A Verifiable Language for Cryptographic Protocols

Nielsen, Christoffer Rosenkilde; Nielson, Flemming; Nielson, Hanne Riis

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

A Verifiable Language for Cryptographic Protocols

Christoffer Rosenkilde Nielsen

Kongens Lyngby 2008
IMM-PHD-2008-210

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

We develop a formal language for specifying cryptographic protocols in a structured and clear manner, which allows verification of many interesting properties; in particular confidentiality and integrity. The study sheds new light on the problem of creating intuitive and human readable languages, that are analysable with respect to interesting properties. Furthermore it motivates and is an example of, a novel, more general methodology of language design by first verbosely describing the semantics in a mathematical language, e.g. a logic, then restricting the properties of interest to be computable, and finally systematically transforming it into a more intuitive specification language, maintaining this tractability.

Resumé

Vi udvikler et formelt sprog til at formulere kryptografiske protokoller på en strukturet og let gennemskuelig måde, der samtidig tillader verifikation af flere interessante egenskaber – navnligt fortrolighed og integritet. Studiet kaster nyt lys på problemet med at konstruere intuitive og menneskeligt læselige sprog, der kan analyseres med henblik på attraktive egenskaber. Desuden motiverer det – og er et eksempel på – en nyskabende, mere generel tilgang til sprogdesign, ved først at udførligt beskrive semantikken i et matematisk sprog, såsom en logik, således at de interessante egenskaber kan beregnes, og derefter systematisk transformere det til et mere intuitivt specifikationssprog, der bevarer denne beregnelighed.

Preface

Most of the work behind this dissertation has been carried out independently and I take full responsibility of its contents. However, three years is a long time and many of the ideas and results are due to inspirations and benefactors.

The idea behind the iterative framework originates from my many conversations with Professor Helmut Seidl during his stay at DTU in 2005. In collaboration with Professor Flemming Nielson and Professor Hanne Riis Nielson, the idea was matured in [88] and this article serves as the basis of Chapter 2. Several parts of the presentation in the chapter, however, has been influenced by my discussions with Thomas Gawlitza, in particular Example 2.20.

Chapter 3 is based on the ideas described in [87]. Nevertheless, the presentation here has been redeveloped radically since the publication of that article, and contains much new material; including the solution to the problem described in §3.5.3 which Professor Flemming Nielson helped me to solve.

The remaining chapters present previously unpublished work, produced exclusively for this dissertation. The syntax and semantics of CryptoKlaim has benefitted from many private discussions with Henrik Pilegaard as well as some with Professor Flemming Nielson and Sebastian Nanz.

Acknowledgements I thank my supervisors, Professor Flemming Nielson and Professor Hanne Riis Nielson, for the opportunity and their supervision. Furthermore, I want to thank the current and former members of the LBT-group: Jörg Kreiker, Han Gao, Henning Makholm, Sebastian Nanz, Henrik Pilegaard, Christian W. Probst, Matthieu Queva, Nataliya Skrypnyuk, Terkel Tolstrup,

Fan Yang, Ender Yüksel, and Ye Zhang, for providing a friendly and interesting working environment. A special thanks goes to Henrik Pilegaard for spending numerous hours discussing nitpicking topics with me – as well as being a good friend.

I would also like to thank Esben Heltoft Andersen and my brother Johan Sebastian Rosenkilde Nielsen for proof-reading the dissertation. I am also grateful for Johan's many hours of help with the implementation of the iterative framework and the analysis – it would have taken me twice the time and ten times the frustration.

I thank Professor Helmut Seidl and his group, as well as Jörg Kreiker, at Technische Universität München for their hospitality and friendliness towards me during my two-months stay there. I would also like to express my gratitude to the employees of the company, which for confidentiality reasons I cannot name, for their help and for my case study.

Finally, I am in debt to family and friends for their everlasting patience and support, and to Nina – the love of my life – who I needed more during these three years than ever. Thank you!

Contents

Chapter 1

Introduction

Building modern IT-systems is often a demanding task, relying on the collaboration between several hundreds or even thousands of individuals with different backgrounds. Nowadays, it is rarely the case that the systems are developed in the dusk of someone's basement, and that changes many aspects of the whole development phase. System design and software engineering is becoming increasingly important, not only to establish a common ground for communication, but also to ensure that the outcome of the development is the intended result.

The design of systems in specific has spawned an entire movement in standardisation, formalisation, and terminology. The key to successful software development seems to be dividing the project into sub-tasks, such that the individual developer (or the responsible group) can focus efforts on a single task and abstract away from the other parts of the project. Naturally, this also increases the demands to common terminology, proper interface, and compatibility. Many approaches help in these aspects, including object oriented design, UML diagrams, coding conventions, formal standards, etc.

The introduction of a design phase, however, raises a somewhat orthogonal issue to the implementation; namely verification of the design. If the original system design is flawed then the quality of the implementation is of little importance. To many, this may sound both obvious and at the same time a trivial task, but it often, if not to say usually, turns out not to be the case. The strongest

arguments for this statement is probably from distributed systems, where flaws in the design can be disastrous. It turns out that more than 70% of the systems used in our everyday life and rely on to be secure, contain fundamental design flaws which can lead to unwanted behaviour such as break of chain of trust or theft of confidential information [48].[1]

In general, flaws in systems such as stock exchanges, e-banking, missile control, electronic voting systems, or similar are catastrophic. However, even flaws which are eventually discovered before release of the software, can be costly. Studies show that the cost of error-correction increases dramatically the later in the development stages the flaw is found; in particular, the cost of correction, relative to discovery in the design phase, is 7 in the implementation phase, 15 during testing, and 100 during maintenance [67]. It is therefore reasonable to demand that verification is done at the design phase and, as human error is too big a factor to be ignored, in a mathematically provable correct manner.

But in order to attack the problem mathematically, the design must relate to a mathematical model. Moreover, as the system being designed is supposed to be executed in a dynamically changing environment (that is, the real world), we must have formal meanings of execution and environment. A person with programming experience notices that this sounds like the description of language. In fact, the language-based approach to model and verify communication systems has, over time, turned out to be a successful approach, and increasingly more systems are being proven secure this way at the design phase.

The topic of this dissertation is on the formal development of one such language for system design validation. More precisely, we shall introduce a language for designing the way distributed, concurrent systems act and establish secure communication. Usually, secure communication over an insecure medium (such as the internet) is established through cryptography, and, in particular, by the use of cryptographic protocols. These protocols are abstract descriptions of how the participants in a concurrent system must act in order to achieve a common goal. The description is abstract in the sense that it does not dictate implementation details or design patterns, and often even the specifics about the cryptographic algorithms or communication medium are omitted; they are all considered orthogonal issues to the protocol. This conforms to the design abstraction paradigm; the protocol abstracts from details unrelated to the task. Unfortunately, however, such descriptions are, albeit succinct and easy to read, informal, ambiguous and often require a fair amount of interpretation. In other words, they do not pose a reliable foundation for a mathematical investigation. This motivates the development of a formal language that can be used to unify the protocol design and verification phases.

[1] Study based on 214 American financial institutions

It follows that the language must allow program development in a way that the program's design can be easily or, preferably, automatically validated. At the same time, the language should be simple and intuitive, and knowledge of the underlying verification method or technique should not be a precondition of use. In fact, the syntax of the language should stay as close as possible to well-established informal protocols descriptions, in order to ensure that programs in the language remains comprehensible and useful to both the protocol designer and the engineer implementing the system; "if it works, don't fix it". These are ambitious goals and to achieve them we shall rely on much related work from the last three decades on this topic. In the remainder of this chapter, we shall present the background for, and related work on, distributed systems design and analysis. We will also state the contributions of the dissertation and outline the contents of the forthcoming chapters.

1.1 Design Goals: What is Secure?

Secure is one of the most popular buzz-words relating to distributed systems. Usually it is used in an informal manner, implying that the systems "are safe against intrusion" or "will never crash".

Formally, however, we describe security through a number of properties. There are several different properties, and the formulation often vary, but in this dissertation we will use Gollman's definition [55]:

- *Confidentiality*: prevention of unauthorised disclosure of information;
- *Integrity*: prevention of unauthorised modification of information;
- *Availability*: prevention of unauthorised withholding of information or resources:

When we discuss communication systems some are added to the list in addition to the main ones:

- *Authentication*: prevention of forging communication of information;
- *Non-repudiation*: prevention of deniability of sending information.

These properties are usually attempted obtained through the use of cryptographic operations. But cryptographic operations alone are not enough to guarantee

security. A simple example of this is that an adversary may violate confidentiality even though all communication is encrypted, if it can obtain the decryption key. Another, less trivial, example is that an adversary can violate authentication, if it could take information from an earlier communication and reuse this to forge a new communication; this is usually called a *replay attack*.

The mentioned examples shows that proving security properties of a design is not as trivial as it initially may seem. The design must be shown to satisfy the properties in any thinkable or unthinkable situation; this and especially the latter can prove to be a challenging task.

1.2 Introducing Alice and Bob

In order to discuss the internals of a distributed system, some manner of formalism must be agreed upon. Typically, for cryptographic protocols, the formalism is some version of the *Alice-Bob notation*.[2] This notation technique is a syntax that solely focuses on the order of the actions that must take place, the cryptographic primitives being applied, and an abstraction of the content of each message being sent.

The Alice-Bob notation enforces many simplifications to make the system analysable. Specifically, implementation details such as timing, data and key length, transport layer information, etc., are all omitted. The omission of these details allows for an abstract algebraic view on communication, and builds on the assumption that security of the systems should be validated on both the abstract protocol level and on the implementation level, independently.

Sometimes an insecure protocol may be prove secure once implemented; e.g. two messages may be mistaken for one another in the abstraction, but in the implementation this could never happen. However, the abstraction ensures that the opposite could never happen. In fact, correcting such a flaw in the abstract protocol, will never introduce new flaws in the actual implementation, so in that sense the abstraction is safe. An abstractly validated protocol can only suffer to flaws introduced through the implementation.

[2] In general, cryptographic protocols originate from various sources; e.g. mathematicians, computer scientists, and engineers. Although the presentation relies heavily on the source, the Alice-Bob notation always seem to occur in some form in the document.

