded configuration of E^{ST}: The initial expression of this new thread is a call of the (only) method $run()$ of the object of (a subclass of) the class $Thread$ pointed by the location `addr` a. In the caller thread, `start`(`addr` a) is reduced to *unit*, instead. The store l of the new thread is initialized as the store of the only thread present at the beginning of the computation, i.e., it takes only the static variables of the caller thread. The shared heap h and the lock map *lock* encoded in the single-threaded configuration of E^{ST} remain unchanged.

b) Whenever the language construct `sync`$(e_1)\{e_2\}$ is to be reduced in a thread of E_r^{MT}, E^{ST} first reduces the expression e_1 to the reference (location) `addr` a. Then, E^{ST} makes sure that this thread can acquire the lock on the object pointed by the location `addr` a. That is, E_r^{ST} checks, in the lock map *lock* encoded in its state, that no other thread has already acquired the lock on this object.

   If the thread cannot acquire the lock, this thread cannot progress, i.e., there are no rules of the multi-threaded semantics which can be applied and then, in the multi-threaded computation,

the scheduler does not schedule this thread until the lock on this object is released by the thread which was holding it. Since E_r^{ST} is defined in such a way that each step of the multi-threaded computation of E_r^{MT} and $\mathscr{S}_r$ is simulated by the computation of E_r^{ST}, the corresponding thread in the multi-threaded configuration encoded in E_r^{ST} is also not scheduled until the lock is released.

Otherwise, if the thread can acquire the lock, the lock map *lock* is updated either by adding the entry with the identifier of this thread (in case the thread is acquiring the lock for the first time) or by incrementing the number of lock acquisitions by this thread on the entry related to this reference (in case the thread has already acquired this lock). Once E_r^{ST} updates the lock map *lock* encoded in its state, it reduces the expression `sync`(`addr` a)$\{e\}$ to `insync`(`addr` a)$\{e\}$.

c) Whenever the language construct `insync`(`addr` a)$\{e\}$ is to be reduced in a thread of E_r^{MT}, E_r^{ST} keeps on reducing the expression e to either `Val` v or `Throw` a'. Then, it removes the lock on the object pointed by `addr` a from the lock map *lock* encoded in its state by decreasing the number of lock acquisitions on the entry related to this reference; in case the number of lock acquisitions is zero, it removes this entry.

In the simulation of the run of E^{MT} under $\mathscr{S}$, E^{ST} maintains the value of its variable `result` to the value of the variable `result` encoded in the multi-threaded state of E^{MT}.

We now define the bit string $r_{E^{ST}}$. The length of this (finite) string is such that $|r_{E^{ST}}| = |r_{E^{MT}}| + |r_{\mathscr{S}}|$. Whenever in the run of $\{E_r^{MT}\}_{\mathscr{S}_r}$ the i-th random bit is chosen according to the j-th bit of $r_{E^{MT}}$, the i-th element of the bit string $r_{E^{ST}}$ is set to the same value: $r_{E^{ST}}[i] := r_{E^{MT}}[j]$. Instead, whenever in the run of $\{E_r^{MT}\}_{\mathscr{S}_r}$ the i-th random bit is chosen according to the k-th bit of $r_{\mathscr{S}}$, the i-th element of the bit string $r_{E^{ST}}$ is set to the same value: $r_{E^{ST}}[i] := r_{\mathscr{S}}[k]$.

Then, since $r_{E^{ST}}$ is a finite bit string and since E^{ST} simulates the same number of steps performed by E^{MT} and $\mathscr{S}$ (which are both bounded systems), we can assert that E_r^{ST} is bounded too.

To show the result relating the (deterministic) runs of $\{E_r^{MT} \cdot S_r\}_{\mathscr{S}_r}$ and $E_r^{ST} \cdot S_r$, we first need to introduce some notation concerning the runs and the states of the two programs.

Representing runs. We can split the (deterministic) run ρ of $\{E_r^{MT} \cdot S_r\}_{\mathscr{S}_r}$ into segments:

$$\rho = A_1^{\lambda_1}[x_1, s_1] B_1^{\lambda_2}[y_1, t_1, u_1] \ldots B_{m-1}^{\lambda_{2m-2}}[x_{m-1}, s_{m-1}, u_{m-1}] \\ A_m^{\lambda_{2m-1}}[x_m, s_m] B_m^{\lambda_{2m}},$$

where m is the number of method calls in the code of S to methods defined in E^{MT}, such that:

– Every $A_i^{\lambda_j}$ is a sequence of configurations (sub-run) where *only* the code of S is executed. That is, since S is atomic, its run is neither interleaved by the run of the scheduler nor of by the run of any thread spawned by E^{MT}. Every $A_i^{\lambda_j}$ ends with a configuration of the form $q_k = \langle \Pi_k \cup \{tID_0 \mapsto \langle e_i[C_i.m_i(\vec{a}_i)], l_i \rangle\}, h_k, lock_k \rangle$ where C_i defined in E and $C_i.m_i(\vec{a}_i)$ is the subexpression which is about to be rewritten. We denote the tuple $(C_i, m_i, \vec{a}_i)$ by x_i and the state $(L_k, h_k, lock_k)$ by s_i, where l_{tID_0} in L_k is such that $l_{tID_0} = l_i$.

– Every $B_i^{\lambda_j}$ is a sequence of configurations (sub-run) where the code of E^{MT} and of the scheduler $\mathscr{S}$ is executed. In each $B_i^{\lambda_j}$ block, the initial configuration of the thread tID_0 which starts executing the code of E^{MT} is of the form $\langle e_i[\{e'_i\}_{C_i}], l_i\rangle$, where $\{e'_i\}_{C_i}$ is the block obtained by the static method call rule applied to $C_i.m_i(\vec{a}_i)$ (it depends only on C_i, m_i, and $\vec{a}_i$). Every $B_i^{\lambda_j}$, except for the last one, ends with a configuration $q_n = \langle \Pi_n \cup \{tID_0 \mapsto \langle e_i[\{y_i\}_{C_i}], l_j\rangle\}, h_n, lock_n\rangle$ where y_i is a value (the value returned by the method denoted by x_i). We denote the state $(L_n, h_n, lock_n)$ by t_i, where l_{tID_0} in L_n is such that $l_{tID_0} = l_j$. Moreover, since in each B_i block also the code of the scheduler is executed, we denote its (single-threaded) state at the and the block by $u_i = (l_{\mathscr{S},i}, h_{\mathscr{S},i})$.

– For each $j \in \{1, \ldots, 2m\}$, every λ_j indicates the number of steps executed in the run ρ before the beginning of the segment $A^{\lambda_j}_{\lfloor j/2 \rfloor}$ or $B^{\lambda_j}_{\lfloor j/2 \rfloor}$.

Similarly, we represent the (deterministic) run $\tilde{\rho}$ of the system $E^{ST} \cdot S$ as

$$\tilde{\rho} = \tilde{A}_1^{\tilde{\lambda}_1}[\tilde{x}_1, \tilde{s}_1]\tilde{B}_1^{\tilde{\lambda}_1}[\tilde{y}_1, \tilde{t}_1] \ldots \tilde{B}_{m-1}^{\tilde{\lambda}_{2m-2}}[\tilde{x}_{m-1}, \tilde{s}_{m-1}]\tilde{A}_m^{\tilde{\lambda}_{2m-1}}[\tilde{x}_m, \tilde{s}_m]\tilde{B}_m^{\tilde{\lambda}_{2m}},$$

where m is the (same) number of method calls in the code of S to methods defined in E^{ST}, such that:

– Every $\tilde{A}_i^{\tilde{\lambda}_j}$ is a sequence of configurations where the code of S is executed. Every $\tilde{A}_i^{\tilde{\lambda}_j}$ ends with a configuration of the form $\tilde{q}_k = \langle \tilde{e}_i[C_i.m_i(\vec{a}_i)], (\tilde{l}_i, \tilde{h}_i)\rangle$ where C_i defined in S and $C_i.m_i(\vec{a}_i)$ is the subexpression which is about to be rewritten. We denote the tuple $(C_i, m_i, \vec{a}_i)$ by $\tilde{x}_i$ and the state $(\tilde{l}_i, \tilde{h}_i)$ by $\tilde{s}_i$.

– Every $\tilde{B}_i^{\tilde{\lambda}_j}$ is a sequence of configurations where the code of E^{ST} is executed. In each $\tilde{B}_i^{\tilde{\lambda}_j}$ block, the initial configuration is of the form $\langle \tilde{e}_i[\{e'_i\}_{C_i}], (\tilde{l}_i, \tilde{h}_i)\rangle$, where $\{e'_i\}_{C_i}$ is the block obtained by the static method call rule applied to $C_i.m_i(\vec{a}_i)$ (it depends only on C_i, m_i, and $\vec{a}_i$). Every $\tilde{B}_i^{\tilde{\lambda}_j}$, except for the last one, ends with a configuration $q_n = \langle \tilde{e}_i[\{\tilde{y}_i\}_{C_i}], (\tilde{l}_j, \tilde{h}_j)\rangle$ where $\tilde{y}_i$ is a value (the value returned by this method). We denote the state of $(\tilde{l}_j, \tilde{h}_j)$ by $\tilde{t}_i$.

– For each $j \in \{1, \ldots, 2m\}$, every λ_j indicates the number of steps executed in the run ρ before the beginning of the segment $\tilde{A}^{\tilde{\lambda}_j}_{\lfloor j/2 \rfloor}$ or $\tilde{B}^{\tilde{\lambda}_j}_{\lfloor j/2 \rfloor}$.

Let f be a bijection from references of $E^{ST} \cdot S$ to references of $\{E^{MT} \cdot S\}_{\mathscr{S}}$. We say that $\tilde{x}_i$ matches x_i w.r.t. f, written $\tilde{x}_i \sim x_i$, if, for each variable v_j in x_i, there exists a variable $\tilde{v}_j$ in $\tilde{x}_i$ such that

- $\tilde{v}_j = v_j$, if $\tilde{v}_j$ and v_j are primitive values;
- $\tilde{v}_j = f(v_j)$, if $\tilde{v}_j$ and v_j are references.

In a similar way, we can say $\tilde{y}_i \sim y_i$.

Representing states. Let $s = (L, h, lock)$ be a multi-threaded SyncJinja+ state that occurs in $E_r^{MT} \cdot S_r$, where $L = \{l_{tID_1} \ldots, l_{tID_n}\}$. By $s|_{E^{MT}} = (L|_{E^{MT}}, h|_{E^{MT}}, lock|_{E^{MT}})$ we denote the part of

the state that is accessible from E^{MT} through the variables that E^{MT} uses. For each store l_{tID_i} in $L_{E^{MT}}$, we leave in the domain of $l_{tID_i}|_{E^{MT}}$ only the variables of l_{tID_i} that E^{MT} can access; if $l_{tID_i}|_{E^{MT}} = \emptyset$, we remove l_{tID_i} from $L|_{E^{MT}}$.

In the domains of $h|_{E^{MT}}$ and $lock|_{E^{MT}}$ we leave only those references that can be reached from the variables in the stores in $L|_{E^{MT}}$, where, for each $l_{tID_i}|_{E^{MT}}$ in $L|_{E^{MT}}$, a reference can be reached from $l_{tID_i}|_{E^{MT}}$ if one of the following holds:

- it is stored in one of the variables of $l_{tID_i}|_{E^{MT}}$,
- it is stored in a field of an object that can be reached from $l_{tID_i}|_{E^{MT}}$.

In an analogous way, we define $s|_S = (L|_S, h|_S, lock|_S)$. In particular, since in $E^{MT} \cdot S$ the only thread which runs the code of S is tID_0, $L|_S = \{l_{tID_0}|_S\}$. Moreover, since S is atomic and hence its code contains no language constructs introduced in SyncJinja+ (`start`, `sync`, and `insync`), we have $lock|_s = \emptyset$ and $lock|_{E^{MT}} = lock$.

Let $\tilde{s} = (\tilde{l}, \tilde{h})$ be a single-threaded Jinja+ state that occurs in $E^{ST} \cdot S$. As for the multi-threaded state, by $\tilde{s}|_{E^{ST}} = (\tilde{h}|_{E^{ST}}, \tilde{l}|_{E^{ST}})$ we denote the part of the state accessible from E^{ST}, whereas by $\tilde{s}|_S = (\tilde{l}|_S, \tilde{h}|_S)$ the part of the state accessible from S.

Assuming there are no clashes neither between variables' names in the stores inside $L|_{E^{MT}}$ nor between these stores and the store $l_{\mathscr{S}}$ of u, we say that the single-threaded state $\tilde{s}|_{E^{ST}}$ *represents* both the multi-threaded state $s|_{E^{MT}}$ and the single-threaded state u of the scheduler, written $\tilde{s}|_{E_r^{ST}} \models s|_{E_r^{MT}} \cup u$, if there exists a bijection f from references of E^{MT} and of $\mathscr{S}$ to references E^{ST} such that:

– The values of both the variables in $L|_{E^{MT}} := \{l_{tID_0}|_{E^{MT}}, \ldots, l_{tIDn}|_{E^{MT}}\}$ and in $l_{\mathscr{S}}$ are—up to mapping f—the same as the value of the corresponding variable in $\tilde{l}|_{E^{ST}}$. (We notice that $\tilde{l}|_{E^{ST}}$ always contains more variables than $l|_{E^{MT}}$, e.g., the variables necessary to encode the whole multi-threaded configuration.)

– The references in $h|_{E^{MT}}$ and $h_{\mathscr{S}}$ point—up to mapping f—to the same objects of the corresponding references in $\tilde{h}|_{E^{ST}}$. (As in the previous item, $\tilde{h}|_{E^{ST}}$ always contains more references than $h|_{E^{MT}}$.)

– There exists a data structure in $\tilde{s}|_{E^{ST}}$ (e.g., a list) which, for each $a \in \mathtt{dom}(lock|_{E^{MT}})$, contains a reference $\tilde{a}$ such that $\tilde{a} = f(a)$ and which records the thread identifier that has acquired the lock on the object pointed by $\tilde{a}$, along with the number of lock acquisitions.