1.2.1 Communication

The name Alice-Bob refers to a simplification usually being used in combination with this notation. If a protocol includes an initiator (or sender) and a responder (or recipient) we usually denote them A and B, respectively, and sometimes refer to them as Alice and Bob. Likewise, we often refer to servers as S, certificate authorities as CA, etc., and in general we refer to a participant in a protocol as a principal. This brings protocols on a common ground, making similarities as well as distinctions more apparent.

As already said, a protocol is a structured manner in which each participant must act to obtain a common goal. In a communication system this equals transmission and reception of specifically formatted data, as well as pre-defined operations performed on this data. In the Alice-Bob notation these actions are divided into two layers: the transmitting and receiving data, and the structure of and the operations (usually cryptographic) on the data.

Each step of a protocol specifies a message that should be sent from one principal to another. Hence, intuitively, this can easily be generalised as follows:

$$A \rightarrow B \quad : \quad M$$

Here a message M is being sent from principal A to principal B. This is the Alice-Bob notation, and in it, a protocol is a just numbered sequence of steps, referred to as a *protocol narration*.

1.2.2 Term Language

The basis of cryptographic protocols is, obviously, cryptography. Nevertheless, the threat of breaking cryptography is usually distinguished from the threat of breaking the protocols itself. Thus, a protocol designer may assume that cryptography is working perfectly and focus exclusively on its usage (analogously, the cryptographer will usually analyse the cryptographic algorithms independent of the context that they are being used in). In other words, we shall assume that, if a *cipher text* is obtained by encrypting a *plain text* with an encryption key, then the plain text can only be recovered from the cipher text using the corresponding decryption key. This assumption is usually called *perfect cryptography* and renders security and complexity issues of the various cryptographic algorithms unimportant in this context, as all algorithms are considered secure. Hence we shall only be concerned with two abstract types of cryptography: *symmetric* (or *shared key*, e.g. DES or AES [2, 1]) and *asymmetric* (or *public key*, e.g. RSA

or ElGamal [113, 51]). Later, in §3.5.2, we shall discuss other cryptographic operations as well, but, for now, the two mentioned will suffice.

The assumption of perfect cryptography allows an algebraic formulation of the cryptographic operations. For symmetric cryptography, where encryption and decryption keys are identical, we shall write $\{\!|text|\!\}_K$ for the cipher text obtained by encrypting the plain text *text* with the symmetric key K, and use bold font brackets to keep the notation distinct from set delimiters. The perfect cryptography assumption then dictates that retrieving *text* from $\{\!|text|\!\}_K$, that is, decrypting the latter, requires knowledge of K. For asymmetric cryptography we shall require a key pair, (K^+, K^-), comprised by a public key and a private key (denoted by their superscripts), respectively. We then write $[\!|text|\!]_{K^+}$ for a cipher text, which can only be decrypted using knowledge of K^-.

Often asymmetric cryptography also allow *digital signature* schemes, simply by reversing the applied order of the keys. Thus we may write $[\!|text|\!]_{K^-}$ for the digital signature of *text*, using the private key K^-. It is important to note that a digital signature does not provide any *confidentiality*, i.e. anybody can read the signed content *text*, but it provides non-repudiation on the message, in the form that anybody knowing the corresponding public key K^+, can prove the message was sent by someone knowing the private key K^-.

Key pairs for asymmetric encryption are generated by a single principal, and usually this principal is intended to have exclusive knowledge of the private key, whereas the public key is allowed for everyone to know. These intentions are usually indicated by the syntax and for a key pair generated by A we shall write (Ka^+, Ka^-). Analogously, symmetric keys are usually intended only to be shared by two principals, and thus we shall denote these as well in the name;[3] e.g. A and B are intended to share and have exclusive knowledge of key Kab.

This has basically summarised the key notation. Apart from cryptography, protocols also involve elements such as nonces (randomly generated strings), time stamps, etc. But we are solely interested in the usage of cryptography in the protocol, and we shall not be concerned with neither time or time outs, and thus we shall merely view all of the above as unstructured binary strings of unspecified type. Following this line of thought, we may also consider principal names and symmetric keys as just different unstructured binary strings, and we shall refer to the any such unstructured binary string as a *name* and denote it n. Note that this is a safe assumption, as it means that we cannot distinguish between the different types of data; if we could, it would only increase the security of the protocol. Key pairs must, however, still be handled separately from names, and,

[3] Often the protocol includes a trusted third party (e.g. a server S), and in these protocols it is customary to leave out the third parties name in the keys. Thus, for example, we would write Ka instead of Kas.

1. $A \rightarrow B \;:\; A, Na$
2. $B \rightarrow S \;:\; B, \{A, Na, Nb\}_{Kb}$
3. $S \rightarrow A \;:\; \{B, Kab, Na, Nb\}_{Ka}, \{A, Kab\}_{Kb}$
4. $A \rightarrow B \;:\; \{A, Kab\}_{Kb}, \{Nb\}_{Kab}$

Figure 1.1: The Yahalom protocol

as already mentioned, we shall use the superscripts + and − to distinguish them and denote their relationships.

As a result, a message is basically built up by cryptography from names according to a simple syntax:

$$M \quad ::= \quad n \mid n^{+} \mid n^{-} \mid \langle M_1, \cdots, M_k \rangle \mid \{M_1\}_{M_0} \mid \lfloor M_1 \rfloor_{M_0}$$

This inductive definition of a syntax is usually referred to as an *abstract syntax* or a *term language*. Apart from the primitives introduced above, we also allow concatenation of terms, the so-called *tuples*. For example if a message consists of four concatenated fields M_1, M_2, M_3 and M_4, in that particular order, then we write $\langle M_1, M_2, M_3, M_4 \rangle$. Usually, however, the brackets $\langle\rangle$, are omitted in the Alice-Bob specification, when they are implicit.

The syntax is abstract in the sense that all the basic components of the message are, as described above, coalesced into one group of names. This serves only a theoretical purpose, and in our modelling we shall still employ the naming conventions introduced above; i.e. capital letter for principals, capital K for keys followed by the lowercase name of each of the principals intended to know the key, etc.

Example 1.1 (Yahalom) *An example of the Alice-Bob notation is given in Figure 1.1 where the Yahalom [31] protocol description for symmetric key distribution is presented.*

The protocol introduces three principals; the initiator A, the responder B, and the server S, and relies on the pre-existence of the long term symmetric keys Ka, Kb, shared between the principal denoted in the subscript and the server, all modelled as names.

The goal of the protocol is to introduce a new symmetric key, Kab, shared by A and B, such that after a successful protocol run, these principals may communicate safely. This is obtained through the mutually trusted third party, S. During the four steps of a protocol run the nonces Na and Nb are generated, as well as the new shared key Kab. The intended scope of the shared keys are the

principals noted in the lowercase letters, although the trusted server, S, will also know Kab after generating it. Accordingly, the lowercase letters on the nonces denote the principal generating them.

1.3 Related Work

Before we embark on the task at hand, we shall present some of the related literature on language design for concurrent systems, the so-called *process calculi*, existing languages for modelling cryptographic protocols, and state-of-the-art techniques for verifying security properties of the designs.

1.3.1 Foundational Process Calculi

Process calculus (or *process algebra*) is a formal approach to the study of concurrent processes which uses algebraic structures to model processes and reason about them. It differs from other concurrency models, such as Petri Nets [106], the Actor Model [62], or Input/Output Automata [74], by its linguistic character and, specifically, its algebraic properties.

A process calculus is constituted by two main ingredients:

Syntax. The language is given as a term language with recursive structure, defined inductively by one or more syntactic categories.

Semantics. A process calculus is given formal meaning through a semantics. The semantics describe how a system, represented in the process calculus, may evolve (or *execute*). As in programming languages, three main styles of formal semantics are distinguished: operational, denotational, and axiomatic semantics. The most popular style, operational semantics, exists in two styles: *Structural Operational Semantics* [110] producing labelled transition systems and the *Chemical Abstract Machine* (CHAM) [16] where the semantics is given as a set of simple reduction rules for term rewriting and a structural congruence on terms with equivalent behaviour. The *denotational semantics* were originally developed with the purpose of modelling the meaning of a program as a function that maps input to output, an approach which has been generalised in domain theory. The same approach has been used for process calculi as well, for example on CSP [63], where processes are mapped into their sets of traces, failures, and divergences. Finally, *Axiomatic Semantics* has been used for ACP

[15], which is described by a set of axioms with respect to a simple process algebra syntax.

Originally, process calculi were created with the purpose of serving as a foundational model for concurrent computation, a "λ-calculus of concurrency". History has shown, however, that the nature of processes, systems and communication is much too diverse to capture with a single model. Instead, a wide range of different calculi has been developed, each with the purpose of modelling a particular aspect of the distributed systems, but only few of them can be considered fundamental. Most specialised calculi rely on the same basic concepts and differ in aspects of theoretical character of importance to the verification phase. Hence, we introduce only a few of the most foundational ones, before moving to those of specific relevance to this dissertation.

CCS. The *Calculus of Communicating Systems (CCS)* by Milner [78, 79] is a static, non-distributed model of processes with synchronous communication. A CCS process, P, is given by the abstract syntax:

$$\begin{array}{lcl} P & ::= & 0 \mid \alpha.P \mid P_1 \,|\, P_2 \mid P_1 + P_2 \\ \alpha & ::= & a \mid \bar{a} \mid \tau \end{array}$$

Processes rely syntactically on actions, α, representing atomic execution steps; either internal (not observable) in the case of τ and otherwise externally observable.

The semantics of CCS is given as an operational semantics in the form of a ternary relation, $\rightarrow$, called the labelled transition relation. We say that the process P can *evolve* to P' by action α whenever $\langle P, \alpha, P' \rangle \in \rightarrow$ and write $P \xrightarrow{\alpha} P'$. The semantics is defined inductively where sequential composition $\alpha.P$, nondeterministic choice $P_1 + P_2$, and the terminated process 0 have straightforward inference rules. In the case of parallel composition $P_1 \,|\, P_2$, the branches may execute individually, or *synchronise*:

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\bar{a}} P_2'}{P_1 \,|\, P_2 \xrightarrow{\tau} P_1' \,|\, P_2'}$$

Restriction $P \backslash A$ expresses that actions contained in A can only be used for internal communications within P, and renaming yields a process $P[f]$ in which all actions α of P are relabelled to $f(\alpha)$.[4] Infinite behaviour is achieved by defining a *process identifier* A, via equations $A \triangleq P$ where A occurs recursively in P. Execution of A is then defined as the execution of P by *process invocation*.