Finally, let $\tilde{s}|_S = (\tilde{l}|_S, \tilde{h}|_S)$ and $s|_S = (l_{tID_0}|_S, h|_S)$ being the state accessible from S of the programs $E^{ST} \cdot S$ and $\{E^{MT} \cdot S\}_{\mathscr{S}}$, respectively. Let f be a bijection f from references of $E^{ST} \cdot S$ to references of $\{E^{MT} \cdot S\}_{\mathscr{S}}$. We say that $\tilde{s}|_S$ matches $s|_S$ w.r.t. f, written $\tilde{s}|_S \sim s|_S$, if

- for each variable v_i in $l_{tID_0}|_S$, there exists a variable $\tilde{v}_i$ in $\tilde{l}|_S$ such that
  - $\tilde{v}_i = v_i$, if $\tilde{v}_i$ and v_i are primitive values,
  - $\tilde{v}_i = f(v_i)$, if $\tilde{v}_i$ and v_i are references;
- for each reference a_i in $h|_S$, there exists a reference $\tilde{a}_i$ in $\tilde{h}|_S$ such that $a_i = f(\tilde{a}_i)$.

Relation between SyncJinja+ and Jinja+ runs. We now state two lemmas, namely Lemma F.8 and Lemma F.7 which will allow us to relate the SyncJinja+ run ρ of $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and the Jinja+ run $\tilde{\rho}$ of $E^{ST} \cdot S$.

Lemma F.7. *Let ρ and $\tilde{\rho}$ be the runs of $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and $E^{ST} \cdot S$ respectively, as defined above. Let $B_i^{\lambda_j}$ and $\tilde{B}_i^{\tilde{\lambda}_j}$ be two segments where the code of $\{E^{MT}\}_{\mathscr{S}}$ and E^{ST} are executed, respectively. Then, for each SyncJinja+ configuration q_k of the run ρ such that $k \in \{\lambda_j, \dots, \lambda_{j+1}\}$, there exists a sequence of Jinja+ configurations $\tilde{q}_o, \dots, \tilde{q}_{o+\xi}$ (where the number ξ is fixed for each SyncJinja+ reduction rule whose application led to q_k) in $\tilde{\rho}$ such that:*

1) $o, o+\xi \in \{\tilde{\lambda}_j, \dots, \tilde{\lambda}_{j+1}\}$,

2) $(\tilde{l}_{o+\xi}|_{E^{ST}}, \tilde{h}_{o+\xi}|_{E^{ST}}) \models (L_k|_{E^{MT}}, h_k|_{E^{MT}}, lock_k) \cup (l_{\mathscr{S},k}, h_{\mathscr{S},k})$.

Lemma F.8. *Let ρ and $\tilde{\rho}$ be the runs of $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and $E^{ST} \cdot S$ respectively, as defined above. Let $\tilde{A}_i^{\tilde{\lambda}_j}/A_i^{\lambda_j}$ be two segments where the code of S is executed. Then, the steps executed in $\tilde{A}_i^{\tilde{\lambda}_j}$ and $A_i^{\lambda_j}$ are $\tilde{\lambda}_{j+1} - \tilde{\lambda}_j = \lambda_{j+1} - \lambda_j$ in both segments and for each $k \in \{\tilde{\lambda}_j, \dots, \tilde{\lambda}_{j+1}\}$ and for each $o \in \{\lambda_j, \dots, \lambda_{j+1}\}$, we have that the Jinja+ configuration $\tilde{q}_k$ and the SyncJinja+ configuration q_o are such that:*

1) $\tilde{l}_k|_S \sim l_{tID_0,o}|_S$,

2) $\tilde{h}_k|_S \sim h_o|_S$.

These two lemmas, which, similarly to the Lemmas F.2 and F.3 above, can be proven by induction on the number of steps of execution, allow us to prove the following result which relates the SyncJinja+ run ρ of $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and the Jinja+ run $\tilde{\rho}$ of $E^{ST} \cdot S$.

Lemma F.9. *Let ρ and $\tilde{\rho}$ be the runs of $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and $E^{ST} \cdot S$ respectively, as defined above. Then, for each $i \in \{1, \dots, m\}$ the following holds:*

(a) $\tilde{s}_i|_S \sim s_i|_S$,

(b) $\tilde{t}_i|_{E^{ST}} \models t_i|_{E^{MT}} \cup u_i$,

(c) $\tilde{x}_i \sim x_i$ and $\tilde{y}_i \sim y_i$.

Proof. (Sketch) Item (a) states that, at the end of each segment $\tilde{A}_i/A_i$ (the segments in $\tilde{\rho}/\rho$ where the code of S is executed), the part of the state accessible from S is the same in both runs $\tilde{\rho}/\rho$. That is, the part of the single-threaded state $\tilde{s}|_S = (\tilde{l}_S, \tilde{h}_S)$ accessible from S in the Jinja+ run $\tilde{\rho}$ is equivalent to the multi-threaded state $s|_S = \{L|_S, h|_S, lock|_S\}$ accessible from S in the SyncJinja+ run ρ.

Since in the SyncJinja+ run ρ S is executed atomically and since, by Definition 3.4, the (only) thread spawned at the beginning of the run ρ, namely tID_0, contains, at the beginning of ρ, the SyncJinja+ expression e_0 referencing to the method `main` of S, we have that $lock|_S = \emptyset$ and that $L|_S = \{l_{tID_0}|_S\}$. Moreover, by Lemma F.8, in each segment $\tilde{A}_i^{\tilde{\lambda}_j}/A_i^{\lambda_j}$ where the code of S is

executed, for each $k \in \{\tilde{\lambda}_j, \dots, \tilde{\lambda}_{j+1}\}$ and for each $o \in \{\lambda_j, \dots, \lambda_{j+1}\}$, we have that $\tilde{l}_k|_S \sim l_{tID_0,o}|_S$ and $\tilde{h}_k|_S \sim h_o|_S$. We can then assert that, at the end of the two segments $\tilde{A}_i^{\tilde{\lambda}_j}/A_i^{\lambda_j}$, we have

$$s_i|_S = \{L_i|_S, h_i|_S, lock_i|_S\} = \{\{l_{tID_0,i}|_S\}, h_i|_S, \emptyset\} = (l_{tID_0,i}|_S, h_i|_S) \sim (\tilde{l}_i|_S, \tilde{h}_i|_S) = \tilde{s}_i|_S.$$

Item (b) states that the Jinja+ system E^{ST} correctly simulates the run of the SyncJinja+ system E^{MT} and of the scheduler $\mathscr{S}$. By Lemma F.7, at the end of each segment $\tilde{B}_i^{\tilde{\lambda}_i}/B_i^{\lambda_i}$, the part of the single-threaded state accessible from E^{ST} represents both the part of the multi-threaded state accessible from E^{MT} and the state of the scheduler.

Item (c) states that in the Jinja+ run $\tilde{\rho}$, E^{ST} and S exchange exactly the same data exchanged by E^{MT} and S in the SyncJinja+ run ρ:

- $\tilde{x}_i \sim x_i$ holds since it represents a method call performed by the code of S in both $\tilde{\rho}$ and ρ. By Lemma F.8, at the end of each segment $\tilde{A}_i^{\tilde{\lambda}_j}/A_i^{\lambda_j}$, there exists a bijection among $\tilde{s}|_S$ and $s|_S$. As the parameters of the method call of $\tilde{x}_i$ and x_i are part of $\tilde{s}|_S$ and $s|_S$, respectively, the same bijection can be used to assert $\tilde{x}_i \sim x_i$.
- $\tilde{y}_i \sim y_i$ holds since E^{ST} encodes the multi-threaded configuration of $\{E^{MT}\}_{\mathscr{S}}$ and simulates each reduction step executed either by a thread of E^{MT} or by the scheduler $\mathscr{S}$. By Lemma F.7, at the end of each segment $\tilde{B}_i^{\tilde{\lambda}_j}/B_i^{\lambda_j}$, the part of the single-threaded state accessible from E^{ST} represents both the part of the multi-threaded state accessible from E^{MT} and the state of the scheduler: $\tilde{t}_i|_{E^{ST}} \models t_i|_{E^{MT}} \cup u_i$. Since, by the definition of $\models$, for each variable v in $t_i|_{E^{MT}}$, there exists a variable $\tilde{v}$ in $\tilde{t}_i|_{E^{ST}}$ such that $\tilde{v} \sim v$, and since $\tilde{y}_i \in \tilde{t}_i|_{E^{ST}}$ and $y_i \in t_i|_{E^{MT}}$, we can conclude that $\tilde{y}_i \sim y_i$.

□

We can now observe that a direct consequence of Lemma F.9 (more precisely, of the fact that $\tilde{t}_i|_{E^{ST}}$ represents $t_i|_{E^{MT}}$, Item (b), and that, from the point of view of S, E^{ST} simulates the execution of E^{MT} under $\mathscr{S}$, Item (c)) is that the final value of variable `result` in ρ and $\tilde{\rho}$ is the same and, therefore, these two runs output the same result.

As all four systems E^{MT}, $\mathscr{S}$, S, and E^{ST} are bounded, there exists a polynomial p such that the probability that the length of the longest run (parameterized with integer size $intsize(\eta)$), among the runs of $\{E^{MT} \cdot S(\eta)\}_{\mathscr{S}}$ and of $E^{ST} \cdot S(\eta)$, exceeds $p(\eta)$ is negligible. So in all runs, except for a negligible fraction, at most $p(\eta)$ random bits are needed. Moreover, for almost all bit strings $r_{E^{MT}}$, $r_{\mathscr{S}}$, r_S, and $r_{E^{ST}}$ of length at most $p(\eta)$ and for all integers of size $intsize(\eta)$, we have that the runs of $\{E_r^{MT} \cdot S_r(\eta)\}_{\mathscr{S}_r}$ and $E_r^{ST} \cdot S_r(\eta)$ terminate for integer size $intsize(\eta)$. Now, since for all random bit strings $r_{E^{MT}}$, $r_{\mathscr{S}}$, r_S, and $r_{E^{ST}}$ and for all security parameters η, both $\{E_r^{MT} \cdot S_r(\eta)\}_{\mathscr{S}_r}$ and $E_r^{ST} \cdot S_r(\eta)$ output the same result, we can conclude that the programs $\{E^{MT} \cdot S\}_{\mathscr{S}}$ and $E^{ST} \cdot S$ output `true` with the same probability, up to a negligible function.

F.4. Proof of Theorem 4.1

As we have mentioned in Section 4.1.4, the proof of Theorem 4.1 is highly modular and leverages such properties of the realization relation stated in Section 2.2 as the composition theorem,

reflexivity, and transitivity. Due to this modular proof technique, we can even make use of the result proven in [KTG12a] for the public-key functionality without corruption and without a PKI.

First, we observer that the ideal functionality Ideal-PKIEnc can be split in the following way:

$$\text{Ideal-PKIEnc} = \text{Ideal-PKEnc} \cdot \text{Ideal-RegEnc} \ ,$$

where Ideal-PKEnc and Ideal-RegEnc are defined as follows:

Ideal-PKEnc consists of the classes `Encryptor` and `Decryptor` of Ideal-PKIEnc, as introduced in Section 4.1.2. Let I_{PKEnc} denote the public interface of these classes. That is, I_{PKEnc} coincides with I_{PKIEnc}, as defined in Section 4.1.1, excluding the interface of the class `RegisterEnc`.

Ideal-RegEnc consists of the class `RegisterEnc` of Ideal-PKIEnc. Let I_{EncPKI} denotes the public interface of this class. That is, the interface I_{PKIEnc}, as defined in Section 4.1.1, restricted to the public interface of the class `RegisterEnc`.

Similarly, Real-PKIEnc can be split in the following way:

$$\text{Real-PKIEnc} = \text{Real-PKEnc} \cdot \text{Real-RegEnc} \ ,$$

where Real-PKEnc and Real-RegEnc are defined as follows:

Real-PKEnc consists of the classes `Encryptor` and `Decryptor` of Real-PKIEnc, as introduced in Section 4.1.3. Note that the public interface of Real-PKEnc is I_{PKIEnc}, and hence, it is the same as the one for Ideal-PKEnc.

Real-RegEnc consists of the `RegisterEnc` of Real-PKIEnc. Note that the public interface of Real-RegEnc is I_{EncPKI}, and hence, it is the same as the one for Ideal-RegEnc.

Now, we prove the following sequence of realization relations:

$$\begin{aligned}
\text{Real-PKIEnc} \cdot \text{Real-PKI} = {} & \text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Real-PKI} \\
\leq^{I_{PKIEnc}} {} & \text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI} && \text{(F.4)} \\
\leq^{I_{PKIEnc}} {} & \text{Real-PKEnc} \cdot \text{Ideal-RegEnc} && \text{(F.5)} \\
\leq^{I_{PKIEnc}} {} & \text{Ideal-PKEnc} \cdot \text{Ideal-RegEnc} = \text{Ideal-PKIEnc} && \text{(F.6)}
\end{aligned}$$

From this, by transitivity of the realization relation, Theorem 4.1 follows. Now, we establish each of the above relations.

Lemma F.10. *Relation* (F.4) *holds true, that is*

$$\text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Real-PKI} \leq^{I_{PKIEnc}} \text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI} \ .$$

Proof. This relation easily follows from the assumption that Real-PKI $\leq^{I_{PKI}}$ Ideal-PKI, the composition theorem, and reflexivity of the realization relation. Indeed, by reflexivity of realization relation, we have that

$$\text{Real-PKEnc} \cdot \text{Real-RegEnc} \leq^{I_{PKIEnc}} \text{Real-PKEnc} \cdot \text{Real-RegEnc}.$$

Note that $I_{PKIEnc} = I_{PKEnc} \cup I_{EncPKI}$. Together with Real-PKI $\leq^{I_{PKI}}$ Ideal-PKI, by the composition theorem we immediately obtain that

$$\text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Real-PKI} \leq^{I_{PKIEnc} \cup I_{PKI}} \text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI}$$

which implies (F.4), since if a relation holds for one interface, then also for all of its subinterfaces. □

Lemma F.11. *The relation* (F.5) *holds true, that is*

$$\text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI} \leq^{I_{PKIEnc}} \text{Real-PKEnc} \cdot \text{Ideal-RegEnc}.$$

The two systems are very similar: the main difference is that in the ideal system (the one on the right hand-side) encryptors are stored directly (in a collection of registered encryptors), while in the real system (the one on the left hand-side) public keys are stored instead with wrapping/unwrapping of public keys in encryptors when necessary. Therefore, the relation holds true even if unbounded environments try to distinguish the two systems. The proof of this lemma is given in Appendix F.4.1.

Finally, relation (F.6) follows immediately by the following fact and again using the composition theorem and reflexivity of realization relation, similarly to the proof of Lemma F.10.

Lemma F.12. Real-PKEnc $\leq^{I_{PKEnc}}$ Ideal-PKEnc

We prove Lemma F.12 by reducing it to the result from [KTG12a], where similar functionalities, but without corruption are considered. In the proof we use the fact that corrupted encryptors can be simulated directly by an environment. The proof is given in Appendix F.4.2.

F.4.1. Proof of Lemma F.11

In this section, we prove that

$$\text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI} \leq^{I_{PKIEnc}} \text{Real-PKEnc} \cdot \text{Ideal-RegEnc}.$$

In order to do this, we need to show that there exists a simulator *Sim* such that

$$S = \text{Real-PKEnc} \cdot \text{Real-RegEnc} \cdot \text{Ideal-PKI} \ \approx^{I_{PKIEnc}}_{\text{comp}} \ Sim \cdot \text{Real-PKEnc} \cdot \text{Ideal-RegEnc} = \tilde{S}$$

Let *Sim* be the simple forwarding simulator that translates calls to the simulator interface of Ideal-RegEnc (that is class `RegisterEncSim`) into calls to the simulator interface of Ideal-PKI (class `PKISim`):

```
public class RegisterEncSim {
  public static boolean register(int id, byte[] domain, byte[] publicKey) {
    return PKISim.register(id, domain, publicKey);
  }
  public static boolean getEncryptor(int id, byte[] domain) {
    return PKISim.getKey(id, domain);
  }
}
```

We are able to prove the stronger property

$$S \approx^{I_{PKIEnc}}_{\mathsf{perf}} \tilde{S}$$

that is, the two systems are perfectly indistinguishable, that is indistinguishable by an unbounded, deterministic adversary (see [KTG12a] for details). For this, let us take an arbitrary deterministic I_{PKIEnc}-environment E for S (and hence for $\tilde{S}$). To complete the proof, it remains to show that

$$E \cdot S \equiv_{\mathsf{perf}} E \cdot \tilde{S} \tag{F.7}$$

which simply means that the environment E in the runs of both $E \cdot S$ and $E \cdot \tilde{S}$ outputs the same value.

On the intuitive level the above statement is quite straightforward. Indeed, the systems S and $\tilde{S}$, from the point of view of any environment E, realize very similar computations with the only difference being how they implement registration of encryptors. In the system S, public keys are kept in a collection (along with user identifiers); those keys are retrieved from an encryptor by method `registerEncryptor` and, conversely, wrapped into a newly created encryptor by method `getEncryptor`. In the system $\tilde{S}$, on the other hand side, encryptors are stored directly (along with user identifiers). Method `registerEncryptor` simply adds such an encryptor (along with a user identifier) into a collection, when method `getEncryptor` retrieves an appropriate encryptor and returns a newly created copy of it. These computations produce the same result, up to the internal state of the component $S/\tilde{S}$. In other words, the state of the computations in the considered systems is the same from the point of view of the environment and so, in particular, the value of variable `result` (which is determined by the environment) at the end of the runs is the same in both systems.

To formalize the intuitive argument given above, we need to introduce some notation.

Structure of states in a run. A configuration q of Jinja+, as defined in [KTG12a], is of the form (e, s), where e is a Jinja+ expression and s is a state. A *state* is a pair (h, l) of a heap and a store. A *heap* is a mapping from references (addresses) to object instances and a *store* is a mapping from variable names to values. A value can be either a reference or a value of a primitive type.

One particular type of expression is a *block expression* of the form $\{V : T;\ e\}_C$ or $\{V : T;\ V := \texttt{Val}\ v;\ e\}_C$, where V is a local variable (whose scope is this block) of type T and, in the second variant, with value $\texttt{Val}\ v$, e is an expression (e can access the local variable V), and C is a class name (denoting that the block originates from the code of the class C).

In general, an expression can contain many blocks as its subexpression. However, when we study expressions that occur in actual runs, it turns out that they have a simpler form, where all blocks are located on one path. Formally, let

$$q_0 = \langle e_0, \langle h_0, l_0 \rangle \rangle \xrightarrow{\ell} \langle e_1, \langle h_1, l_1 \rangle \rangle \xrightarrow{\ell} \cdots$$

be a run with the initial state $s_0 = \langle h_0, l_0 \rangle$. By the definition of the initial state [KTG12a, KTG12b], h_0 is empty and l_0 bounds the static variables of the program to their initial values (and no other variables). By inspecting the rules of Jinja+ (see Appendix B.1), one can see that, for every $i = 0, 1, \ldots,$

– l_i bound only static variables,

– for every subexpression e of e_i (including e_i) either e contains no block as its subexpression or e is of the form $E[b]$, where E contains no block and b is a block. That is, e can contain, directly, at most one block (although b can contain other blocks).

The definitions and results given below assume that expressions originate from runs of Jinja+ systems and therefore they are of the above form.

By $\mathfrak{C}[\cdot]$ we denote an expression context (that is, an expression with a hole) and by $\mathfrak{C}[e]$ we denote the expression obtained by replacing the hole by e.

We can now state the two following lemmas.

Lemma F.13. *$\langle e,(h,l)\rangle \xrightarrow{-} \langle e',(h',l')\rangle$ if and only if e contains no block.*

The proof can be easily done by structural induction, considering all possible rules of Jinja+ (Appendix B.1) that produce a reduction step with label $-$.

Lemma F.14. *If $\langle e,(h,l)\rangle \xrightarrow{D} \langle e',(h',l')\rangle$, then*

- *e is of the form $\mathfrak{C}[e_0]$, where e_0 is a block expression of class D without blocks;*
- $\langle e_0,(h,l)\rangle \xrightarrow{D} \langle e_0',(h',l')\rangle$*;*
- $e' = \mathfrak{C}[e_0']$.

Again, this lemma can be easily proven by structural induction.

Pruning. Let $\mathscr{C}$ be a set of classes (intuitively, representing a subprogram) and e be an expression. We define a *pruning* operator $\mathrm{sub}_{\mathscr{C}}(e)$ in such a way that it removes from e all those parts that come from classes not in $\mathscr{C}$ and only leaves the code originating in $\mathscr{C}$. Formally, we define $\mathrm{sub}_{\mathscr{C}}(e)$ as follows:

– if e contains no block, then $\mathrm{sub}_{\mathscr{C}}(e) = e$,

– if e is not a block, but contains one, that is $e = E[b]$, then $\mathrm{sub}_{\mathscr{C}}(e) = E[\mathrm{sub}_{\mathscr{C}}(b)]$,

– if $e = \{V : T;\ e'\}_D$ with $D \in \mathscr{C}$, then $\mathrm{sub}_{\mathscr{C}}(e) = \{V : T;\ \mathrm{sub}_{\mathscr{C}}(e')\}_D$ (and similarly for $e = \{V : T;\ V := \texttt{Val}\ v;\ e'\}_D$),

– if $e = \{V : T;\ e'\}_D$ with $D \notin \mathscr{C}$, and e' contains no blocks, then $\mathrm{sub}_{\mathscr{C}}(e) = \bot$.

– if $e = \{V : T;\ E[b]\}_D$, where b is a block with $D \notin \mathscr{C}$, then $\mathrm{sub}_{\mathscr{C}}(e) = \mathrm{sub}(b)$,

Corresponding states. As it has already been stated, our goal is to show that (F.7) holds true. The systems S and $\tilde{S}$ we consider in this equivalence share the component Real-PKEnc and it will be useful for the remainder of the proof to use the following notation. Let E' denote $E \cdot$ Real-PKEnc (that is, E' is the environment enlarged by the shared functionality Real-PKEnc), let

$$T = \text{Real-RegEnc} \cdot \text{Ideal-PKI},$$
$$\tilde{T} = \text{Ideal-RegEnc} \cdot \mathit{Sim}.$$

Using this notation, (F.7), that is the equivalence to be proven, can be represented as

$$E' \cdot T \ \equiv_{\mathsf{perf}} \ E' \cdot \tilde{T} \tag{F.8}$$

Let $q = \langle e, (h,l) \rangle$ be a configuration of $E' \cdot T$ and $\tilde{q} = \langle \tilde{e}, (\tilde{h}, \tilde{l}) \rangle$ be a configuration of $E' \cdot \tilde{T}$.

We say that a bijection $f : R_1 \to R_2$, where R_1 and R_2 are subsets of the set of all references, is an $(h,\tilde{h})$-*congruence*, if for all $r, \tilde{r}$ such that $\tilde{r} = f(r)$ one of the following conditions holds true:

(i) Both r and $\tilde{r}$ point to objects of the same class C defined in E' and for every field m of C, either (a) both $r' = h(r).m$ and $\tilde{r}' = \tilde{h}(\tilde{r}).m$ have the same primitive value or (b) both r' and $\tilde{r}'$ are references and $\tilde{r}' = f(r')$.

(ii) Both r and $\tilde{r}$ point to an array of the same type T and the same length l such either (a) T is a primitive type and r and $\tilde{r}$ contain the same values or (b) T is a class and, for every $i \in \{0, \ldots, l-1\}$ and every pair of corresponding references $r' = h(r)[i]$ and $\tilde{r}' = \tilde{h}(\tilde{r})[i]$, we have $\tilde{r}' = f(r')$.

Let f be an $(h,\tilde{h})$-congruence. For primitive values $v, \tilde{v}$, we write $v \equiv_f \tilde{v}$, if simply $v = \tilde{v}$. For references $r, \tilde{r}$, we write $r \equiv_f \tilde{r}$, if $\tilde{r} = f(r)$. Finally, we extend the relation $\equiv_f$ to expressions by the structural isomorphism, that is $e \equiv_f \tilde{e}$ holds if and only if e and $\tilde{e}$ are (syntactically) equal, up to references occurring as their corresponding subexpressions which need to be in the relation $\equiv_f$.

We also define $l \equiv_f \tilde{l}$ to be true if, intuitively, the state of E' (given by static variables of E') is the same up to reference renaming f and the state of T and $\tilde{T}$, although different, represent essentially the same store of registered encryptors. Formally, we put $l \equiv_f \tilde{l}$ if and only if

(a) For every static variable x defined in E' we have $l(V) \equiv_f \tilde{l}(V)$.

(b) Static variable `IdealPKI.entries` (defined in T) and static variable `RegisterEnc.registeredAgents` (defined in $\tilde{T}$) contain information which is strictly corresponding in the following sense.

First let us observe that `IdealPKI.entries` points to a list of entries, each containing `id` of type `int`, and `domain` and `key` of type `byte[]`. Similarly, `RegisterEnc.registeredAgents` points to a list of entries, each containing `id` of type `int`, `domain` of type `byte[]`, and `domain` of type `Encryptor`.

Now, for we require that, for all values *id*, *domain*, and *key*, where *id* is an integer and *domain* and *key* are arrays of bytes, the following equivalence holds: the list pointed to by `IdealPKI.entries` contains a tuple with values (*id*, *domain*, *key*) if and only if the list pointed to by `RegisterEnc.registeredAgents` contains a tuple with values *id*, *domain* and an encryptor containing *key* as its public key (that is its field `publicKey` points to an array containing the bitstring *key*).

We say that $q = \langle e, (h,l) \rangle$ and $\tilde{q} = \langle \tilde{e}, (\tilde{h}, \tilde{l}) \rangle$ are *corresponding*, if there exists a $(h,\tilde{h})$-congruence f such that

1. $\mathrm{sub}_{E'}(e) \equiv_f \mathrm{sub}_{E'}(\tilde{e})$,

2. $l \equiv_f \tilde{l}$.

Condition 1 above means that the expressions e and $\tilde{e}$ (representing the code being executed), when stripped off the code originatin in $T/\tilde{T}$, are the same (up to reference renaming). Condition 2 says that the state, as given by static variables, is the same, up to reference renaming an up to (not-essential) differences in how T and $\tilde{T}$ store public keys.

We will sometimes write that q and $\tilde{q}$ are *f-corresponding* to make it explicit which congruence is used.

Lemma F.15. *Let $q = \langle e,(h,l)\rangle$ and $\tilde{q} = \langle \tilde{e},(\tilde{h},\tilde{l})\rangle$ be f-corresponding configurations for an $(h,\tilde{h})$-congruence f. Let $q \bar{\rightarrow} q'$ and $\tilde{q} \bar{\rightarrow} \tilde{q}'$. Then $q' = \langle e',(h',l')\rangle$ and $\tilde{q}' = \langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding for an $(h',\tilde{h}')$-congruence f' which is an extension of f.*

Proof. First of all, we prove that the Jinja+ rules applied to q and to $\tilde{q}$ are indeed the same. By Lemma F.13, since $\langle e,(h,l)\rangle \bar{\rightarrow} \langle e',(h',l')\rangle$ and $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle \bar{\rightarrow} \langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle$, e and $\tilde{e}$ contain no blocks. Therefore, by the definition of the pruning operator, we have $\text{sub}_{E'}(e) = e$ and $\text{sub}_{E'}(\tilde{e}) = \tilde{e}$ and hence, since they are f-corresponding, we have $e \equiv_f \tilde{e}$. Moreover, the f-corresponding relation means also $l \equiv_f \tilde{l}$.

The relation $\equiv_f$ between expressions implies that e and $\tilde{e}$ are syntactically equal, up to reference occurring as their corresponding subexpressions. Since applicability of no Jinja+ rule depends on the particular values of references, the same rule is applied to $\langle e,(h,l)\rangle$ and to $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle$.

We can now prove the q' and $\tilde{q}'$ are also corresponding, depending on the Jinja+ rule applied to both q and $\tilde{q}$.

Rule B.1. In this case we have

$$q = \langle \texttt{Cast}\, C\, e_1,(h,l)\rangle \bar{\rightarrow} \langle \texttt{Cast}\, C\, e_1',(h',l')\rangle = q'$$

and analogously

$$\tilde{q} = \langle \texttt{Cast}\, C\, \tilde{e}_1,(\tilde{h},\tilde{l})\rangle \bar{\rightarrow} \langle \texttt{Cast}\, C\, \tilde{e}_1',(\tilde{h}',\tilde{l}')\rangle = \tilde{q}',$$

where, by the premise of the rule,

$$\langle e,(h,l)\rangle \bar{\rightarrow} \langle e',(h',l')\rangle \quad \text{and} \quad \langle \tilde{e},(\tilde{h},\tilde{l})\rangle \bar{\rightarrow} \langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle.$$

By definition of $\equiv_f$ (which is by structural isomorphism), we have $e_1 \equiv_f \tilde{e}_1$ and, therefore, $\langle e_1,(h,l)\rangle$ and $\langle \tilde{e}_1,(\tilde{h},\tilde{l})\rangle$ are also f-corresponding. Therefore, by the inductive hypothesis, there exists an $(h',\tilde{h}')$-congruence f' such that $\langle e_1',(h',l')\rangle$ and $\langle \tilde{e}_1',(\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding, where f' is an extension of f. By the definition of $\equiv_{f'}$, we conclude that configurations $\langle \texttt{Cast}\, C\, e_1',(h',l')\rangle$ and $\langle \texttt{Cast}\, C\, \tilde{e}_1',(\tilde{h}',\tilde{l}')\rangle$ are also f'-corresponding.