[4] If f is the map of a to b we write either $P[a \mapsto b]$ or $P[^b/_a]$. In this dissertation we shall prefer the former.

A value-passing variant of CCS also exists [79]. Here the calculus is extended with expressions e, values v, and variables x, and and actions then work as channels through which values can pass. Prefixes are accordingly redefined to $a(x) \mid \bar{a}\langle e\rangle \mid \tau$ and evaluated using the following new rules:

$$a(x).P \xrightarrow{a\langle v\rangle} P[x \mapsto v] \qquad \bar{a}\langle e\rangle.P \xrightarrow{\bar{a}\langle v\rangle} P, \text{ if } e \text{ evaluates to } v$$

The value-passing variant can be encoded in the basic calculus and it is therefore not more expressive than the above-mentioned calculus. However, in some cases the extension may be preferred, as it allows more elegant and succinct programs than the regular CCS.

CSP. Hoare [63, 64] and Brookes, Hoare, and Roscoe [28] proposed Communicating Sequential Processes (CSP), providing a process model similar to CCS. In fact, the calculi were highly influenced each other. Differences are CSP's concepts of choice (external choice of the environment and nondeterministic choice) as well as that its parallel composition operator does not hide communications and thus allows any number of processes to participate in the same event.

π-calculus. The π-calculus of Milner, Parrow, and Walker [80, 81, 116], is most prominent and influential among the calculi with a notion of mobility. In pure π-calculus mobility is obtained through links, which can be passed around and then used for interaction with other processes. This is achieved syntactically by action prefixes $\pi ::= x(y) \mid \bar{x}\langle y\rangle \mid \tau$, much like the value-passing version of CCS. However, instead of keeping values and actions distinct as in CCS, they are coalesced in the syntactic category *names*. The implication of this change is that communicated names can be used for new communication links after reception. Names are treated based on their *scope* and the restriction $(\nu x)P$ limits the scope of a name x to the process P, similar to restriction in CCS, but the scope of a restriction may change dynamically[5] to allow communication. Infinite behaviour is introduced by the replication operator $!P$, semantically defined by the equivalence rule $!P \equiv P|!P$ of the structural congruence. The use of structural congruence implies that the language uses an operational semantics in CHAM style leading to very concise descriptions. The semantics is given by a binary relation $\rightarrow$ over processes, where the most interesting case is reaction:

$$(x(y).P_1 + M_1) \,|\, (\bar{x}\langle z\rangle.P_2 + M_2) \;\rightarrow\; P_1[y \mapsto z] \,|\, P_2$$

Where M ranges over processes with only prefixing, choice, and termination.

[5] This is called *scope extrusion*, similarly to the λ-calculus [35, 12]. The scope is only changed to make the semantics function properly and is not reflected in any real change of state under program control.

Klaim. The above three mentioned calculi have no notion of distribution; i.e. all processes share a single execution space. The process calculus KLAIM [17] by Bettini et al. differs from this approach in that it extends process algebra with aspects from coordination languages. Coordination languages view concurrent systems as a conjunction of two ingredients [54]: a computation model and a coordination model. The computation model can be described by any sequential calculus and the coordination model by a coordination language, such as Linda [53]. Linda introduces the concept *tuple space* which is a multiset of tuples. Interaction is then modelled through the tuple space by placing and retrieving tuples via pattern matching mechanisms, realising *asynchronous communication*.

KLAIM extends the Linda paradigm by having multiple tuple spaces, which can be accessed remotely. Technically, each tuple space is given a unique identifier, l, called a *locality*. KLAIM also introduces a new syntactic category of nets, N, containing either a located tuple, a located process, or a parallel composition of nets:

$$N \quad ::= \quad l :: P \quad | \quad l :: \langle t \rangle \quad | \quad N1 \parallel N2$$

Processes are built from Linda actions such as $\mathsf{out}\,(t)@l$ and $\mathsf{in}\,(t)@l$ for, respectively, placing and retrieving tuples from localities. KLAIM also includes operators for creating new tuple spaces and to allow distribution of the associated localities, in order to model code mobility and remote execution.

1.3.2 Calculi for Cryptographic Protocols

Focussing on communication in concurrent system and abstracting from practical issues, process calculi seems suitable for the analysis of the design of communication systems or protocols. Especially cryptographic protocols have been researched and used in conjuction with methods for establishing security properties of the systems. In order to express these protocols, specialised calculi have surfaced to provide ways to model cryptographic primitives.

CSP. Modelling cryptographic protocols in a process calculus setting first showed its worth when Lowe successfully found a number of flaws in well-known protocols [70, 71, 72, 73]. Lowe used CSP (see §1.3.1) and expressed encryption of a message m with a key k by the CSP event $Encrypt.m.k$. In Lowe's setting, protocols were analysed against attacks by modelling an attacker that was capable of all actions allowed by the assumption of perfect cryptography, a so-called Dolev-Yao attacker [44], and Lowe then investigated whether any possible execution trace could result in an event that violated a security property (in particular the properties authentication and confidentiality). Later, the CSP-approach has been further developed [115] to successfully cover a variety of

protocols.

Spi Calculus. Abadi and Gordon [7] proposed the Spi Calculus as a variant of the π-calculus with explicit modelling of cryptographic primitives. For example, in the case of symmetric key cryptography, they extend the set of terms of π-calculus with $\{M\}_N$, representing encryption of the message M under key N, where M and N are names. Similarly, processes are extended with a construct for decryption,

$$\mathsf{case}\ L\ \mathsf{of}\ \{x\}_N\ \mathsf{in}\ P$$

and the reduction relation $\rightarrow$ then dictates that only encryptions with the key N can be decrypted:

$$\mathsf{case}\ \{M\}_N\ \mathsf{of}\ \{x\}_N\ \mathsf{in}\ P \quad \rightarrow \quad P[x \mapsto M]$$

The language incorporates similar constructs for asymmetric cryptography.

Initially, Gordon and Jeffrey [58] used correspondence assertions to show that the protocol reached its goals by typing (see §1.3.3). Specifically, they embedded labelled begin and end assertions into the specification, and then proved that every end had to be proceeded by a corresponding begin). This, however, only helped them establish non-repudiation, which is arguably one of the less interesting properties for most cryptographic protocols. Later the same researchers were inspired by Abadi's seminal work on confidentiality by typing [3], and extended the technique to the Spi Calculus [57]. As for CSP, this work was based on a formalisation of an attacker in the Dolev-Yao style.

Applied pi calculus. Another variant of the π-calculus, called the applied pi calculus, was proposed by Abadi and Fournet [5]. The applied pi calculus allows transmission of messages composed of simple function and equations instead of just names. The perfect cryptography assumption is ensured by a careful definition of the equational theory for functions representing cryptographic primitives. The changes loses some of the elegant simplicity of the original calculus, but establishes a robust framework that can cope with a variety of cryptographic operations without extensive encodings. Recently, Bhargavan, Fournet, and Gordon [18] used the applied pi calculus as a basis for a theorem prover-driven (see §1.3.3) protocol verifier for industrial protocols in the SOAP standard.

LySa. Bodei et al. [22] introduced the calculus LySa, a polyadic version of π-calculus (i.e. a version that allows tuples to be transmitted) that employs cryptographic primitives in the same way as the Spi Calculus. The language

uses pattern matching on input to distinguish between messages of different formats. In connection with the calculus, a control flow analysis framework, with a Dolev-Yao style attacker, was developed and shown capable of successfully analysing most of the well-known cryptographic protocols [24]. Buchholtz, Nielson, and Nielson [30] later suggested a more extensive calculus, LySa^{NS}, which incorporated a complex pattern matching to allow more succinct and comprehensible protocol specifications. The article also suggested a control flow analysis for the language, but this analysis resulted in constraints which lacked solver support and was never implemented.

1.3.3 Formal Protocol Analysis

We now turn to formal methods for verification of protocols. Roughly speaking there are two main trends in this area; *proof-based* approaches and *language-based* approaches. Proof-based approaches use a mathematical formalism to describe each step of the protocol, whereafter proof techniques are applied to establish various properties. We shall describe Belief logics, Strand Spaces, and theorem proving as examples of proof-based verification methods. Language-based approaches use process calculi to model the protocols and base an investigation of the protocol on the formal semantics of the language. We mention model checking, control flow analysis, and type systems as representatives of language-based verification methods.

Belief Logics. In their seminal work [31], Burrows, Abadi, and Needham developed BAN Logic, which is probably the first attempt at formal reasoning on cryptographic protocols. BAN Logic uses relations to represent the beliefs of principals and uses inference systems to derive the results of executing the steps of the protocol. Each step of the protocol is therefore a logical formula and the initiation of the protocol is described as a set of postulates (facts).

The central statement is that principal A believes statement X, which is written $A|\equiv X$. This means that A may behave as if X was true. Other constructs includes: $A \overset{K}{\longleftrightarrow} B$ expressing that A and B share the key K and may use that for communication; $A \triangleleft X$ means that A has seen X at some point; $\{X\}_K$ for the encryption of X with the key K. A typical rule could then be that if A believes that he shares a key K with B and has seen a message encrypted with K then A can see the contents of the encryption:

$$\frac{A|\equiv (A \overset{K}{\longleftrightarrow} B) \qquad A \triangleleft \{X\}_K}{A \triangleleft K}$$

The original semantics turned out to be unsatisfactory and later several attempts

were made to mend the technique [8, 21]. It is, for example, not clear whether A can believe something which is, in fact, false. BAN Logic has been seminal to security analysis, but it is also infamous for the unfortunate fact that it claimed several protocols to be secure, which later turned out to be flawed (e.g. Otway-Rees and the Needham-Schroeder Public-Key). It should also be noted that belief logics are only capable of establishing authentiation and that confidentiality must be established on top of the formula [56, 75], which tends to sacrifice the intuitive nature of the logic itself.

Strand Spaces. Another important approach [47] is due to Fábrega, Herzog, and Guttman. The technique relies on the notion of a strand *strand*; a sequential protocol history (message sending or receiving) from the viewpoint of a single peer. A *strand space* is then an unspecified set of strands, some for the principals of the system, some for the attacker.