Rule B.16. We have

$$q = \langle \texttt{new}\, C,(h,l)\rangle \bar{\rightarrow} \langle addr\ a,(h(a \mapsto\ (C,\ init\text{-}fields\ FDTs)),l)\rangle = q'$$

and

$$\tilde{q} = \langle \texttt{new}\, C,(\tilde{h},\tilde{l})\rangle \bar{\rightarrow} \langle addr\ \tilde{a},(\tilde{h}(\tilde{a} \mapsto\ (C,\ init\text{-}fields\ FDTs)),\tilde{l})\rangle = \tilde{q}',$$

where a and $\tilde{a}$ are fresh references (that is, references unused in h and $\tilde{h}$, respectively).

We extend $(h,\tilde{h})$-congruence f to an $(h',\tilde{h}')$-congruence f' in the following way: (i) $\mathtt{dom}(f') = \mathtt{dom}(f) \cup \{a\}$; (ii) $\forall r \in \mathtt{dom}(f), f(r) = f'(r)$; (iii) $\tilde{a} = f'(a)$. By the definition of $\equiv_{f'}$, we have $a \equiv_{f'} \tilde{a}$. Furthermore, since the rule leaves the stores l and $\tilde{l}$ unchanged, $l \equiv_{f'} \tilde{l}$. Therefore, q' and $\tilde{q}'$ are f'-corresponding.

Rule B.20. We have

$$q = \langle V := \mathtt{Val}\ v, (h,l)\rangle \stackrel{-}{\rightarrow} \langle unit, (h, l(V \mapsto\ v))\rangle = q'$$

and

$$\tilde{q} = \langle V := \mathtt{Val}\ \tilde{v}, (\tilde{h},\tilde{l})\rangle \stackrel{-}{\rightarrow} \langle unit, (\tilde{h}, \tilde{l}(V \mapsto\ \tilde{v}))\rangle = \tilde{q}'$$

with $l \equiv_f \tilde{l}$ by the lemma's hypothesis. Since $q \equiv \tilde{q}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we also have $v \equiv_f \tilde{v}$ and, since $l' = l(V \mapsto v)$ and $\tilde{l}' = \tilde{l}(V \mapsto \tilde{v})$, we also have $l' \equiv_f \tilde{l}'$. Therefore, since the rule leaves unchanged the heaps h and $\tilde{h}$, q' and $\tilde{q}'$ are f'-corresponding where $f' = f$.

Rule B.23. We have

$$q = \langle addr\ a.F\{D\} := \mathtt{Val}\ v, (h,l)\rangle \stackrel{-}{\rightarrow} \langle unit, (h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v))), l)\rangle = q'$$

and

$$\tilde{q} = \langle addr\ \tilde{a}.F\{D\} := \mathtt{Val}\ \tilde{v}, (\tilde{h},\tilde{l})\rangle \stackrel{-}{\rightarrow} \langle unit, (\tilde{h}(\tilde{a}\ \mapsto\ (C, fs((F,D)\ \mapsto\ \tilde{v}))), \tilde{l})\rangle = \tilde{q}'.$$

Since $q \equiv \tilde{q}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we have $a \equiv_f \tilde{a}$ and $v \equiv_f \tilde{v}$. The rule changes the heaps $h, \tilde{h}$ in such a way that the fields $F\{D\}$ of the references a,$\tilde{a}$ are updated with the values $v, \tilde{v}$, respectively:

$$h' = h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v)))$$

and

$$\tilde{h}' = \tilde{h}(\tilde{a}\ \mapsto\ (C, fs((F,D)\ \mapsto\ \tilde{v}))).$$

Since the two fields are updated with the corresponding values $v, \tilde{v}$, they remain corresponding also after the application of the rule. Therefore, by the definition of $\equiv_f$, the $(h,\tilde{h})$-congruence f is also an $(h',\tilde{h}')$-congruence. Since the rule leaves unchanged the stores l and $\tilde{l}$, $\langle e', (h', l')\rangle = \langle unit, (h', l)\rangle$ and $\langle \tilde{e}', (\tilde{h}', \tilde{l}')\rangle = \langle unit, (\tilde{h}', \tilde{l})\rangle$ are f'-corresponding where $f' = f$.

Rule B.24. In this case we have $q = \langle addr\ a.M(map\ \mathtt{Val}\ vs), (h,l)\rangle$ and analogously $\tilde{q} = \langle addr\ \tilde{a}.M(map\ \mathtt{Val}\ \widetilde{vs}), (\tilde{h},\tilde{l})\rangle$. By the definition of f-corresponding and of $\equiv_f$, we have $a \equiv_f \tilde{a}$ and $vs \equiv_f \widetilde{vs}$. Therefore both a and $\tilde{a}$ point to objects of the same class C defined in E'.

By the premise of the rule we have "$P \vdash\ C\ sees\ M:\ Ts \rightarrow T = (pns,\ body)\ in\ D$", which means that method M called for an object of class C (which is the actual class of the object a that we call M for) is defined in class D as $(pns, body)$, where pns are the parameters of the method

and *body* is its body). Now, when configuration q is considered, $P = E' \cdot T$; when $\tilde{q}$ is considered, $P = E' \cdot \tilde{T}$. One can see that in both these cases, the above relation gives the same *pns* and *body*. This is because, as we have noted, class C is defined in E' and this component does not refer to $T/\tilde{T}$.

Therefore

$$e' = blocks_D(this \cdot pns,\ Class\ D \cdot Ts,\ addr\ a \cdot vs,\ body)$$

and

$$\tilde{e}' = blocks_D(this \cdot pns,\ Class\ D \cdot Ts,\ addr\ \tilde{a} \cdot \widetilde{vs},\ body)$$

respectively.

Since $a \equiv_f \tilde{a}$ and $vs \equiv_f \widetilde{vs}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we have $e' \equiv_f \tilde{e}'$. Furthermore, since the rule does not change the state i.e., $(h,l) = (h',l')$ and $(\tilde{h},\tilde{l}) = (\tilde{h}',\tilde{l}')$ respectively, also $l' \equiv_f \tilde{l}'$ holds. Therefore $\langle e',(h',l')\rangle$ and $\langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding where $f' = f$.

We consider the remaining rules only quickly, as the reasoning they require is either trivial or similar to the cases discussed above.

- Rules B.2-B.7, B.11-B.15, B.35, B.37, B.55-B.57 can be proven by following the inductive reasoning (rule induction) for the rule B.1, since they also perform just a reduction step in one of their subexpressions.
- Rules B.8, B.9, B.10 cannot be applied to q and $\tilde{q}$ because the function $g(\ell,D)$, which defines the transition's label of these rules, never returns $-$.
- Rules B.17, B.19, B.21, B.22, B.31, B.39, B.40, B.64-B.67, B.69,B.70 can be proven by following the reasoning for rule B.24 since they also leave the state of the configurations unchanged: $(h,l) = (h',l')$ and $(\tilde{h},\tilde{l}) = (\tilde{h}',\tilde{l}')$ respectively. Therefore, the $(h',\tilde{h}')$-congruence f' is such that $f' = f$.
- Rules B.18, B.27-B.30, B.32-B.34, B.36, B.38, B.41-B.47, B.50-B.54, B.58-B.60, B.63 can be proven by following the reasoning for rule B.24 because also these rules leave the state of the configurations unchanged (hence, $f' = f$) and, moreover, they are trivial to prove since they do not have any premise (as in case of rule B.20).
- Rules B.25, B.26, B.48, B.49 cannot be applied to q and $\tilde{q}$ because their transition's label is D.
- Rule B.61 can be proven by following the reasoning for rule B.24, with the only difference that, since the method $D.M$ is static, the local variable *this* does not appear as argument of the auxiliary block function.
- Rule B.62 can be proven by following the reasoning for rule B.16: since also in this case two new array references a and $\tilde{a}$ are created inside their heaps h and $\tilde{h}$ respectively, the $(h,\tilde{h})$-congruence f must be extended in the same way i.e., with a $(h',\tilde{h}')$-congruence f' such that $a \in \mathtt{dom}(f')$ and $\tilde{a} = f'(a)$.

- Rule B.68 can be proven by following the same reasoning of rule B.23: two array references $r = h(a)[n]$ and $\tilde{r} = \tilde{h}(\tilde{a})[n]$ are updated, but also in this case the $(h,\tilde{h})$-congruence f remains unchanged for $(h',\tilde{h}')$ i.e., $f' = f$.

□

E'-configurations. We say that a configuration q is an E'-configuration, if E' has control at q, i.e. $q \xrightarrow{\ell} q'$ for $\ell \in E'$.

Let q be an E'-configuration. We write $q \mapsto q'$, if q' is also an E'-configuration and

$$q \xrightarrow{\ell_0} q_1 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_{n-1}} q_n \xrightarrow{\ell_n} q'$$

where ($\ell_0 \in E'$ and) $\ell_i \notin E'$ for $i \in \{1,\ldots,n\}$, that is $q_1,\ldots,q_n$ are not E'-configurations. (Note that the special case of the above definition is when q' is obtained from q in one step). As we can see, q' is the first E'-configuration after q.

Now one can prove that two E' configurations q and $\tilde{q}$ that are corresponding reduce in one step to configurations which are also corresponding:

Lemma F.16. *Let q and $\tilde{q}$ be corresponding E'-configurations. Let $q \to q'$ and $\tilde{q} \to \tilde{q}'$. Then q' and $\tilde{q}'$ are also corresponding.*

Proof. Let $q = \langle e,(h,l)\rangle$ and $\tilde{q} = \langle \tilde{e},(\tilde{h},\tilde{l})\rangle$. Since they are E'-configurations, we have $\langle e,(h,l)\rangle \xrightarrow{D} \langle e',(h',l')\rangle$ and $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle \xrightarrow{\tilde{D}} \langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle$, respectively, with $D,\tilde{D} \in E'$. By Lemma F.14, we then have:

- $e = \mathfrak{C}[e_0]$, where e_0 is a block expression of class D without nested blocks;
- $\tilde{e} = \tilde{\mathfrak{C}}[\tilde{e}_0]$, where $\tilde{e}_0$ is a block expression of class $\tilde{D}$ without nested blocks;
- $\langle e_0,(h,l)\rangle \xrightarrow{D} \langle e_0',(h',l')\rangle$ and $e' = \mathfrak{C}[e_0']$;
- $\langle \tilde{e}_0,(\tilde{h},\tilde{l})\rangle \xrightarrow{\tilde{D}} \langle \tilde{e}_0',(\tilde{h}',\tilde{l}')\rangle$ and $\tilde{e}' = \tilde{\mathfrak{C}}[\tilde{e}_0']$.

Since $\langle e,(h,l)\rangle$ and $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle$ are corresponding configurations, we have $\mathrm{sub}_{E'}(\mathfrak{C}[e_0]) \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}}[\tilde{e}_0])$ for some $(h,\tilde{h})$-congruence f and hence, by the definition of pruning operator, $\mathrm{sub}_{E'}(\mathfrak{C})[\mathrm{sub}_{E'}(e_0)] \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}})[\mathrm{sub}_{E'}(\tilde{e}_0)]$. By the definition of $\equiv_f$ (which is by structural isomorphism), we then have:

- $\mathrm{sub}_{E'}(\mathfrak{C}) \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}})$ and
- $\mathrm{sub}_{E'}(e_0) \equiv_f \mathrm{sub}_{E'}(\tilde{e}_0)$.

Therefore $\langle e_0,(h,l)\rangle$ and $\langle \tilde{e}_0,(\tilde{h},\tilde{l})\rangle$ are also f-corresponding and $D = \tilde{D}$, because $D,\tilde{D} \in E'$ and, hence, the block expressions e_0 and $\tilde{e}_0$ are preserved by $\mathrm{sub}_{E'}$. Furthermore, since e_0 and $\tilde{e}_0$ contain no block as their proper subexpressions, by the definition of pruning operator, we have $\mathrm{sub}_{E'}(e_0) = e_0$ and $\mathrm{sub}_{E'}(\tilde{e}_0) = \tilde{e}_0$ and therefore $e_0 \equiv_f \tilde{e}_0$.

The relation $\equiv_f$ between expressions implies that e_0 and $\tilde{e}_0$ are syntactically equal, up to reference occurring as their corresponding subexpressions. But, since aplicability of Jinja+ rules does not depend on particular values of references, the same block rule is applied to both $\langle e_0,(h,l)\rangle$ and $\langle \tilde{e}_0,\tilde{h},\tilde{l})\rangle$.

We now prove that $\langle e'_0,(h',l')\rangle$ and $\langle \tilde{e}'_0,(\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding for an $(h',\tilde{h}')$-congruence f' which is an extension of f. We need to distinguish the following cases, depending on the block rule applied to e_0 (and thus to e'_0).

(a) Rule B.8, B.9: we have $e_0 = \{V : T; e_1\}_D$, where neither $e_1 = \texttt{Val}\ u$ nor $e_1 = \texttt{Throw}\ a$, and $g(\ell,D) = D$.

(b) Rule B.25: we have $e_0 = \{V : T;\ \texttt{Val}\ u\}_D$.

(c) Rule B.48: we have $e_0 = \{V : T;\ \texttt{Throw}\ a\}_D$.

(d) Rule B.10: we have $e_0 = \{V : T; V := \texttt{Val}\ v; e_1\}_D$, where neither $e_1 = \texttt{Val}\ u$ nor $e_1 = \texttt{Throw}\ a$, and $g(\ell,D) = D$.

(e) Rule B.26: we have $e_0 = \{V : T;\ V := \texttt{Val}\ v\}_D$.

(f) Rule B.49: we have $e_0 = \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D$.

Let us consider the case (a) and (b); the case (d) is analogous to case (a), whereas the cases (c), (e) and (f) are analogous to case (b) and, moreover, trivial because the rules do not assume any premise and do not change the state of the configurations.

Case (a): We have $e_0 = \{V : T; e_1\}_D$ and $\tilde{e}_0 = \{V : T; \tilde{e}_1\}_D$. Furthermore, by definition of $\equiv_f$, we have $e_1 \equiv_f \tilde{e}_1$ and hence $\langle e_1,(h,l)\rangle$ and $\langle \tilde{e}_1,(\tilde{h},\tilde{l})\rangle$ are f-corresponding.

The block rule applied is either B.8 or B.9. Let us consider the rule B.8; reasoning for the other other is analogous. Since e_1 and $\tilde{e}_1$ do not contain any blocks, by Lemma F.13

$$\langle e_1,(h,l(V := None)\rangle \bar{\to} \langle e'_1,(h',l'_1)\rangle \text{ with } l' = l'_1(V := l\,V) \tag{F.9}$$

and

$$\langle \tilde{e}_1,(\tilde{h},\tilde{l}(V := None)\rangle \bar{\to} \langle \tilde{e}'_1,(\tilde{h}',\tilde{l}'_1)\rangle \text{ with } \tilde{l}' = \tilde{l}'_1(V := \tilde{l}\,V). \tag{F.10}$$

By Lemma F.15, $\langle e'_1,(h',l'_1)\rangle$ and $\langle \tilde{e}'_1,(\tilde{h}',\tilde{l}'_1)\rangle$ are f'-corresponding for an $(h',\tilde{h}')$-congruence f' which is an extension of f. It implies $\text{sub}_{E'}(e'_1) \equiv_{f'} \text{sub}_{E'}(\tilde{e}'_1)$ and $l'_1 \equiv_{f'} \tilde{l}'_1$. Therefore, by the definition of l' in $(F.9)$ and $\tilde{l}'$ in $(F.10)$, we have

$$l' \equiv_{f'} \tilde{l}'. \tag{F.11}$$

By the definition of $\equiv_{f'}$ (which is by structural isomorphism), we have

$$\{V : T;\ \text{sub}_{E'}(e'_1)\}_D \equiv_{f'} \{V : T;\ \text{sub}_{E'}(\tilde{e}'_1)\}_D, \tag{F.12}$$

and, hence, by the definition of the pruning operator, we obtain

$$\text{sub}_{E'}(e'_0) = \text{sub}_{E'}(\{V : T; e'_1)\}_D) \equiv_{f'} \text{sub}_{E'}(\{V : T; \tilde{e}'_1\}_D) = \text{sub}_{E'}(\tilde{e}'_0). \tag{F.13}$$

By relations $(F.11)$ and $(F.13)$, $\langle e'_0,(h',l')\rangle$ and $\langle \tilde{e}'_0,(\tilde{h}',\tilde{l}')\rangle$ are also f'-corresponding.

Case (b): We have $e_0 = \{V : T;\ \texttt{Val}\ u\}_D$ and $\tilde{e}_0 = \{V : T;\ \texttt{Val}\ \tilde{u}\}_D$. Furthermore, by the definition of $\equiv_f$, we have $\texttt{Val}\ u \equiv_f \texttt{Val}\ \tilde{u}$. The block rule applied is B.25, where $\langle e'_0, (h', l')\rangle = \langle \texttt{Val}\ u, (h, l)\rangle$ and $\langle \tilde{e}'_0, (\tilde{h}', \tilde{l}')\rangle = \langle \texttt{Val}\ \tilde{u}, (\tilde{h}, \tilde{l})\rangle$, respectively. In particular, since $l \equiv_f \tilde{l}$ (by the lemma's hypothesis) and since $l' = l$ and $\tilde{l}' = \tilde{l}$, we also have $l' \equiv_f \tilde{l}'$.

Therefore $\langle e'_0, (h', l')\rangle$ and $\langle \tilde{e}'_0, (\tilde{h}', \tilde{l}')\rangle$ are f'-corresponding, where $f' = f$.

We have proven that $\langle e'_0, (h', l')\rangle$ and $\langle \tilde{e}'_0, (\tilde{h}', \tilde{l}')\rangle$ are f'-corresponding. In particular, it means that $\text{sub}_{E'}(e'_0) \equiv_{f'} \text{sub}_{E'}(\tilde{e}'_0)$. Since $\text{sub}_{E'}(\mathfrak{C}) \equiv_f \text{sub}_{E'}(\tilde{\mathfrak{C}})$ and f' is an extension of f, by the definition of the pruning operator, we have

$$\text{sub}_{E'}(e') = \text{sub}_{E'}(\mathfrak{C})[\text{sub}_{E'}(e'_0)] \equiv_{f'} \text{sub}_{E'}(\tilde{\mathfrak{C}})[\text{sub}_{E'}(\tilde{e}'_0)] = \text{sub}_{E'}(\tilde{e}'). \tag{F.14}$$

By $(F.11)$ and $(F.14)$, we conclude that $q' = \langle e', (h', l')\rangle$ and $\tilde{q}' = \langle \tilde{e}', (\tilde{h}', \tilde{l}')\rangle$ are corresponding, which completes the proof. □

Now we are ready to prove the following statement.

Lemma F.17. *Let q_0 be the initial configuration of the run of $E' \cdot T$ and $\tilde{q}_0$ be the initial configuration of the run of $E' \cdot \tilde{T}$. Let $q_1, \dots, q_n$ and $\tilde{q}_1, \dots, \tilde{q}_n$ be configurations such that $q_0 \mapsto q_1 \mapsto \cdots \mapsto q_n$ and $\tilde{q}_0 \mapsto \tilde{q}_1 \mapsto \cdots \mapsto \tilde{q}_m$ where q_n and $\tilde{q}_m$ are final configuration (i.e. configuration that do not reduce). Then $n = m$ and, for all $i \in \{0, \dots, n\}$, the configurations q_i and $\tilde{q}_i$ are corresponding.*

Proof. We will prove a more general fact, than the one stated in the lemma, allowing q_0 and $\tilde{q}_0$ to be any corresponding E-states (not necessarily the initial ones). The proof proceeds by induction on the number of E'-blocks—that is block expressions of the form $\{\dots\}_C$ with $C \in E'$—in q_0 (and $\tilde{q}_0$), where to prove that the statement is true for configurations with a given number of E'-blocks, we assume that it holds true for configurations with bigger numbers of E'-blocks.

If q and $\tilde{q}$ are corresponding and $q \mapsto q'$ where q' is obtained from q in one step, then $\tilde{q} \mapsto \tilde{q}'$ where $\tilde{q}'$ is obtained also in one step. Moreover, by Lemma F.16, the configurations q' and $\tilde{q}'$ are corresponding. Therefore, to complete the proof, it is enough to consider the remaining case (i.e., if q' is not obtained from q in one step) and show that calls to the (public) methods of T and $\tilde{T}$ do not brake this property. We consider, on the case by case basis, all calls from E' to methods of $T/\tilde{T}$ (that is to `registerEncryptor` and to `getEncryptor`) made in corresponding states and show that they end in corresponding states as well. Here we present the reasoning only for the former case (the proof for `getEncryptor` proceeds in a similar way).

Method `registerEncryptor` with arguments `encryptor, id, pki_domain`:
First, we can observe (by inspecting the code of this method in $T/\tilde{T}$, see Appendix D.2 and D.2), these methods do not change the state of E' which formally means, that they preserver condition (a) of the definition of corresponding states. Therefore, it is enough to show that a call to this method also preserves condition (b) of this definition.

This call, in both systems T and $\tilde{T}$ makes three steps:

1. Method `PKI.register` is called with arguments `id`, the PKI domain `pki_domain`, and k, where k is the public key stored in `encryptor`.

Indeed, in the system T, the control is immediately handed over to `PKISim.register` (with arguments `id`, `pki_domain`, k; see line 41), where method `PKISim.register` is called with the same arguments. In the system $\tilde{T}$, on the other hand, the first things that happens is the call to `register` method of class `RegisterEncSim` (line 48, Appendix D.2) with arguments `id`, `pki_domain`, and k. By the definition of the simulator, this call is directly translated into the corresponding call to method `PKISim.register`.

As this is the first action in both systems, the configurations of the systems $T/\tilde{T}'$ when this method is called remain corresponding. Hence, by the inductive hypothesis, the state of these systems after the call are corresponding as well. This includes the return value from the method call (as this value has been determined by E' in corresponding states).

Finally, in both systems, if the return value from the call to `PKISim.register` is true (which is, as we have noticed, the same in both systems), exception `NetwokError` is thrown. If this is the case, the method call is aborted in corresponding states. Otherwise, the systems enter the next step in corresponding states.

2. It is checked if a key has already been registered for the given `id` and `pki_domain` (line 5 for the system T and line 50 for the system $\tilde{T}$).

   One can notice, again, that this step does not change the state of the system and therefore preserves correspondence of states of $T/\tilde{T}$. Moreover, by condition (b) of the definition of corresponding states, the result of this step is the same in both systems. Therefore either both T and $\tilde{T}$ throw `PKIError` in corresponding states, or both T and $\tilde{T}$ enter the next step in corresponding states.

3. A public key/encryptor is registered under `id` and `pki_domain`.

   This is done in line 6 for the system T and in line 52 for the system $\tilde{T}$. One can see, by inspecting the code of the invoked methods, that the only part of the state that is changed are the collections considered in condition (b) of the definition of corresponding states. So, condition (a) of this definition is preserved by this step. Moreover, the changes made to the considered collections are such that condition (b) is also preserved after this step. Hence, the states of T and $\tilde{T}$ after this step are still corresponding.

□

Now we can complete the proof of Lemma F.11. By the above lemma, the final configurations of $(E \cdot S) = (E' \cdot T)$ and $(E \cdot \tilde{S}) = (E' \cdot \tilde{T})$ (that is, respectively, q_n and $\tilde{q}_m$, as defined in the lemma) are corresponding, which means, in particular, that the state of E', which includes the variable `result` in those configurations is the same. Therefore the environment outputs the same result in both cases. □

F.4.2. Proof of Lemma F.12

In this section, we provide the proof of the realization result for public-key encryption:

$$\text{Real-PKIEnc} \leq^{I_{PKIEnc}} \text{Ideal-PKIEnc}. \tag{F.15}$$

In our previous paper [KTG12a] we considered the case without corruption. In this paper, we consider an extended case with (static) corruption: in our Jinja+ implementation, we model corruption by allowing the direct creation of `Encryptor` objects with an arbitrary public key provided by the adversary.

We structure the proof in the following way: first we discuss the functionalities without corruption by referencing to a result obtained in [KTG12a] and then, based on this result, we consider the case with corruption and prove Lemma F.12.

The Functionalities without Corruption.

The real functionality of public key encryption without corruption, as considered in [KTG12a] coincides with the real functionality with corruption we consider in this paper. The ideal functionality for public key encryption without corruption, as considered in [KTG12a], is, however, different. We will denote it by Ideal-PKEnc^{-}.

Similarly, the interface they implement (again, as considered in [KTG12a]) will be denoted by I^{-}_{PKEnc}. For completeness, we recall this interface (note that the difference to I_{PKEnc} is the lack of the constructor of class `Encryptor`):

```
1  public final class Decryptor {
2      public Decryptor();
3      public Encryptor getEncryptor();
4      public byte[] decrypt(byte[] message);
5  }
6  public final class Encryptor {
7      public byte[] getPublicKey();
8      public byte[] encrypt(byte[] message);
9  }
```

We have the following result proven in [KTG12a]:

Lemma F.18. $\text{Real-PKEnc} \leq^{I^{-}_{PKIEnc}} \text{Ideal-PKEnc}^{-}$

The Functionalities with Corruption.

Now, using the result just discussed, we prove (F.15). That is, we show that there exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I_{PKEnc}-environment E we have (Section 2.3):

$$E \cdot \text{Real-PKEnc} \equiv_{\text{comp}} E \cdot S \cdot \text{Ideal-PKEnc} \tag{F.16}$$

In order to reduce this proof to the case without corruption, we take an arbitrary I_{PKEnc}-environment E and construct a new I^{-}_{PKEnc}-environment E^{-} out of it. This environment E^{-} consists of the following parts:

1. A copy of the code of the class `Encryptor` from the real functionality renamed `EncryptorCorr`. (Note that the code of class `Encryptor` in the real functionality and the ideal is identical).

2. A new class `EncryptorWrapper` which is meant to wrap either an object of class `Encryptor` of the interface I^{-}_{PKEnc} (objects of this class are returned by `PKI.Decryptor.getEncryptor()`), or an object of class `EncryptorCorr`, as introduced above.

```
public class EncryptorWrapper {
  Encryptor enc;
  EncryptorCorr encCorr;

  public EncryptorWrapper(Encryptor enc) {
    this.enc=enc;
    this.encCor=null;
  }
  public EncryptorWrapper(EncryptorCorr encCorr) {
    this.encCorr=encCorr;
    this.enc=null;
  }
  public byte[] encrypt(byte[] message) {
    if(enc!=null)
      return enc.encrypt(message);
    else
      return encCorr.encrypt(message);
  }
  public byte[] getPublicKey(){
    if(enc!=null)
      return enc.getPublicKey();
    else
      return encCorr.getPublicKey();
  }
}
```

3. A copy of the code of E modified in the following way:

   (a) every expression where an encryptor is obtained by a decryptor i.e.,
   ```
   decryptor.getEncryptor()
   ```
   is replaced by
   ```
   new EncryptorWrapper(decryptor.getEncryptor())
   ```

   (b) every expression where a corrupted encryptor is directly created i.e.,
   ```
   EncryptorCorr(pubk)
   ```
   is replaced by
   ```
   new EncryptorWrapper(new EncryptorCorr(pubk));
   ```

The reason for using the wrapper class is to make it possible to treat objects of two, formally unrelated classes (the encryptor class provided by the environment and the encryptor class provided by the functionality) in a uniform way.

Using this construction, we can state the two following lemmas.

Lemma F.19. $E \cdot \textsf{Real-PKEnc} \equiv_{\textsf{comp}} E^{-} \cdot \textsf{Real-PKEnc}$

sketch. The proof is quite straightforward and it it follows by the construction of E^-. The class `EncryptorCorr` contains a copy of the code of class `Encryptor` of Real-PKEnc. Furthermore, the wrapper and the modified version of E perform the same actions (up to additional relaying steps of the wrapper class).