The verification of processes rely on *bundles*; a set of strands sufficiently expressive to formalise a protocol session. Bundles relating to a particular protocol participant may then be seen as all possible combinations of events from that participants's view of the protocol run. Verification is then performed by showing that the bundles do not contain any strands where the attacker is allowed to do evil. Originally proofs of this type was carried out by pen and paper using inductive reasoning techniques, and was successfully applied to three classical protocols: Needham-Schroeder [47], Otway-Rees, and Yahalom [46]. Temporal logic has also been used to assist the verification methodology [33] and Song, Berezin, and Perrig [120] have combined model checking and theorem proving techniques in a tool called Athena, based on the Strand Spaces model.

Theorem Proving. Paulson [104] used higher-order logic and the interactive theorem prover Isabelle/HOL for protocol verification. Here protocols are described by inference rules producing sets of event sequences. Authentication and confidentiality properties are defined and shown as predicates over these event sequences. The technique allows verification of infinite state systems, but relies heavily on user interaction for generating proofs.

Model Checking. Model checking is a general verification technique performing exhaustive exploration of the possible states of a system. Model checking faces a problem of intrinsic character, the so-called *state explosion* problem: the state space grows exponentially with the size of the system specification. Over the years techniques, such as symbolic model checking [76], has been developed to overcome this limitation and model checking has significantly contributed to

the advance of the entire protocol verification field.

The use of model checking for verification purposes dates back more than two decades, where Lowe used CSP (see also §1.3.1 and §1.3.2) and the model checker FDR [114] to discover a number of attacks on classical cryptographic protocols [70, 71, 72]. Other model checkers such as ASTRAL [41], Brutus [36, 37], and Murphi [82], just to mention a few, have also been used for protocol verification purposes yielding perhaps less seminal results. Recent research by Basin, Mödersheim, and Viganò [13] should be mentioned, as it makes significant contributions to the field. Here infinite elements are represented using *lazy datatypes*: constructors that build datatypes without evaluating their arguments. This leads to a finite representation of the model which can be extracted on-demand. Another recent trend in model checking is the use of SAT-based techniques which have boosted the efficiency of protocol verification [11, 10].

Control Flow Analysis. Program analysis is concerned with statically estimating how programs may behave under execution [93]. Bodei et al. [25, 26, 22, 24] show how these techniques, namely *control flow analysis*, can be adapted to the verification of cryptographic protocols. Initially, the control flow analysis is applied to the π-calculus [25] and focusses on the use of channels. The authors then show [26] how to safely over-approximate variable bindings in Spi Calculus processes. The work is further matured in [22, 24], where a new variant of the π-calculus, LySa, is introduced. The authors here use control flow analysis to verify confidentiality and, with manual annotation techniques, authentication. The method is supported by an automated constraint solver, Succinct Solver [97], and has been applied successfully to a variety of protocols to find well-known as well as new flaws [23, 86].

Type Systems. Another program analysis approach to protocol security is based on type systems. A type systems is a set of inference rules that assign types to program expressions or terms. A type is identified with a certain meaning or behaviour and, thus, the type system assign meanings to programs. In his seminal paper [3], Abadi presents a type system for controlling information flow in a polyadic version of the Spi Calculus (see §1.3.2). This helps him to prove confidentiality and gave rise to a multitude of type systems for establishing various security properties of process calculi [58, 57, 65]. Abadi and Blanchet [4] have also extended it to a language in the applied pi calculus family (see §1.3.2) and shown its relation to the Prolog-based protocol checker ProVerif [19], which translates a process into a formula in first-order logic that can be automatically solved.

1.4 Contributions

Despite the fact that formal protocol modelling and verification has been an active research topic for three decades, it is still in the embryonic stage in many respects. Typically, verification methods assume that a protocol is executed in isolation, without other protocols sharing the network. However, studies [9, 68] show that attacks exists on protocols, otherwise considered secure, when executed multi-protocol scenarios. This is bad news for the classical protocol validation community, because it basically states that the verification of a security protocol is relatively uninteresting, unless the cryptographic preconditions of the protocol (i.e. the shared keys, key pairs, etc.) are used exclusively for that protocol. This criterion is rarely met as modern day systems have complex structures and seldom rely on the use of a single cryptographic protocol.

Instead the verification should be moved to the system design phase, taking the entire system design into account; ideally as a sub-component of the design tool where verification is done by the press of a button. To accommodate this, we shall, in this dissertation, be concerned with the creation of a language that can serve as a basis for such a component. Specifically, we build a formal language, CryptoKlaim, that allows modelling and automatic verification of cryptographic protocols and systems.

The works of Abadi and Gordon [7] and Buchholtz et al. [22, 24], which we have described above, are based on the same general idea and deserve thus a critical comparison with our approach. Both of the calculi created in these lines of work, that is the Spi calculus and LySa, are variations of the π-calculus with simple cryptographic constructs. CryptoKlaim differs from these languages in two respects: (1) it employs complex patterns matching on input using a term language in patterns; (2) it is based on the coordination language KLAIM and allows distribution of processes. The complex pattern matching allows succinct protocol specifications closely simulating the Alice-Bob notation, making CryptoKlaim a convenient and more intuitive alternative to the contemporary calculi. The support for distribution of processes allows the designer to take the structure of the network into account in the verification phase. It should be mentioned that the language LySa^{NS}, suggested by Buchholtz, Nielson, and Nielson in [30], has similar features. However, this language succumbed to its own expressiveness in that the control flow analysis for it was outside any known decidable domain, and never implemented. We ensure that CryptoKlaim will not suffer the same fate by designing the language in a analysis-directed manner. Specifically, we shall begin by showing how cryptographic protocols may be modelled and analysed in first-order predicate logic, the so-called Horn clauses, whereafter we shall engineer the process calculus around the same methodology. This way we are certain that CryptoKlaim remains verifiable.

As part of reaching the above stated goal, the dissertation makes a number of research contributions. These are summarised as follows:

(1) *An iterative framework for Horn clauses.* In Chapter 2 we present a generic iterative algorithm for finding a suitable model of Horn clauses. The algorithm promises termination and soundness, and gives sufficient criteria for completeness. The introduction of a terminating, sound manner of deciding a model (i.e. a solution to the constraints) of a Horn formula, makes Horn clauses much more attractive for static analysis purposes.

(2) *Implementation of the iterative $\mathcal{H}_1$-solver.* A specific instance of the iterative framework, using the decidable class of Horn clauses $\mathcal{H}_1$, was implemented. In Appendix B we give a succinct pseudo-code implementation of the $\mathcal{H}_1$-normalisation part, i.e. an $\mathcal{H}_1$-solver. This is a non-trivial programming task and we consider this succinct solution a programming pearl.

(3) *Definition of cryptographic components.* As a stand-alone module, we introduce the cryptographic components, encodings and patterns, in Chapter 3. The modelling of cryptography is orthogonal to the modelling of communication, and, thus, it seems natural to design the elements independently. The cryptographic components employ complex pattern matching features and allows modelling of both symmetric and asymmetric cryptography.

(4) *Definition of CryptoKlaim.* The core language of this dissertation is built as a conjunction of the coordination language, KLAIM, and the cryptographic components. The development is presented in Chapter 4 and requires subtle changes to the semantics of KLAIM. Several examples of use are also given, both in the form of running examples as well as in Appendix C.

(5) *Definition of well-formedness.* CryptoKlaim is equipped with a well-formedness condition that statically guarantees that programs never violate the assumption of perfect cryptography and that they remain closed under evaluation. Well-formedness is developed in two stages, first for the cryptographic components in Chapter 3 and afterwards extended to the language setting in Chapter 4.

(6) *A control flow analysis of CryptoKlaim.* a proof-of-concept control flow analysis is presented in Chapter 5 which safely approximates all messages that may be sent on the network and all values variables may be bound to during execution of a program. The correctness of the analysis is shown as well as how it may be implemented in Horn clauses.

(7) *Implementation of the control flow analysis.* The control flow analysis was implemented and applied to a number of well-known cryptographic protocols in Chapter 6, yielding the expected results.

(8) *A formal verification of an e-banking system.* A non-trivial case study was undertaken on an e-banking system provided by a commercial partner. The case study is a multi-protocol system which involves distribution, and it was formally verified using the control flow analysis.

This basically also establishes the outline of the remainder of the dissertation. Apart from the above-mentioned results, we shall, in Chapter 7, discuss future work and make concluding remarks. Although the reader is expected to have a basic understanding of set and order theory, we include, in Appendix A, the most important definitions and standard notations used in the theoretical parts of the dissertation.

Chapter 2

Analysis Base

In this chapter, we present an analysis base that will serve as the foundation of the language design in the following chapters. To ensure that we choose a proper such base, capable of expressing all the subtleties of the cryptographic systems, we turn to one of the oldest techniques in the book: Horn clauses.

Horn clauses have proven to be useful in many areas of computer science. They are very expressive (they are, in fact, Turing complete [123]) and yet they maintain a high clarity due to the simple format. This makes them attractive for many theoretical developments as well as for practical purposes, exemplified by the Prolog language.

Using Horn clauses for verifying cryptographic protocols is by no means a new idea. Since it was suggest in [69], it has shown a legitimate approach over the years [19, 118, 119]. Unfortunately, Horn clauses are not the panacea. The practicality of Horn clauses is limited by the very same reason that they are interesting: unrestricted Horn clauses are Turing complete. Thus, Horn clause-based problems are often undecidable, and, similarly, Horn clause-based algorithms usually have termination problems.

These are unsettling properties if we are to base an entire language design on Horn clauses. A verification module that sometimes cannot terminate will have a hard time finding any practically oriented users. Instead we shall approach the

$\varphi ::= \{c_1, \ldots, c_k\}$	$g ::= \mathsf{p}(t_1, \ldots, t_m)$
$c ::= g_0 \Leftarrow g_1 \wedge \cdots \wedge g_l$	$t ::= X \quad \mid \quad f(t_1, \ldots, t_n)$

Table 2.1: Horn syntax

problem in the typical static analysis manner; we will develop a framework that always presents us with an answer, by accepting that this answer may involve some approximation.