The presented above reasoning can be strictly formalized, as it has been in the proof of Lemma F.11. The difference to the proof of Lemma F.11 is that now we cannot prove perfect indistinguishability of the considered system, but the following property (which is still stronger than the one postulated in the lemma): the considered systems behave in exactly the same way from the point of view of an unbounded (but possibly probabilistic) adversary, for the same sequence of random coins. □

In a very similar way we can prove the following result.

Lemma F.20. $E \cdot S \cdot \mathsf{Ideal\text{-}PKEnc} \equiv_{\mathsf{comp}} E^- \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^-$

Now we are ready to complete the proof of Lemma F.12. From Lemma F.18, we know that $\mathsf{Real\text{-}PKEnc} \leq^{I^-_{PKEnc}} \mathsf{Ideal\text{-}PKEnc}^-$ i.e., exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I^-_{PKEnc}-environment E^- we have (Section 2.3):

$$E^- \cdot \mathsf{Real\text{-}PKEnc} \equiv_{\mathsf{comp}} E^- \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^- \tag{F.17}$$

Therefore, we obtain:

$$\begin{aligned} E \cdot \mathsf{Real\text{-}PKEnc} &\overset{\text{Lemma F.19}}{\equiv_{\mathsf{comp}}} E^- \cdot \mathsf{Real\text{-}PKEnc} \\ &\overset{(F.17)}{\equiv_{\mathsf{comp}}} E^- \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^- \\ &\overset{\text{Lemma F.20}}{\equiv_{\mathsf{comp}}} E \cdot S \cdot \mathsf{Ideal\text{-}PKEnc} \end{aligned} \tag{F.18}$$

F.5. Proof of Theorem 4.2

The proof of Theorem 4.2 is, as it is in the case of the realization result for PKIEnc, modular and uses such properties of the realization relation as the composition theorem, reflexivity, and transitivity.

We begin with analyzing the structure of the ideal and real functionalities we consider. The ideal functionality consists of the following components:

Ideal-Sig — the (ideal) implementation of digital signatures, i.e. classes `Verifier` and `Signer`, as described in Section 4.2.2. Let $I_{\mathsf{CryptoLibSig}}$ denote the public interface of this component.

Ideal-SigPKI — the (ideal) implementation of verifier registration, that is of the class `RegisterSig`. Let I_{SigPKI} denotes the public interface of this component.

Similarly, the real functionality for PKISig consists of the following components:

Real-Sig — the (real) implementation of classes `Verifier` and `Signer`, as sketched in Section 4.2.3. Note that the public interface of this component is $I_{\mathsf{CryptoLibSig}}$, as in the case of Ideal-Sig.

Real-SigPKI — the (real) implementation of verifier registration, that is `RegisterSig`. This implementation uses the next component, Real-PKI, and only wraps/unwraps verification keys into/from verifiers. Note that the public interface of this component is I_{SigPKI}.

Real-PKI — The real implementation of the functionality for the public key infrastructure (see Section 4.1.3).

Now, proving Theorem 4.2 can be reduced (in an analogous way as in the proof of Theorem 4.1) to proving the following two facts:

Lemma F.21. Real-Sig · Real-SigPKI · Ideal-PKI $\leq^{I_{PKISig}}$ Real-Sig · Ideal-SigPKI.

The proof of this lemma is very similar to the proof of Lemma F.11 given in Appendix F.4.1.

Lemma F.22. Real-Sig $\leq^{I_{PKISig}}$ Ideal-Sig.

The rest of this section is devoted to proving this lemma. We organize this proof in a similar way to the proof of Lemma F.12. First, we discuss the case without corruption, where the adversary (the environment) cannot create verifiers with arbitrary verification keys. Then we extend the proof to the case with (static) corruption, where the adversary can directly create verifiers with an arbitrary verification keys.

F.5.1. The Functionality without Corruption

As in the proof of Lemma F.12, we denote ideal functionality without corruption as Ideal-Sig^{-} (see Appendix D.3.1 for the code). Similarly, the interface the real and ideal functionalities for digital signatures without corruption implement is denoted by $I^{-}_{\mathsf{CryptoLibSig}}$.

```
1 public final class Signer {
2   public Signer();
3   public byte[] sign(byte[] message);
4   public Verifier getVerifier();
5 }
6 public class Verifier {
7   public boolean verify(byte[] signature, byte[] message);
8   public byte[] getVerifKey();
9 }
```

Note that the only difference to the interface $I_{\mathsf{CryptoLibSig}}$ is the lack of the constructor of the class `Verifier`.

To prove the following theorem we use here a proof technique similar to those used in [KTG12a]: we reduce the problem to the corresponding problem stated in the Turing Machine representation in order to use a result from [KT08a, KT08b].

Lemma F.23. Real-Sig $\leq^{I^{-}_{PKISig}}$ Ideal-Sig^{-}

Proof. Before we give the proof, we want to point out some critical points and assumptions that are used in this proof.

1. First, we assume that we have a correct implementation of an EUF-CMA-secure (existential unforgeability under adaptive chosen-message attacks) digital signatures scheme (we do not prove correctness of this implementation). We assume that, in particular, the above mentioned implementation does not fail (i.e. always returns the expected result) unless the expected result is too big to fit within an array (recall that the maximum size of an array depends on the security parameter and the function *intsize*).

   We also assume that this signature scheme is such that the length of a signature is the (polynomially computable) function of the length of the signed message and vice versa.

2. It is critical to assume that the Jinja+ program has unbounded memory, as otherwise the asymptotic notion of security our results are based upon does not make sense.

Now, we shortly present an ideal functionality $\mathscr{F}$ and a real functionality $\mathscr{R}$ for digital signatures in the Turing machine model following [KT08a, KT08b].

TM functionalities. Different instances of functionalities are distinguished by different *id*-s, sent with each request. The functionalities accept the following requests (where the request is written on the input tape of a TM):

1. *Initialization-Signer*: The functionality is supposed to return a verification key *vk*.

2. *Initialization-Verifier* The functionality responds with the message "*comleted*".

3. *Signature-Generation(m)*: The functionality is supposed to sign *m* using the stored signing key and return the signature σ.

4. *Signature-Verification(vk, m, σ)*: The functionality is supposed to verify that σ is a valid signature for the message *m* under the verification key *vk*.

Both the real and the ideal functionalities, on initialization, obtain a corruption bit. As already explained at the beginning of this section, because for now we handle the case without corruption i.e., in our simulation, the environment never corrupts functionalities, we will skip the description of actions of these functionalities if this bit is set to 1.

The real functionality $\mathscr{R}$, on initialization (be it *Initialization-Signer* or *Initialization-Verifier*), generates a fresh verification/signing key pair and returns the verification key. Then, it uses the signing key to sign messages, and the key *vk* provided in the verification request to verify messages.

The ideal functionality, on creation, asks the environment (the simulator) for verification and signing algorithms as well as a verification (public) and a signing (private) key. On signing requests, it (similarly to the considered Jinja+ functionality) computes a signature for the message provided using the given signing algorithm and private key. Then, by using the recorded verification algorithm and public key, it checks whether the signature verifies or not. If this check fails, it returns an error message. Otherwise, it records the message provided (to prevent forgery) and returns the signature. On verification, if the key *vk* is the same as the verification key stored in the functionality, it verifies the signature σ for *m* using the provided *vk* and checks that *m* has been stored as signed; see [KT08a, KT08b] for details.

We want to prove that Real-Sig realizes Ideal-Sig$^-$ w.r.t. I^-_{PKISig}. In this proof we will use a result from [KT08a, KT08b] that $\mathscr{R}$ realizes $\mathscr{F}$. Let $\mathscr{S}$ be the simulator used in the realization proof in [KT08a, KT08b]. The simulator for Ideal-Sig$^-$ we will use in the proof is $S =$ EUF-CMA, as described above.

Let E be a bounded-environment with $I^-_{PKISig} \vdash E$.

Simulating E. We define a Turing machine M_E that simulates E. Clearly, every complete Jinja+ program can be simulated by Turing machine. Moreover, if a program is bounded (for a given *intsize*), then its simulation is also polynomial (recall that a run with security parameter η uses integers of maximal size $intsize(\eta)$; operations on integers of this size can be polynomially simulated by a Turing machine).

In our case, however, the system E we consider is not a complete Jinja+ program; it interacts with another system (such as Real-Sig or Ideal-Sig$^-$). Therefore we assume that M_E communicates with another Turing machine (or more generally, a system of Turing machines).

The machine M_E is defined in such a way that it maintains a representation of a Jinja+ state, the state of E. In this representation, references are represented by consecutive identifiers. We distinguish two types of references: those pointing to an *internal* object, that is instances of a classes defined in E or an arrays, and those pointing to an *external* object which can be either instances of Signer or Verifier. For each reference to an internal object, a representation of this object is maintained by M_E. For references to external objects this is not the case (some additional information, however, is stored along with these references; see below). A method call for an internal reference is modelled internally by M_E; a method call to an external object is realized by triggering another Turing Machine.

When the simulation of E by M_E if finished, this machine outputs the value of the (simulated) variable `result`.

Method invocations for external references are simulated in the following way:

1. **Creating a new instance of `Signer`:** M_E creates a new instance of *Signer* (Turing Machine) by sending the *Initialization-Signer* request with a fresh identifier *id*. This identifier will be used as the reference to this object. M_E waits then for a response containing a verification key. This key is stored together with *id*.

2. **`Signer.getVerifier` for an object represented by *id*:** M_E creates a new instance of *Verifier* (TM) by sending the *Initialization-Verifier* request with *id* and a fresh identifier *id*$'$, which will serve as the reference to this object. The identifier *id*$'$ is stored together with *id*.

3. **`Signer.sign` for an object represented by *id* and array *m*:** M_E sends *Signature-Generation* request to machine *id* with the data stored under *m*, and waits for the response. A response is a sequence of bytes. M_E simulates creation of a new array and copies the obtained byte-sting to this array.

4. **`Verifier.verify` for an object represented by *id*$'$ and two arrays *m* and σ:** M_E retrieves the verification key associated with *id*$'$ and uses it in the request *Signature-Verification* along with *id*$'$ and the data stored under both *m* (the message) and σ (the signature). A response is one bit. M_E retrieves the response.

5. **`Verifier.getVerifKey` for an object represented by id':** M_E retrieves the verified key associated with the Verifier (without any external call).

Representing runs. Let T be either the system Real-Sig or the system $(S \cdot \text{Ideal-Sig}^-)$. Let u be a random input (a sequence of bits) and η be a security parameter. The (deterministic) finite run ρ of $E \cdot T$ with random input u and security parameter η can be represented as

$$A_1[s_1, x_1]B_1[t_1, y_1]A_2 \cdots B_{n-1}[t_{n-1}, y_{n-1}]A_n[s_n]$$

where

- Every A_i is a part of the run (a sequence of configurations) where only expressions originating from E are reduced, i.e. all the transitions in A_i are labelled with names of classes defined in E. Every A_i, except for the last one, ends with a state of the form $(e_i[e'_i], s_i)$ where the subexpression e'_i is about to be rewritten by a method invocation rule.
- Every B_i is a part of run where only expressions originating from T are reduced. It begins with $(e_i[\{e''_i\}_D], s_i)$, where $\{e''_i\}_D$ is the block obtained by applying the method invocation rule to e'_i for some class D defined in T (it depends only on e'_i), and ends with $(e_i[\{v_i\}_D], \bar{s}_i)$, where v_i is a value (that is return by the method).
- s_i and t_i are the states after A_i and B_i, respectively.
- By x_i we denote the *invocation data* consisting of the name of the called method and the values passed as arguments (if an argument is of type `byte[]` then x_i contains the values in the array, not the reference to this array). This data is determined by e_i and s_i.
- By y_i we denote the return value (again, if an array is returned, then y_i contains the values in this array, not the reference). This return value is determined by v_i and t_i.

Similarly, we represent the (deterministic) execution $\tilde{\rho}$ of the system of Turing Machines $M_E|M_T$ with random input u and security parameter η, where M_E is defined above and M_T is either $\mathscr{R}$ or $(\mathscr{S}|\mathscr{F})$ as

$$\tilde{A}_1[\tilde{s}_1, \tilde{x}_1]\tilde{B}_1[\tilde{t}_1, \tilde{y}_1]\tilde{A}_2 \cdots \tilde{B}_{n-1}[\tilde{t}_{n-1}, \tilde{y}_{n-1}]\tilde{A}_n[s_n]$$

where

- Every $\tilde{A}_i$ is a part of the run of the system where M_E is active. Every $\tilde{A}_i$, except for the last one, ends with M_E sending data $\tilde{x}_i$ to M_T (and activating M_T).
- Every $\tilde{B}_i$ is a part of the run of the system where M_T is active. It ends with M_T sending a response $\tilde{y}_i$ back to M_E.
- $\tilde{s}_i$ is the state of M_E after $\tilde{A}_i$ (notice the difference to s_i which was the state of the whole system after A_i).
- $\tilde{t}_i$ is the state of M_T after $\tilde{B}_i$ (notice, as above, the difference to t_i).

Let $s = (h,l)$ be a Jinja+ state that occurs in the run ρ of $E \cdot T$. We want to define the part of the state s that "belongs" to E and the part that "belongs" to T.

We define $h|_E$ to be the restriction of h to only those references that, in the run ρ, have been created by E or have been obtained by E as a return value from a call to T. By $h|_T$ we denote the restriction of s to the remaining references, that is the references in the run ρ that have been created by T but not returned to E.

We define $l|_E$ to be the restriction of l to those (static) variables that are accessible from E. Similarly, $l|_T$ denotes the restriction of l to those static variables that are accessible from T. Note that these restrictions are disjoint except for the read-only security parameter (T does not access any static fields of E; E does not access any static fields of T).

We take $s|_E = (h|_E, l|_E)$ and $s|_T = (h|_T, l|_T)$.

Let $\tilde{s}$ be a Jinja+ state as represented by M_E and s be a (real) Jinja+ state. We say that $\tilde{s}$ represents $s = (h,l)$, written $\tilde{s} \models s$, if there is a function f from identifiers (that represent references in M_E) to (Jinja+) references (addresses) such that

- the domain of h is $f(X)$ where X is the set of identifiers used by M_E to represent references,
- if $\tilde{r} \in X$, then the representation of the object pointed by $\tilde{r}$ agrees with the object pointed by $r = f(\tilde{r})$ (in the Jinja state) in the following sense: (i) corresponding fields (in the TM representation and in the Jinja object) of primitive types have the same values, (ii) if a field of the TM representation contains an identifier *id*, then the corresponding field of the Jinja object contains $f(id)$.
- The values of variables in l are—up to mapping f—the same as the values in the TM representation of l.

We say that $\tilde{x}_i$ matches x_i, where $\tilde{x}_i$ and x_i are as above, if the requests $\tilde{x}_i$ is the translation of the method invocation x_i, as specified in the simulation process above. In a similar way, we can say that a response $\tilde{y}_i$ matches y_i.

Relation between Jinja runs and TM runs. Now we are ready to relate the runs of the corresponding Jinja programs and Turing machine systems, as introduced above.

Lemma F.24. *For every random input u and every security parameter η (and $A_i, \tilde{A}_i, \ldots$ as above) we have:*

(a) $s_i|_E = t_i|_E$ *and* $t_i|_T = s_{i+1}|_T$,

(b) $\tilde{x}_i$ *matches* x_i,

(c) $\tilde{y}_i$ *matches* y_i,

(d) $\tilde{s}_i \models s_i|_E$,

(e) $\tilde{t}_i \models t_i|_T$,

Item (a) states that sub-states of E and T are separated (the execution of A_i does not changes what T can access and the execution of B_i does not change what E can access).

Items (b) and (c) state that the components E and T in the Jinja run and the corresponding components in the TM system exchange exactly the same data, up to the provided translation.

Item (d) states that M_E correctly simulates E (which is given by the definition of M_E).

Item (e) states that the Jinja+ program T is functionally equivalent to the corresponding Turing Machine M_T. In particular, for the same input, $\mathscr{R}$ produces the same data as Real-Sig and $\mathscr{S}|\mathscr{F}$ produces the same data as $S \cdot$ Ideal-Sig$^-$. This is given by the definition of these systems.

In the reasoning below, we leverage the fact that, without loss of generality, we can assume that E, when connected with T, never makes requests to T that fail (i.e. never makes method calls that return `null`). This is because E can compute the expected size of the output message (recall that we assumed that the length of a plaintext and a corresponding ciphertext are polynomially related). Therefore E can predict potential failure and avoid requests that would fail (E does not lose any information by not executing these requests, as it knows the result up front).

Now, we can observe that a direct consequence of the above lemma (more precisely, of the fact that $\tilde{s}_n \models s_n|_E$) is that the final value of variable `result` in ρ and $\tilde{\rho}$ is the same and, therefore, these (finite) runs output the same result. As it holds for all random input u and all security parameters η, up to some negligible function, the system $E \cdot$ Real-Sig outputs true with the same probability the system $M_E|\mathscr{R}$ outputs 1 and the system $E \cdot S \cdot$ Ideal-Sig outputs true with exactly the same probability the system $M_E|\mathscr{S}|\mathscr{F}$ outputs 1. Now, as we know that $M_E|\mathscr{R} \equiv M_E|\mathscr{S}|\mathscr{F}$, it follows that the probability that true is output by $E \cdot$ Real-Sig and by $E \cdot S \cdot$ Ideal-Sig is the same up to some negligible value.

□

F.5.2. Proof of Lemma F.22

As in the realization proof for public-key encryption, we take an $I_{\mathsf{CryptoLibSig}}$-environment E and we construct a $I^-_{\mathsf{CryptoLibSig}}$-environment E' which consists of (1) a copy of the class `Verifier` (renamed as `VerifierCorr`), (2) a wrapper class `VerifierWrapper` providing unified access to corrupted and uncorrupted verifiers, and (3) an appropriately aligned copy of E (as in Appendix F.4.2). Using this construction, we obtain results analogous to Lemmas F.19 and F.20:

Lemma F.25. $E \cdot$ Real-Sig $\equiv_{\mathsf{comp}}$ $E' \cdot$ Real-Sig

Lemma F.26. $E \cdot S \cdot$ Ideal-Sig $\equiv_{\mathsf{comp}}$ $E' \cdot S \cdot$ Ideal-Sig$^-$

From Lemma F.23 we know that Real-Sig $\leq^{I^-_{PKISig}}$ Ideal-Sig$^-$, i.e. there exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I^-_{PKISig}-environment E' we have (Section 2.3):

$$E' \cdot \text{Real-Sig} \equiv_{\mathsf{comp}} E' \cdot S \cdot \text{Ideal-Sig}^- \tag{F.19}$$

Therefore we obtain

$$
\begin{aligned}
E \cdot \mathsf{Real\text{-}Sig} &\overset{\text{Lemma F.25}}{\equiv_{\mathsf{comp}}} E' \cdot \mathsf{Real\text{-}Sig} \\
&\overset{(F.19)}{\equiv_{\mathsf{comp}}} E' \cdot S \cdot \mathsf{Ideal\text{-}Sig}^{-} \\
&\overset{\text{Lemma F.26}}{\equiv_{\mathsf{comp}}} E \cdot S \cdot \mathsf{Ideal\text{-}Sig} \qquad \text{(F.20)}
\end{aligned}
$$

which completes the proof of Lemma F.22.

Bibliography

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *SAS*, pages 100–115, 2004.

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[ABB+14] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, number 8471 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 2014.

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[ACAE09] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Abstract Certification of Global Non-interference in Rewriting Logic. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium (FMCO 2009). Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2009.

[ACW13] Mathilde Arnaud, Véronique Cortier, and Cyrille Wiedling. Analysis of an electronic boardroom voting system. In *E-Voting and Identify - 4th International Conference, Vote-ID 2013, Guildford, UK*, pages 109–126, 2013.

[Adi08] Ben Adida. Helios: Web-based Open-Audit Voting. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

[AdMPQ09] Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jaques Quisquater. Electing a University President Using Open-Audit Voting: Analysis of Real-World

Use of Helios. In *USENIX/ACCURATE Electronic Voting Technology (EVT 2009)*, 2009.

[AGJ11] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 331–340. ACM, 2011.

[AGJ12] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 712–723. ACM, 2012.

[AKBW14] Claudia Z. Acemyan, Philip T. Kortum, Michael D. Byrne, and Dan S. Wallach. Usability of voter verifiable, end-to-end voting systems: Baseline data for helios, prêt à voter, and scantegrity II. In *2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections, EVT/WOTE '14*. USENIX Association, 2014.

[ANL03] N. Asokan, Valtteri Niemi, and Pekka Laitinen. On the Usefulness of Proof-of-Possession. In *Proceedings of the 2nd Annual PKI Research Workshop*, pages 122–127, 2003.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext Recovery Attacks against SSH. In *IEEE Symposium on Security and Privacy (S&P 2009)*, pages 16–26. IEEE Computer Society, 2009.

[BBB+13] Susan Bell, Josh Benaloh, Mike Byrne, Dana DeBeauvoir, Bryce Eakin, Gail Fischer, Philip Kortum, Neal McBurnett, Julian Montoya, Michelle Parker, Olivier Pereira, Philip Stark, Dan Wallach, , and Michael Winn. STAR-Vote: A Secure, Transparent, Auditable, and Reliable Voting System. *USENIX Journal of Election Technology and Systems (JETS)*, 1:18–37, August 2013.

[BBF+08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008*, pages 17–32. IEEE Computer Society, 2008.

[BC02] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCG+15] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions.

In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA*, pages 499–516. IEEE Computer Society, 2015.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In H. Krawczyk, editor, *Advances in Cryptology, 18th Annual International Cryptology Conference (CRYPTO 1998)*, volume 1462 of *Lecture Notes in Computer Science*, pages 549–570. Springer, 1998.

[BDR04] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, pages 100–114. IEEE Computer Society, 2004.

[Bec00] Bernhard Beckert. A dynamic logic for Java Card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000.

[Ben06] Josh Benaloh. Simple verifiable elections. In *2006 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'06, Vancouver, BC, Canada*, 2006.

[BFG10] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 445–456. ACM, 2010.

[BFG+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–206, 2014.

[BFK+13] Karthikeyan Bhargavan, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with Verified Cryptographic Security. In *IEEE Symposium on Security and Privacy (S&P 2013)*. IEEE Computer Society, 2013.

[BGH+16] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. On Improvements Of Low-Deterministic Security. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust , POST 2016*, pages 68–88. Springer Berlin Heidelberg, 2016.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[BGP11] Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with Helios. In *USENIX/ACCURATE Electronic Voting Technology (EVT 2011)*, 2011.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in Lecture Notes in Computer Science. Springer, 2007.

[BJST08] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally Sound Mechanized Proofs for Basic and Public-key Kerberos. In Masayuki Abe and Virgil D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2008)*, pages 87–99. ACM, 2008.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA*, 2016.

[Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001.

[Bla06] Bruno Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy (S&P 2006)*, pages 140–154. IEEE Computer Society, 2006.

[BMR93] Alexander Borgida, John Mylopoulos, and Raymond Reiter. "... and nothing else changes": The frame problem in procedure specifications. In Victor R. Basili, Richard A. DeMillo, and Takuya Katayama, editors, *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, pages 303–314. IEEE Computer Society / ACM Press, 1993.

[BMU10] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 387–398. ACM, 2010.

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. *IEEE Symp. on Security and Privacy*, pages 339–353, 2008.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption — How to encrypt with RSA. In Alfredo De Santis, editor, *Advances in Cryptology, EUROCRYPT 1994, Workshop on the Theory and Application of Cryptographic Techniques*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 1995.

[BR96] Mihir Bellare and Phillip Rogaway. The Exact Security of Digital Signatures - How to Sign with RSA and Rabin. In *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, pages 399–416, 1996.

[BT94] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing (STOC 1994)*, pages 544–553. ACM Press, 1994.

[Can00] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Technical Report 2000/067, Cryptology ePrint Archive, 2000. Available at `http://eprint.iacr.org/2000/067` with new versions from December 2005 and July 2013.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 136–145. IEEE Computer Society, 2001.

[Can04] Ran Canetti. Universally Composable Signature, Certification, and Authentication. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004)*, pages 219–233. IEEE Computer Society, 2004.

[CB12] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*, pages 65–74, 2012.

[CB13] David Cadé and Bruno Blanchet. Proved Generation of Implementations from Computationally Secure Protocol Specifications. In David A. Basin and John C. Mitchell, editors, *Principles of Security and Trust - Second International Conference (POST 2013)*, volume 7796 of *Lecture Notes in Computer Science*, pages 63–82. Springer, 2013.

[CCC+08] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily Shen, and Alan T. Sherman. Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems using Invisible Ink Confirmation Codes. In *USENIX/ACCURATE Electronic Voting Technology (EVT 2008)*. USENIX Association, 2008. See also `http://www.scantegrity.org/elections.php`.

[CCC+10] Richard Carback, David Chaum, Jeremy Clark, John Conway, Aleksander Essex, Paul S. Herrnson, Travis Mayberry, Stefan Popoveniuc, Ronald L. Rivest, Emily Shen, Alan T. Sherman, and Poorvi L. Vora. Scantegrity II Municipal Election at Takoma Park: The First E2E Binding governmental Elecion with Ballot Privacy. In *USENIX Security Symposium/ACCURATE Electronic Voting Technology (USENIX 2010)*. USENIX Association, 2010.

[CCFG16] Pyrros Chaidos, Véronique Cortier, Georg Fuchsbauer, and David Galindo. BeleniosRF: A Non-interactive Receipt-Free Electronic Voting Scheme. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a Secure Voting System. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 354–368. IEEE Computer Society, 2008.

[CD09] Sagar Chaki and Anupam Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 172–185. IEEE Computer Society, 2009.

[CDD+17] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. Machine-Checked Proofs of Privacy for Electronic Voting Protocols. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 993–1008, 2017.

[CEK+15] Véronique Cortier, Fabienne Eigner, Steve Kremer, Matteo Maffei, and Cyrille Wiedling. Type-based verification of electronic voting protocols. In *Principles of Security and Trust - 4th International Conference, POST*, 2015.

[CGGI14] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Election Verifiability for Helios under Weaker Trust Assumptions. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, LNCS, Wroclaw, Poland, 2014. Springer.

[CGK+16] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. SoK: Verifiability Notions for E-Voting Protocols. In *IEEE 37th Symposium on Security and Privacy (S&P 2016)*, pages 779–798. IEEE Computer Society, 2016.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. In *Advances in Cryptology — EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1233 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Cha81] David Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[Com12] `http://www.computerworld.com/s/article/9233058/Election_watchdogs_keep_wary_eye_on_paperless_e_voting_systems`, October 30th 2012.

[CP92] David Chaum and Torben P. Pedersen. Wallet Databases with Observers. In *12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, 1992.

[CRST14] Chris Culnane, Peter Y. A. Ryan, Steve Schneider, and Vanessa Teague. vVote: a Verifiable Voting System (DRAFT). *CoRR*, abs/1404.6822, 2014. Available at `http://arxiv.org/abs/1404.6822`.

[CS11] Véronique Cortier and Ben Smyth. Attacking and Fixing Helios: An Analysis of Ballot Secrecy. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France*, pages 297–311, 2011.

[CS14] Chris Culnane and Steve A. Schneider. A peered bulletin board for robust use in verifiable voting systems. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 169–183. IEEE, 2014.

[CW17] Véronique Cortier and Cyrille Wiedling. A formal analysis of the Norwegian E-voting protocol. *Journal of Computer Security*, 25(1):21–57, 2017.

[DFK+17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *IEEE Symposium on Security and Privacy, San Jose, CA, USA*, pages 463–482, 2017.

[DLM82] Richard A. DeMillo, Nancy A. Lynch, and Michael Merritt. Cryptographic Protocols. In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 1982)*, pages 383–400. ACM, 1982.

[Don15] Don Syme and Adam Granicz and Antonio Cisterino. *Expert F# 4.0*. Apress, 2015.

[DY83] Danny Dolev and Andrew C. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 341–350. ACM, 2011.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, April 1979.

[FOPS01] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP Is Secure under the RSA Assumption. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2001.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, pages 186–194, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[GHM13] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2013.

[GM82a] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GM82b] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–377, 1982.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[GP05] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic Protocol Analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, volume 5, pages 363–379. Springer, 2005.

[GRCC15] Gurchetan S. Grewal, Mark Dermot Ryan, Liqun Chen, and Michael R. Clarkson. Du-Vote: Remote Electronic Voting with Untrusted Computers. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 155–169, 2015.

[GS15] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, 2015.

[Har84] David Harel. Dynamic logic. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Co., Dordrecht, 1984.

[HKHB14] Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In Gianluigi Zavattaro Elvira Albert, Emil Sekerinski, editor, *Proceedings of the 11th International Conference on Integrated Formal Methods*, LNCS, pages 55–70. Springer, September 2014.