The following sections develop this framework. Specifically, we shall build an algorithm for finding models of Horn clauses that is guaranteed to terminate and be sound (i.e. the output of the algorithm is, in fact, a model of the clause) while giving sufficiency criteria for completeness (the algorithm finds the least model). Additionally, we shall show how cryptographic protocols may be modelled in Horn clauses, and how our framework may be used for verifying these models.

2.1 Horn Clauses

A Horn *formula* $\varphi \in \mathbf{HC}$ is a finite set of implications, usually referred to as *clauses*. Every clause, c, is of the form $g_0 \Leftarrow g_1 \wedge \cdots \wedge g_l$ where the literals g_0 and $g_1 \wedge \cdots \wedge g_l$ are the conclusion and the precondition of c, respectively. A *literal* is of the form $\mathsf{p}(t_1, \ldots, t_m)$ where p is an m-ary predicate symbol and t_i is a term built up from variables (indicated by a capitalised first letter) through constructor applications. In the following, we shall sometimes refer to nullary constructors as *constants*, and clauses without preconditions as *facts*, written without the implication arrow. Table 2.1 presents the resulting syntax of Horn clauses.[1]

We let expressions, e, range over formulae, clauses, literals and terms, and define $\mathsf{Var}(e)$, $\mathsf{Con}(e)$, and $\mathsf{Pred}(e)$, to be the set of variable, constructor and predicate symbols, respectively, that occur in e.[2] A term, t, is called *ground* if it does not contain any variables variables, i.e. $\mathsf{Var}(t) = \emptyset$, and for a given formula, φ, the entire set of all ground terms built from $\mathsf{Con}(\varphi)$ is called its *Herbrand universe*, denoted H_φ.

[1] The visually inclined reader notices the use of a simple notational scheme where predicate names in sans font, constructor names are written in red, and variable names in blue.

[2] Variables occurring in a clause are implicitly universally quantified in that clause; thus, without loss of generality, one may assume that variables in different clauses are α-renamed apart.

(HSat1)	$\rho \models \varphi$	*iff*	$\forall c \in \varphi : \rho \models c$
(HSat2)	$\rho \models g_0 \Leftarrow g_1 \wedge \cdots \wedge g_l$	*iff*	$\forall \theta : \bigwedge_{i=1}^{l} (\rho, \theta \models g_i) \Rightarrow (\rho, \theta \models g_0)$
(HSat3)	$\rho, \theta \models \mathsf{p}(t_1, \ldots, t_m)$	*iff*	$\langle t_1\theta, \ldots, t_m\theta \rangle \in \rho(\mathsf{p})$

Table 2.2: Satisfaction relation

Horn formulae are interpreted relative to an interpretation, $\rho \in \mathbf{R}$, that maps predicate symbols to corresponding term relations; i.e. an m-ary predicate symbol is mapped to an m-ary term relation, which is a subset of H_φ^m. Letting θ denote a substitution, mapping variables to ground terms, we then define a satisfaction relation $\models$ as presented in Table 2.2.[3]

An interpretation, ρ, which satisfies a Horn formula φ is called a model of φ. We then state the following well-known result:

Proposition 2.1 (Least model [125]) *The set of models of a Horn formula $\varphi \in \mathbf{HC}$ constitutes a Moore family (i.e. it is closed under arbitrary intersection), and thus it has a unique least model: $\rho_\varphi = \bigcap\{\rho \mid \rho \models \varphi\}$.*

Given a set of Horn clauses, we usually want to answer questions such as *membership* (does $\langle t_1, \ldots, t_m \rangle$ belong to the least model of p) or *satisfiability* (is p mapped to a non-empty set in the least model). Unfortunately, unrestricted Horn clauses are Turing complete [123], and these questions may be undecidable. Therefore, we are particularly interested in subsets of Horn clauses where they are decidable, and we shall refer to these fragments as *decidable classes*.

All our questions concern the least model of the formula, but as this model may include infinite sets, we need a finite representation. This finite representation can, in fact, also be Horn clauses; namely a carefully selected subset of Horn clauses on a form which allows membership in the least model to be determined in a straightforward manner (i.e. linear time). Such a form is called a *normal form*, and a transformation from a decidable class into a normal form is referred to as a *normalisation*, when it preserves the set of models from the original formula and can be performed through a finite number of operations. We present an example of a normal form in Table 2.3; notice that we require the conclusions to be *linear*; i.e. no variable occurs twice. This normal form essentially describes context-free grammars in a succinct manner and provides a linear time lookup for both membership and satisfiability.

[3]We adopt the usual postfix notation for substitutions, i.e. $\theta\sigma$ stands for $\sigma \circ \theta$; for an expression e then $e\theta$ represents the expression resulting from replacing all variables $X \in dom(\theta)$ in e with their corresponding ground terms. When convenient, we shall use the notation

(N1)	$\mathsf{p}(X_1, \ldots, X_m)$	$\Leftarrow$	$\mathsf{q}_1(X_1) \wedge \cdots \wedge \mathsf{q}_m(X_m)$	,	$\forall_{i \neq j}\ X_i \neq X_j \ \wedge m \geq 2$
(N2)	$\mathsf{p}(f(X_1, \ldots, X_n))$	$\Leftarrow$	$\mathsf{q}_1(X_1) \wedge \cdots \wedge \mathsf{q}_n(X_n)$	,	$\forall_{i \neq j}\ X_i \neq X_j$

Table 2.3: A normal form for Horn clauses

To ease the understanding of the development in the following sections, the reader may think of this particular normal form whenever we refer to an unspecified normal form for Horn clauses, bearing in mind, of course, that the results still apply to any other normal form as well. We shall denote the set of formulae belonging to a given normal form $\widehat{\mathbf{HC}}$ and write $\widehat{\varphi}$ for a formula belonging to this set.

2.2 Modelling Protocols

Before we delve more into decidable classes and the search for the least model of Horn clauses, we shall digress shortly, to argue that Horn clauses are a proper choice of basis for modelling and analysing cryptographic protocols.

2.2.1 Horn Clause Interpretation

Modelling protocol narrations in Horn clauses is relatively straightforward, as Horn clauses provide the means for an almost direct translation from the classic notation. One such translation scheme was presented intuitively in [19] and refined in [119] and we shall draw inspiration from these approaches.

Initially we notice that the protocol narration does not reveal all actions that must take place within each step; e.g. recall the third step of Yahalom from Example 1.1:

$$3. \quad S \rightarrow A : \{B, Kab, Na, Nb\}_{Ka}, \{A, Kab\}_{Kb}$$

This specifies that the server S sends a message to principal A as well as the syntax of this particular message. However, the notation assumes the existence of some sort of medium, upon which the message is sent, and implies that A must perform a series of actions after receiving the message. First A must recognise the principal S, and look up the key Ka that it shares with S. Afterwards, it

$[X \mapsto t]$ for the substitution mapping only the variable X to the ground term t.

1. $\mathsf{net}(\langle a, na\rangle)$
2. $\mathsf{net}(\langle b, \{\langle X_a, X_{na}, nb\rangle\}_{kb}\rangle) \Leftarrow \mathsf{net}(\langle X_a, X_{na}\rangle)$
3. $\mathsf{net}(\langle\{\langle b, kab, X_{na}, X_{nb}\rangle\}_{ka}, \{\langle a, kab\rangle\}_{kb}\rangle) \Leftarrow \mathsf{net}(\langle b, \{\langle a, X_{na}, X_{nb}\rangle\}_{kb}\rangle)$
4. $\mathsf{net}(\langle X_{enc}, \{X_{nb}\}_{X_{kab}}\rangle) \wedge \mathsf{a\_key}(X_{kab}) \Leftarrow \mathsf{net}(\langle\{\langle b, X_{kab}, na, X_{nb}\rangle\}_{ka}, X_{enc}\rangle)$
5. $\mathsf{b\_key}(X_{kab}) \Leftarrow \mathsf{net}(\langle\{\langle a, X_{kab}\rangle\}_{kb}, \{nb\}_{X_{kab}}\rangle)$

Figure 2.1: Yahalom in Horn clauses

must decrypt the first encrypted sequence of the message, using this key, and verify that the first part of that sequence is, in fact, the principal B that it wishes to establish a key with. Secondly, A must verify that the third part of the sequence is the nonce Na that it generated earlier during the protocol execution, and only then should it accept the key Kab and the nonce Nb. Finally, as A does not know Kb, A can only store the entire second encryption for use in the following step, without performing any checks on it.

This kind of interpretation is usually done manually by the protocol analyst and often results in some sort of extended protocol narration [77, 24]. Although some work has been done on automating this process [32, 34, 27], we shall settle for the manual method in this dissertation, because it is usually a trivial task. Note, however, that the Alice-Bob narration may be ambiguous, relying on an verbal description of intent, in which case an automatic interpretation is impossible.

For the purpose of this chapter, we will assume that all messages are sent on a single global network, here represented by the predicate net, meaning that all principals have access to this network. This is supposed to be a realistic model of the internet, without adding an extra layer of security to the protocol. Additionally, we shall model encryptions and concatenation through constructors, but to ease readability, we will employ the familiar notation $\{t_1\}_{t_0}$ for the binary encryption constructor, and $\langle t_1, \ldots, t_n\rangle$ for the n-ary concatenation constructor representing tuples. Finally, in order to stress the intent of the protocol, we add the predicates a_key and b_key for recording the ground terms that principals A and B, respectively, binds and believes to be the distributed key; that is, hopefully, Kab.

The resulting translation of Yahalom into Horn clauses is given in Figure 2.1, where we have used the notation $g_0 \wedge g_0' \Leftarrow g_1 \wedge \cdots \wedge g_l$ as a shorthand for the clauses $g_0 \Leftarrow g_1 \wedge \cdots \wedge g_l$ and $g_0' \Leftarrow g_1 \wedge \cdots \wedge g_l$. Notice that all constants are converted to lower-case to adhere to the Horn syntax. Each step of the

protocol is translated into one clause, where the right-hand side represents the requirements imposed by the sender, and the left hand side the message sent upon the network in that particular step. In addition to the four steps of the protocol, a fifth step is added to simulate the reception of the fourth message. Observe that the basic assumption of the protocol is preserved, namely that the initiator A and the server S both know the two other principals on advance, but that the responder B does not necessarily need to initially know (or trust) A. This is apparent in the second step, where B simply registers the sender's identity, unable to perform any validation of it.