[Hoa69] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 1969.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory, Third International Symposium,*

ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings, pages 267–288, 1998.

[HS09a] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[HS09b] Christian Hammer and Gregor Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

[HUMQ09] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. Polynomial Runtime and Composability. Technical Report 2009/023, Cryptology ePrint Archive, 2009. Available at `http://eprint.iacr.org/2009/023`.

[Jos87] Josh D. C. Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, 1987.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KJRM16] Burt Kaliski, Jakob Jonsson, Andreas Rusch, and Kathleen M. Moriarty. PKCS #1 RSA Cryptography Specifications Version 2.2. Technical report, RFC Editor, 2016. `https://www.rfc-editor.org/info/rfc8017`.

[KKO+11] Fatih Karayumak, Michaela Kauer, Maina M. Olembo, Tobias Volk, and Melanie Volkamer. User Study of the Improved Helios Voting System Interfaces. In *1st Workshop on Socio-Technical Aspects in Security and Trust, STAST 2011*, pages 37–44. IEEE, 2011.

[KMST16a] Ralf Küsters, Johannes Müller, Enrico Scapin, and Tomasz Truderung. sElect: A Lightweight Verifiable Remote Voting System. In *IEEE 29th Computer Security Foundations Symposium (CSF 2016)*, pages 341–354. IEEE Computer Society, 2016.

[KMST16b] Ralf Küsters, Johannes Müller, Enrico Scapin, and Tomasz Truderung. sElect: A Lightweight Verifiable Remote Voting System. Technical report, Cryptology ePrint Archive, Report 2016/438, 2016. Available at `http://eprint.iacr.org/2016/438`.

[KN06] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[KOKV11] Fatih Karayumak, Maina M. Olembo, Michaela Kauer, and Melanie Volkamer. Usability Analysis of Helios - An Open Source Verifiable Remote Electronic Voting System. In Hovav Shacham and Vanessa Teague, editors, *2011 Electronic*

Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '11. USENIX Association, 2011.

[KR17] Ralf Küsters and Daniel Rausch. A framework for universally composable diffie-hellman key exchange. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA*, pages 881–900, 2017.

[KSTG14a] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014*, volume 8414 of *Lecture Notes in Computer Science*, pages 220–239. Springer, 2014. A full version is available at http://eprint.iacr.org/2014/038.

[KSTG14b] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. Technical Report 2014/38, IACR Cryptology ePrint Archive, 2014. Available at `http://eprint.iacr.org/2014/038`.

[KT08a] Ralf Küsters and Max Tuengerthal. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 270–284. IEEE Computer Society, 2008.

[KT08b] Ralf Küsters and Max Tuengerthal. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. Technical Report 2008/006, Cryptology ePrint Archive, 2008. Available at `http://eprint.iacr.org/2008/006`.

[KT09] Ralf Küsters and Max Tuengerthal. Universally Composable Symmetric Encryption. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 293–307. IEEE Computer Society, 2009.

[KT11a] Ralf Küsters and Max Tuengerthal. Composition Theorems Without Pre-Established Session Identifiers. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 41–50. ACM, 2011.

[KT11b] Ralf Küsters and Max Tuengerthal. Ideal Key Derivation and Encryption in Simulation-based Security. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011, The Cryptographers' Track at the RSA Conference 2011, Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2011.

[KT13] Ralf Küsters and Max Tuengerthal. The IITM Model: a Simple and Expressive Model for Universal Composability. Technical Report 2013/025, Cryptology ePrint Archive, 2013. Available at `http://eprint.iacr.org/2013/025`.

[KT14] Ralf Küsters and Tomasz Truderung. Security in E-Voting. *it - Information Technology*, 56(6):300–306, 2014.

[KTB+15] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A Hybrid Approach for Proving Noninterference of Java Programs. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015*, pages 305–319. IEEE, 2015.

[KTG12a] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 198–212. IEEE Computer Society, 2012.

[KTG12b] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. Cryptology ePrint Archive, Report 2012/153, 2012. `http://eprint.iacr.org/2012/153`.

[KTV10a] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. A Game-based Definition of Coercion-Resistance and its Applications. In *23th IEEE Computer Security Foundations Symposium, CSF 2010*, pages 122–136. IEEE Computer Society, 2010.

[KTV10b] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountability: Definition and Relationship to Verifiability. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 526–535. ACM, 2010.

[KTV10c] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Proving Coercion-Resistance of Scantegrity II. In Miguel Soriano, Sihan Qing, and Javier López, editors, *Proceedings of the 12th International Conference on Information and Communications Security (ICICS 2010)*, volume 6476 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2010.

[KTV11] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *32nd IEEE Symposium on Security and Privacy (S&P 2011)*, pages 538–553. IEEE Computer Society, 2011.

[KTV12a] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. A Game-Based Definition of Coercion-Resistance and its Applications. *Journal of Computer Security (special issue of selected CSF 2010 papers)*, 20(6/2012):709–764, 2012.

[KTV12b] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Clash Attacks on the Verifiability of E-Voting Systems. In *33rd IEEE Symposium on Security and Privacy (S&P 2012)*, pages 395–409. IEEE Computer Society, 2012.

[Küs06] Ralf Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proceedings of the 19th IEEE Computer Security Foundations*

Workshop (CSFW-19 2006), pages 309–320. IEEE Computer Society, 2006. See http://eprint.iacr.org/2013/025/ for a full and revised version.

[KZZ15a] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. DEMOS-2: scalable E2E verifiable elections without random oracles. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 352–363, 2015.

[KZZ15b] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. End-to-end verifiable elections in the standard model. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 468–498. Springer, 2015.

[KZZ17] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. An efficient E2E verifiable e-voting system without setup assumptions. *IEEE Security & Privacy*, 15(3):14–23, 2017.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[Loc12] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[Man00] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 185–199, 2000.

[McL96] John McLean. A General Theory of Composition for a Class of "Possibilistic" Properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[MCN+01] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantien Zheng, and Steve Zdancewic. *JIF: Java Information Flow (software release)*, July 2001. http://www.cs.cornell.edu/jif/.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Mih98] Mihir Bellare and Phillip Rogaway. PSS: Provably Secure Encoding Method for Digital Signatures. Technical report, IEEE Working Group, 1998. Available at http://grouper.ieee.org/groups/1363/.

[Mih04] Mihir Bellare and Phillip Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. *IACR Cryptology ePrint Archive*, 2004:331, 2004.

[MN06] Tal Moran and Moni Naor. Receipt-Free Universally-Verifiable Voting With Everlasting Privacy. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 373–392. Springer, 2006.

[MS10] Heiko Mantel and Henning Sudbrock. Flexible Scheduler-Independent Security. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2010.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.

[New16] `http://nymag.com/daily/intelligencer/2016/11/activists-urge-hillary-clinton-to-challenge-election-results.html`, November 22th 2016.

[New17] `https://www.nytimes.com/2017/06/01/world/europe/vladimir-putin-donald-trump-hacking.html`, June 1st 2017.

[NORV14] Stephan Neumann, Maina M. Olembo, Karen Renaud, and Melanie Volkamer. Helios Verification: To Alleviate, or to Nominate: Is That the Question, or Shall we Have Both? In Andrea Ko and Enrico Francesconi, editors, *Electronic Government and the Information Systems Perspective - Third International Conference, EGOVIS 2014. Proceedings*, volume 8650 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2014.

[NvO98] Tobias Nipkow and David von Oheimb. Java_{light} is Type-Safe — Definitely. In *POPL*, pages 161–170, 1998.

[OBV13] Maina M. Olembo, Steffen Bartsch, and Melanie Volkamer. Mental Models of Verifiability in Voting. In James Heather, Steve A. Schneider, and Vanessa Teague, editors, *E-Voting and Identify - 4th International Conference, Vote-ID 2013*, volume 7985 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 2013.

[Oka97] Tatsuaki Okamoto. Receipt-Free Electronic Voting Schemes for Large Scale Elections. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Proceedings of the 5th International Workshop on Security Protocols*, volume 1361 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 1997.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[PCW14] `http://www.pcworld.com/article/2154000/estonian-electronic-voting-system-vulnerable-to-attacks-researches-say.html` and `https://estoniaevoting.org/`, May 12th 2014.

[PHN12] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In *Certified Programs and Proofs - Second International Conference, CPP 2012*, pages 109–125, 2012.

[PHN13] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Noninterfering schedulers - when possibilistic noninterference implies probabilistic noninterference. In *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, pages 236–252, 2013.

[PW01] Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *IEEE Symposium on Security and Privacy*, pages 184–201. IEEE Computer Society, 2001.

[RBH+10] Peter Y. A. Ryan, David Bismark, James Heather, Steve Schneider, and Zhe Xia. The Prêt à Voter Verifiable Election System. Technical report, University of Luxembourg, University of Surrey, 2010. `http://www.pretavoter.com/publications/PretaVoter2010.pdf`.

[RRI16] Peter Y. A. Ryan, Peter B. Roenne, and Vincenzo Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, 2016.

[RS07] Ronald L. Rivest and Warren D. Smith. Three Voting Protocols: ThreeBallot, VAV and Twin. In *USENIX/ACCURATE Electronic Voting Technology (EVT 2007)*, 2007.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[RWW94] A. William Roscoe, Jim Woodcock, and Lars Wulf. Non-interference through determinism. In *Computer Security - ESORICS 94, Third European Symposium on Research in Computer Security, Brighton, UK*, pages 33–53, 1994.

[Sab01] Andrei Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*, pages 225–239, 2001.

[Sab03] Andrei Sabelfeld. Confidentiality for Multithreaded Programs via Bisimulation. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, pages 260–274, 2003.

[Sca15a] Enrico Scapin. A Proof Technique for Noninterference In Open Systems. Workshop on Foundations of Computer Security (FCS), 2015. Affiliated with IEEE CSF 2015, `http://software.imdea.org/~bkoepf/FCS15/`.

[Sca15b] Enrico Scapin. A Proof Technique for Noninterference In Open Systems. Technical report, University of Trier, 2015.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley& sons, New York, 1996.

[Sch14] Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[SFD+14] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. Security Analysis of the Estonian Internet Voting System. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 703–715. ACM, 2014.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[SHM17] Enrico Scapin, Mihai Herda, and Martin Mohr. Verification of the Mix Server of sElect, 2017. Available at `https://github.com/escapin/MixServerVerification`.

[SK95] Kazue Sako and Jue Kilian. Receipt-Free Mix-Type Voting Scheme — A practical solution to the implementation of a voting booth. In *Advances in Cryptology — EUROCRYPT '95, International Conference on the Theory and Application of Cryptographic Techniques*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer-Verlag, 1995.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, 2003.

[Smi06] Geoffrey Smith. Improved typings for probabilistic noninterference in a multithreaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE 13th Computer Security Foundations Workshop, CSFW 2000*, pages 200–214. IEEE, 2000.

[SS11] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 27–47, 2011.

[SS12] Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In *Formal Verification of Object-Oriented Software*, LNCS 7421, pages 232–249. Springer, 2012.

[SS13] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[SS16] Enrico Scapin and William Standard. An Election Manager for sElect, 2016. Available at `https://github.com/escapin/ElectionManager`.

[SST16] Enrico Scapin, William Standard, and Tomasz Truderung. sElect: Secure and Simple Elections, 2016. Available at `https://github.com/escapin/sElect`.

[STB+14a] Enrico Scapin, Tomasz Truderung, Daniel Bruns, Michael Kirsten, and Martin Mohr. An E-Voting Machine with Auditing Procedures, 2014. Available at `https://github.com/escapin/EVotingMachine`.

[STB+14b] Enrico Scapin, Tomasz Truderung, Daniel Bruns, Christoph Scheben, and Jürgen Graf. E-voting Case Studies, 2014. Available at `https://github.com/escapin/EVotingVerif`.

[STG13] Enrico Scapin, Tomasz Truderung, and Jürgen Graf. A Cloud Storage System with strong cryptographic guarantees, 2013. Available at `https://github.com/escapin/CloudStorageSystem`.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA*, pages 355–364, 1998.

[TSK13] Tomasz Truderung, Enrico Scapin, and Andreas Koch. A Library of Cryptographic Operations: A Real and Ideal Java Implementation, 2013. Available at `https://github.com/escapin/CVJFunct`.

[VP17] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 24nd ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[VS97] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 156–169, 1997.

[VS99] Dennis Volpano and Geoffrey Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security*, 7(1), 1999.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[Was10] Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

[WH09] Janna-Lynn Weber and Urs Hengartner. Usability Study of the Open Audit Voting System Helios, 2009. Available at `http://www.jannaweber.com/wp-content/uploads/2009/09/858Helios.pdf`.

[WL10] Daniel Wasserrab and Denis Lohner. Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing. In *6th International Verification Workshop - VERIFY-2010*, July 2010.

[WLS09] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-Based Noninterference and its Modular Proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*, pages 31–44. ACM, June 2009.

[Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 80–91, 1982.

[ZCC+13] Filip Zagórski, Richard Carback, David Chaum, Jeremy Clark, Aleksander Essex, and Poorvi L. Vora. Remotegrity: Design and Use of an End-to-End Verifiable Remote Voting System. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013*, volume 7954 of *Lecture Notes in Computer Science*, pages 441–457. Springer, 2013.

[ZM03] Steve Zdancewic and Andrew C. Myers. Observational Determinism for Concurrent Program Security. In *IEEE Computer Security Foundations Workshop - CSFW*, pages 29–. IEEE Computer Society, 2003.

Academic Curriculum

January 2017 - January 2018	**University of Stuttgart, Germany.** Ph.D. student at the Institute of Information Security. Ph.D. supervisor: Prof. Dr. Ralf Küsters.
December 2012 - December 2016	**University of Trier, Germany.** Ph.D. student at the Chair for Information Security and Cryptography. Ph.D. supervisor: Prof. Dr. Ralf Küsters.
October 2010 - October 2012	**University of Verona, Italy.** Master of Science in "Ingegneria e Scienze Informatiche". Thesis: Field-Sensitive Unreachability and Non-Cyclicity Analysis, supervised by Prof. Dr. Nicola Fausto Spoto.
January 2010 - June 2010	**Queen Mary University of London, UK.** Exchange semester as Erasmus student.
September 2006 - October 2010	**University of Verona, Italy.** Bachelor of Science in "Informatica Multimediale". Final Project: Architectures and Implementations of Single Sign-On Systems, supervised by Prof. Dr. Luca Viganò.