2.2.2 Multiple Sessions and Instances

Modelling protocols as depicted in Figure 2.1 is sufficient for analysing the properties of a single execution of the protocol. But, in general, a protocol may be executed many times, leading to an unbounded number of *sessions*, and may also be used between many different principals, leading to an unbounded number of *instances*. Unfortunately, history shows that attacks may occur through multiple instances or parallel sessions, and thus a general framework should be able to cover this as well.

For Yahalom, there are several interesting scenarios one could think of, but for simplicity we shall settle for one, namely multiple A's and B's and one server S. Notice, though, that the methodology presented can easily be applied to model other scenarios as well, and we discuss this further in §2.6.

We assume that the same protocol (Yahalom) is used to establish a key between an arbitrary number of initiators $A_1, \ldots, A_k$ and an arbitrary number of responders $B_1, \ldots, B_l$ through a single server S. This means that the number of instances is unbounded and immediately poses a modelling problem. To deal with this, we shall proceed in the manner of [29], and adopt the abstraction that the infinite number of principals are divided into finitely many groups, usually referred to as *equivalence classes*; in particular, we shall use three equivalence classes for each role. Each equivalence class is then appointed a unique representative, a *canonical name*, indicated by the enclosing $\lfloor \cdot \rfloor$, and the modelling will then only be concerned with the canonical names of the equivalence classes, such that two different principals, from the same equivalence class, are indistinguishable.

Between each canonical initiator $\lfloor A_i \rfloor$ and canonical responder $\lfloor B_j \rfloor$, we may also have an unbounded number of sessions, each creating new nonces and a fresh key. Again, we may model this through equivalence classes, but this time we shall only introduce one equivalence class for each generated element in the protocol for each initiator and responder pair. For now it shall suffice to remark

0. $\mathsf{a}(\lfloor a_1 \rfloor), \mathsf{a}(\lfloor a_2 \rfloor), \mathsf{a}(\lfloor a_3 \rfloor), \mathsf{b}(\lfloor b_1 \rfloor), \mathsf{b}(\lfloor b_2 \rfloor), \mathsf{b}(\lfloor b_3 \rfloor)$

1. $\mathsf{net}(\langle X_a, \lfloor na \rfloor(X_a, X_b)\rangle) \Leftarrow \mathsf{a}(X_a) \wedge \mathsf{b}(X_b)$

2. $\mathsf{net}(\langle X_b, \{\langle X_a, X_{na}, \lfloor nb \rfloor(X_a, X_b)\rangle\}_{k(X_b,s)}\rangle) \Leftarrow \mathsf{net}(\langle X_a, X_{na}\rangle) \wedge \mathsf{b}(X_b)$

3. $\mathsf{net}(\langle\{\langle X_b, \lfloor kab \rfloor(X_a, X_b), X_{na}, X_{nb}\rangle\}_{k(X_a,s)}, \{\langle X_a, \lfloor kab \rfloor(X_a, X_b)\rangle\}_{k(X_b,s)}\rangle)$
$\Leftarrow \mathsf{net}(\langle X_b, \{\langle X_a, X_{na}, X_{nb}\rangle\}_{k(X_b,s)}\rangle) \wedge \mathsf{a}(X_a) \wedge \mathsf{b}(X_b)$

4. $\mathsf{net}(\langle X_{enc}, \{X_{nb}\}_{X_{kab}}\rangle) \wedge \mathsf{a\_key}(X_a, X_b, X_{kab})$
$\Leftarrow \mathsf{net}(\langle\{\langle X_b, X_{kab}, \lfloor na \rfloor(X_a, X_b), X_{nb}\rangle\}_{k(X_a,X_b)}, X_{enc}\rangle) \wedge \mathsf{a}(X_a) \wedge \mathsf{b}(X_b)$

5. $\mathsf{b\_key}(X_a, X_b, X_{kab}) \Leftarrow \mathsf{net}(\langle\{\langle X_a, X_{kab}\rangle\}_{k(X_b,s)}\rangle, \{\lfloor nb \rfloor(X_a, X_b)\}_{X_{kab}}) \wedge \mathsf{b}(X_b)$

Figure 2.2: Multiple instances and sessions of Yahalom in Horn clauses

that both of these abstractions are *sound*; i.e. they retain any true behaviour of the system, but may introduce additional false behaviour. We shall, however, justify this claim in §2.6.

The resulting modelling of multiple instances and sessions of the Yahalom protocol is given in Figure 2.2. Here we introduce the canonical names $\lfloor a_1 \rfloor$, $\lfloor a_2 \rfloor$, and $\lfloor a_3 \rfloor$ for the initiators, and similarly $\lfloor b_1 \rfloor$, $\lfloor b_2 \rfloor$, and $\lfloor b_3 \rfloor$ for the responders, and model the scenario in which all initiators may attempt to establish a key with all responders. Note that $\lfloor na \rfloor$ and $\lfloor nb \rfloor$ are now represented by binary constructors instead of nullary, as this allows for a much more generic modelling; by using the name of the initiating principal as the first argument and the name of the responder as the second, we uniquely identify the nonce with the instance that it is generated in. The same modelling trick is used for the initially shared keys between the server and the principals, as well as for the generated key. Finally, the arities of the predicates $\mathsf{a\_key}$ and $\mathsf{b\_key}$ are also extended with information about the initiator and responder, to distinguish keys that are distributed between different principals, and thus, in different instances.[4]

Observe that also this modelling captures the basic assumption of the protocol. Each action of the server and the initiators are guarded by the existence of the interacting principals, whereas the actions of the responders are not guarded by the existence of the initiator.

[4]Note that the long-term keys shared with the server need not be canonical, as they are static throughout the execution, in this protocol.

2.2.3 Network Attacker and Analysis

After modelling how the legitimate principals of the system interact, we usually want to apply an analysis for ensuring that the protocol cannot be compromised by one or more malicious principals.

One method that has been employed successfully numerous times before is modelling the so-called Dolev-Yao attacker [44], also known as the hardest network attacker [94], and investigate whether various security properties are upheld in presence of this attacker. In the Dolev-Yao model, the capabilities of this attacker are defined to be:

(1) receive and intercept all messages sent on the global network and send new messages onto the global network;

(2) decrypt encrypted messages, if it knows the encryption key, and construct new encryptions from known terms;

(3) decompose and compose tuples; and

(4) generate new ground terms.

The attacker can, in principle, compose and decompose tuples of any arity. However, in [24] it is shown that restricting the attacker to work on a limited set of arities, does not limit its capabilities, as long as the set includes the finite set of arities occurring in the protocol and at least one additional arity. This means that a Dolev-Yao attacker, for a specific protocol, is directly translatable into a finite set of Horn clauses. Assuming that $\mathcal{K}$ denotes the finite set of occurring arities (for Yahalom this set is $\{2, 3, 4\}$), we can create an extended set $\mathcal{K}_+ = \mathcal{K} \cup \{1\}$, as unary tuples never occurs, and a generic Dolev-Yao attacker can then be formulated as in Figure 2.3.

The attacker is described through a predicate dy, storing its accumulated knowledge. As the formulae in Table 2.1 and Table 2.2 uses the predicate net for transmitting all messages, then sending, receiving and intercepting becomes just a matter of using this predicate as well. The first three rules are then straightforward, and the rule (DY4) expresses that the attacker may generate arbitrary ground terms, $m_1, m_2, \ldots$, but, in the modelling, all of these are coalesced into one equivalence class represented by the canonical constant $\lfloor m_\bullet \rfloor$. Note that a direct consequence of the rule (DY1) is that the attacker's knowledge and the net are identical.

A conjunction of a protocol description with the attacker in Figure 2.3 constitutes an instance of an analysis problem, and the least model of such a problem allows

$$
\begin{array}{ll}
\text{(DY1)} & \mathsf{dy}(X) \Leftarrow \mathsf{net}(X) \\
 & \mathsf{net}(X) \Leftarrow \mathsf{dy}(X) \\
\text{(DY2)} & \mathsf{dy}(X) \Leftarrow \mathsf{dy}(\{X\}_{X_{key}}) \wedge \mathsf{dy}(X_{key}) \\
 & \mathsf{dy}(\{X\}_{X_{key}}) \Leftarrow \mathsf{dy}(X) \wedge \mathsf{dy}(X_{key}) \\
\text{(DY3)} & \left.\begin{array}{l} \mathsf{dy}(X_1) \wedge \cdots \wedge \mathsf{dy}(X_k) \Leftarrow \mathsf{dy}(\langle X_1, \ldots, X_k \rangle) \\ \mathsf{dy}(\langle X_1, \ldots, X_k \rangle) \Leftarrow \mathsf{dy}(X_1) \wedge \cdots \wedge \mathsf{dy}(X_k) \end{array}\right\} \quad \text{if } k \in \mathcal{K}_+ \\
\text{(DY4)} & \mathsf{dy}(\lfloor m_\bullet \rfloor)
\end{array}
$$

Figure 2.3: Dolev-Yao attacker in Horn clauses

us to investigate certain properties of the protocol. In its most simple form, such as the formulae in Figure 2.1 or Figure 2.2, the protocol description allows direct verification of confidentiality of certain secrets, e.g. the distributed keys. This simply amounts to investigating whether they are present in the attacker's knowledge (the predicate dy) according to the least model.

Furthermore, the protocol fulfills its purpose if the only key that may be successfully distributed between A_i and B_j, is $K_{A_iB_j}$. Hence, we can also validate integrity through the predicates a_key and b_key; e.g. for Figure 2.2 by investigating whether they are only bound to tuples of the form $\langle \lfloor a_i \rfloor, \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor) \rangle$ in the least model. Several additional properties can also be verified with the same methodology using annotations on the protocols and embedding them into the the Horn clause modelling. The use of annotations is an orthogonal issue to the development of the static analysis itself, and lies beyond the scope of this dissertation. We shall, however, do a cursory review on how annotations can improve the verification process in §7.2.

2.3 Finding the (Least) Model

In §2.2 it is shown how gently cryptographic protocols can be translated into Horn clauses. Although the presentation may seem somewhat informal, in particular we did not prove the soundness of the analysis, it hopefully still appears convincingly through the transparency of the translation scheme and the limited amount of approximation steps involved. At least it seems to achieve its goal, namely showing that analysing cryptographic protocols in Horn clauses can be done in a simple and elegant manner.

Thus, it appears, that it only remains to show how the least model of these

clauses can be determined. Unfortunately, as already mentioned, problems formulated in unrestricted Horn clauses may be undecidable.

In this section we shall attempt to circumvent this shortcoming of Horn clauses, thereby making them more attractive for theoretical purposes; in particular for static analysis and our purpose. Specifically, we shall present a framework for finding, and iteratively improving the precision of, a model for any set of unrestricted Horn clauses. This framework guarantees *soundness* (i.e. it always returns a correct model) and termination, and it may in some cases also provide *completeness* (the model is the least model). Completeness depends on the chosen instance of the framework, as the framework is generic in the sense that it can be instantiated with any known decidable fragment of Horn clauses.

2.3.1 Decidability and beyond...

In general, it may be undecidable whether a given protocol violates a specific security property [45]. It follows, that any attempt at constructing a complete framework for modelling and precisely analysing protocols, will in some cases also lead to undecidability. Our attempt in §2.2 is not an exception to this rule. In fact, an inspection of the most general decidable classes in related literature [49, 38, 118, 119] reveals that both the formulae in Figure 2.1 and Figure 2.2 fall, at least to the best of our knowledge, outside the known decidable subclasses of Horn clauses. In particular we have that:

- The clauses do *not* belong to $\mathcal{H}_1$ (the class of strongly connected clauses originally introduced by Nielson, Nielson, and Seidl [96]). For Figure 2.2 this can be directly concluded from the first requirement to $\mathcal{H}_1$ clauses, namely that they must have linear conclusions (no variables occur twice), which is obviously not the case for step 1, 2, and 3. The formula in Figure 2.1 is, in fact, also outside $\mathcal{H}_1$, due to the second, more complicated, requirement which we shall elaborate on in §2.4.1.

- The clauses do *not* belong to the *general flat* class (a generalisation of the flat clauses introduced by Seidl and Verma [118]), as flat clauses are only allowed to contain literals of the form $\mathsf{p}(X)$ and $\mathsf{p}(f(X_1, \ldots, X_n))$. The clauses generated in the previous section are obviously outside this class.

- The clauses do *not* belong to the *k-variable class*.[5] This class includes

[5] This class was introduced by Seidl and Verma [119] as k-variable and flat clauses. The flat clauses are, however, subsumed by the k-variable class and the distinction was only made to improve complexity.

weakly covering[6] clauses with at most k variables, for some constant k. A term or literal is defined to be weakly covering if every non-ground functional sub-term contains all variables of the entire term or literal. A clause is then called weakly covering if all literals in it are weakly covering and every non-ground non-unary literal contains all variables of the entire clause. It is quite obvious that even this general class cannot contain the clauses produced in the previous section.

Furthermore, the classes *One-Variable Clauses* and *One-Variable and Flat Clauses* are also specified in [119], but these are both subsumed by the k-variable class. This is also the case for the extension of the Skolem class $\mathcal{S}^+$ from [49], as well as the Skolem class extension presented in [38].

So it seems that a relatively simple, classical key distribution protocol has forced us beyond known decidable fragments, illustrating the complexity of protocol analysis. So we may not be able to find the *least model* of the problem, but we can always find *some model* (although the Herbrand universe is a rather useless model). Sometimes this model is small enough to guarantee some properties, whereas in other cases it gives some insight into how the search space can be restricted, such that a new, better model can be determined. This motivates an iterative approach.

2.3.2 The Iterative Framework

The general structure of our iterative framework is shown in Figure 2.4. It is parameterised on a decidable fragment $\mathcal{H}$ and assumes four main ingredients:

- an *$\mathcal{H}$-normalisation* (sometimes called an $\mathcal{H}$-solver), η, a function that finds the least model of any formula in $\mathcal{H}$;
- an *$\mathcal{H}$-relaxation*, α, a function producing approximations within $\mathcal{H}$ of any unrestricted Horn formula;
- a *specialisation*, β, a function that, given a formula and the corresponding model, produces a specialised formula that preserves all models smaller than or equal to the input; and
- a *simplification ordering*, $\prec$, a well-founded order on formulae.

[6] Note that this was referred to as *covering* in [119], but that we have chosen to follow the weakly covering definition of [49], to avoid confusion.

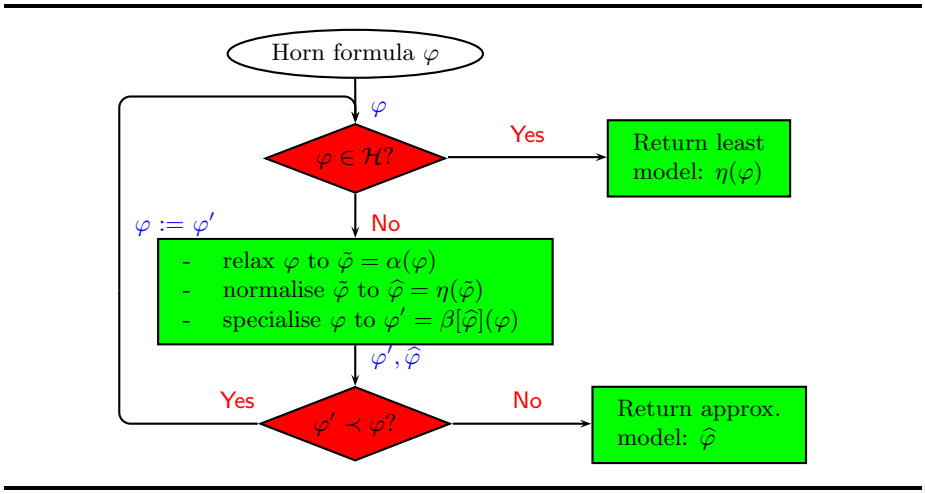

Figure 2.4: Iterative framework

Later we shall give a more formal definition of each of these operators.

The algorithm then proceeds as follows. Given a formula, φ, the first step is to check whether φ is in $\mathcal{H}$. If so, then we can immediately construct its least model using the normalisation procedure for $\mathcal{H}$ (denoted $\eta(\varphi)$ in the figure) and we are done.

If φ is not in $\mathcal{H}$ then we apply an *$\mathcal{H}$-relaxation*, α, and the resulting formula, $\tilde{\varphi} = \alpha(\varphi)$, will be in $\mathcal{H}$. The relaxation also guarantees that the least model of this formula (denoted by the normal form $\widehat{\varphi} = \eta(\tilde{\varphi})$ on the figure) is also a model of φ, but it is possibly too large to be useful by itself. It may, however, contain useful information that can be used to *specialise* the original formula φ; this operation is denoted $\varphi' = \beta[\widehat{\varphi}](\varphi)$ on Figure 2.4. In order to compare the new and the original formula, we introduce a well-founded, *simplification ordering*, $\preceq$, over formulae; if the new formula is an improvement of the old, according to this ordering, then the above steps can be repeated. Otherwise the iteration stops and $\widehat{\varphi}$ will be the resulting (approximative) model of φ.

We shall now present the key ingredients that we, in addition to the normalisation, informally introduced in §2.1; including the relaxation mechanism, the specialisation function and the simplification ordering.

2.3.3 The Refinement Scheme

Recall that **HC** and $\widehat{\mathbf{HC}}$ denote the (infinite) sets of Horn formulae and normal form Horn formulae, respectively. We then define the premise of our intuition as follows: there exists a function that takes a general Horn formula and returns an over-approximative formula (ie. a formula, for which all models are also models of the original formula, and thus its least model is a super set or the same as the least model of the original) which belongs to some decidable class $\mathcal{H}$. We shall call such a function an *$\mathcal{H}$-relaxation* and formally define it as follows:

Definition 2.2 ($\mathcal{H}$-Relaxation) *Let $\mathcal{H} \subseteq \mathbf{HC}$, then a function $\alpha : \mathbf{HC} \to \mathcal{H}$ is an $\mathcal{H}$-relaxation if:*

$$\forall \varphi : \forall \rho : \quad \rho \models \alpha(\varphi) \Rightarrow \rho \models \varphi$$

The definition allows the relaxed formula to have a larger least model than the original and even define additional predicate symbols. However, as we are only concerned with the predicate symbols from the original formula, we shall introduce the notation, $\sqsubseteq_\varphi$, for relations between models with respect to a formula φ:

$$\rho \sqsubseteq_\varphi \rho' \quad \Leftrightarrow \quad \forall \mathsf{p} \in \mathsf{Pred}(\varphi) : \ \rho(\mathsf{p}) \subseteq \rho'(\mathsf{p})$$

We shall furthermore adopt the familiar notation $\equiv_\varphi$ to denote the symmetric subset of $\sqsubseteq_\varphi$. Notice that from the definition of the relaxation, it then follows that $\rho_\varphi \sqsubseteq_\varphi \rho_{\alpha(\varphi)}$.

As $\mathcal{H}$ is decidable, then there also exists a function that takes a formula $\varphi \in \mathcal{H}$ and returns a formula in normal form representing its unique least model ρ_φ. Such a function is usually called an *$\mathcal{H}$-normalisation*:

Definition 2.3 ($\mathcal{H}$-Normalisation) *Let $\mathcal{H} \subseteq \mathbf{HC}$, then a function $\eta : \mathcal{H} \to \widehat{\mathbf{HC}}$ is an $\mathcal{H}$-normalisation if:*

$$\forall \varphi \in \mathcal{H} : \quad \rho_\varphi \equiv_\varphi \rho_{\eta(\varphi)}$$

Thus, a formula $\widehat{\varphi}$ is a normalisation of a formula φ when it is in normal form and the least models of the formulae agrees on the map of all the predicates in φ.

The combination of the relaxation and the normalisation establishes that we can always find some model of a formula, but we are usually only interested

in the least model or a model close to the least model. Naturally, if α returns a formula with the same least model as the original, then the result is precise and we are done. But this is usually not the case, and often the approximation will be too coarse to be useful for the intended purpose. In this case, we may inspect this coarse model and look for information that allow us to produce a safe transformation of the formula that may produce better results. This is called a specialisation.

Definition 2.4 (Specialisation) *A specialisation is a higher-order function* $\beta : \widehat{\mathbf{HC}} \to \mathbf{HC} \to \mathbf{HC}$ *that satisfies:*

$$\forall \varphi : \forall \widehat{\varphi} : \forall \rho \sqsubseteq_{\varphi} \rho_{\widehat{\varphi}} : \quad \rho_{\widehat{\varphi}} \models \varphi \Rightarrow (\rho \models \beta[\widehat{\varphi}](\varphi) \Leftrightarrow \rho \models \varphi)$$

A specialisation takes a formula in normal form $\widehat{\varphi}$ and specialises the formula φ, such that the specialised formula $\beta[\widehat{\varphi}](\varphi)$ maintains the set of models, smaller than or equal to $\rho_{\widehat{\varphi}}$; in particular it preserves the least model. Notice that, contrary to the relaxation and normalisation, the specialisation may not introduce any auxiliary predicates.

The fourth, and last, ingredient we shall require for the iterative framework, is an order that ensures termination.

Definition 2.5 (Simplification ordering) *A relation* $\preceq : \mathbf{HC} \times \mathbf{HC}$ *is a simplification ordering if it is a well-founded, partial order.*

When convenient, we shall use familiar notation $\succeq$ for the dual of $\preceq$ and likewise $\succ$ and $\prec$ for their respective strict counterparts.

The combination of these operators is then supposed to form the basis of the iterative framework for finding solutions for unrestricted Horn problems. But, for this to be achieved, we shall furthermore require them to form an $\mathcal{H}$-refinement scheme:

Definition 2.6 ($\mathcal{H}$-Refinement Scheme) *A quadruple* $(\alpha, \eta, \beta, \preceq)$*; consisting of an* $\mathcal{H}$*-relaxation, an* $\mathcal{H}$*-normalisation, a specialisation, and a simplification ordering, is an* $\mathcal{H}$*-refinement scheme whenever:*

$$\forall \varphi : \forall \widehat{\varphi} : \quad \rho_{\alpha(\beta[\widehat{\varphi}](\varphi))} \sqsubseteq_{\varphi} \rho_{\alpha(\varphi)}$$

Or in plain terms: we require the specialisation to weakly reduce (i.e. reduce or maintain) the over-approximation introduced by the relaxation.

2.3.4 Properties of the Framework

The refinement scheme forms the basis of the larger iterative framework, that was graphically presented in Figure 2.4 in §2.3.2. Essentially, for any decidable class $\mathcal{H}$ for which there exists an $\mathcal{H}$-refinement scheme, $(\alpha, \eta, \beta, \preceq)$, the algorithm will determine a best solution to any unrestricted Horn problem.

Recall that the algorithm can terminate in two ways: (1) Either the scheme, through specialisation, eventually produces a formula within $\mathcal{H}$, which is therefore decidable and the result is the normalised formula; or (2) the scheme eventually fails to improve the formula further, with respect to $\preceq$, and the result is then the normalised, relaxed formula.

As part of justifying the proposed framework, we shall then show that it satisfies three key properties: we shall prove that the algorithm always terminates, that the normalised output represents a model of the input formula, and that if the algorithm produces a formula within the decidable class then the corresponding normal form represents the least model of the original input formula.

First we shall give the termination result.

Theorem 2.7 (Termination) *If α, η, and β are terminating then the algorithm is terminating.*

Proof The result follows directly from the fact that $\preceq$ is well-founded. □

The next result states that the algorithm is sound and justifies the iterative approach.

Theorem 2.8 (Soundness) *The algorithm will always produce a model of the input formula φ. Each iteration (weakly) improves precision of this model.*

Proof Let φ_i be the formula used as input in the ith step of the protocol. It follows, that if the algorithm terminates in the ith iteration then the result is either the least model of $\eta(\varphi_i)$ or $\widehat{\varphi}_i = \eta(\alpha(\varphi_i))$. Thus, the proof amounts to show that these normalised clauses represent models of the input formula for all i. This is proven by showing that each iteration maintains $\forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} : (\rho \models \varphi_i \Leftrightarrow \rho \models \varphi)$. This follows by induction:

Case i=0 holds vacuously by Definition 2.4.

Case i+1. Assume the input φ_i and $\widehat{\varphi}_i = \eta(\alpha(\varphi_i))$:

$$
\begin{array}{lll}
& \widehat{\varphi}_i = \eta(\alpha(\varphi_i)) & \\
\Rightarrow & \rho_{\widehat{\varphi}_i} \models \varphi_i & \text{(by Def. 2.2 \& 2.3)} \\
\Rightarrow & \forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} : (\rho \models \beta[\widehat{\varphi}_i](\varphi_i) \Leftrightarrow \rho \models \varphi_i) & \text{(by Def. 2.4)} \\
\Leftrightarrow & \rho_{\alpha(\beta[\widehat{\varphi}_i](\varphi_i))} \sqsubseteq_\varphi \rho_{\alpha(\varphi_i)} \wedge & \text{(by Def. 2.6)} \\
& \forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} : (\rho \models \beta[\widehat{\varphi}_i](\varphi_i) \Leftrightarrow \rho \models \varphi_i) & \\
\Leftrightarrow & \rho_{\widehat{\varphi}_{i+1}} \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} \wedge \forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} : (\rho \models \varphi_{i+1} \Leftrightarrow \rho \models \varphi_i) & \text{(by Def. 2.3)} \\
\Leftrightarrow & \rho_{\widehat{\varphi}_{i+1}} \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} \wedge \forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_i} : (\rho \models \varphi_{i+1} \Leftrightarrow \rho \models \varphi) & \text{(by Ind. Hyp.)} \\
\Leftrightarrow & \forall \rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}_{i+1}} : (\rho \models \varphi_{i+1} \Leftrightarrow \rho \models \varphi) &
\end{array}
$$

Where the last step uses the transitivity of $\sqsubseteq_\varphi$. As both $\rho_{\varphi_{i+1}} \sqsubseteq_\varphi \rho_{\widehat{\varphi}_{i+1}}$ and $\rho_{\widehat{\varphi_{i+1}}} \sqsubseteq_\varphi \rho_{\widehat{\varphi}_{i+1}}$ this concludes the proof. □

And lastly we have the partial completeness result:

Theorem 2.9 (Partial Completeness) *If the algorithm terminates in state* (1)*, then it produces the least model of the input formula* φ*.*

Proof The result follows directly from the proof for Theorem 2.8. □

2.4 $\mathcal{H}_1$-Refinement Scheme

We now present an instance of, or rather a specific refinement scheme for, the iterative framework of §2.3.2. For this purpose, we have chosen the decidable subclass $\mathcal{H}_1$ [96].

$\mathcal{H}_1$ describes strongly recognisable relations; i.e. finite unions of Cartesian products of recognisable tree languages. In fact, every clause in $\mathcal{H}_1$ is normalisable (to the normal form presented in §2.1) and the equivalent normal form can be constructed in deterministic exponential time [96].

Previously, this class has proven very useful for static analysis purposes, both for specifying a Control-Flow Analysis of Spi [96] as well as for verifying real implementations of cryptographic protocols in the C language [60]. Yet, as we discussed in §2.3.1, a direct attempt at specifying protocol analysis, results in clauses even outside this class. This motivates the use of the iterative framework of §2.3.2.

We begin by briefly introducing the $\mathcal{H}_1$-class itself, whereafter we shall define the operators in the $\mathcal{H}_1$-refinement scheme, $(\alpha_1, \eta_1, \beta_1, \preceq_1)$, needed for using the iterative framework.

(H1.1)	g_0 is linear.
(H1.2)	If $X, Y \in \mathsf{Var}(g_0)$ are connected in $\{g_1, \ldots, g_k\}$, then they are siblings in g_0.

Table 2.4: Property $H1$ for clauses of the form $g_0 \Leftarrow g_1 \wedge \cdots \wedge g_k$.

2.4.1 The Class $\mathcal{H}_1$

Let the relation *connected* be the transitive closure of variables occurring in the same literal over a set of literals; i.e. two variables X and Y are connected within the set $\{g_1, \ldots, g_k\}$ if there exists a sequence of literals $g_{n_1} \ldots g_{n_l}$ with $l \geq 1$ and $n_i \in \{1, \ldots, k\}$ such that $X \in \mathsf{Var}(g_{n_1})$, $Y \in \mathsf{Var}(g_{n_l})$ and $\mathsf{Var}(g_{n_i}) \cap \mathsf{Var}(g_{n_{i+1}}) \neq \emptyset$ for all $i \in \{1, \ldots, l-1\}$. Hence, for example, X and Y are connected in $\{\mathsf{p}(X, f(Y))\}$ and $\{\mathsf{p}(X, Z), \mathsf{q}(Z, f(Y))\}$, but not in $\{\mathsf{p}(X, Z), \mathsf{q}(f(Y))\}$.

We then say that the clause $g_0 \Leftarrow g_1, \ldots, g_k$ has the property H1 if it satisfies the requirements given in Table 2.4. Here we call two variables *siblings* in a literal or term, if they occur as arguments of a common parent; e.g. X, Y are siblings in $\mathsf{p}(X, Y)$ and $\mathsf{p}(Z, f(X, Y))$ but not in $\mathsf{p}(X, f(Y))$.

A formula φ belongs to the class $\mathcal{H}_1$ if all clauses in φ have property H1.

Example 2.10 *Recall the following clause from the modelling of Yahalom in Figure 2.1 in §2.2:*

$$\mathsf{net}(\langle X_{enc}, \{X_{nb}\}_{X_{kab}}\rangle) \Leftarrow \mathsf{net}(\langle \{\langle b, X_{kab}, na, X_{nb}\rangle\}_{ka}, X_{enc}\rangle)$$

Remembering that $\langle t_1, \ldots, t_n\rangle$ is an n-ary constructor and $\{t_1\}_{t_0}$ a binary constructor, it follows that this clause is not *in $\mathcal{H}_1$; the variables X_{enc} and X_{nb} (and similarly X_{enc} and X_{kab}) are connected in the precondition but are not siblings in the conclusion, thus violating the rule* (H1.2) in Table 2.4.

2.4.2 Relaxation Operator

We now want to define a relaxation function, α, that always produces clauses within $\mathcal{H}_1$. In [59] it is shown how general Horn clauses can be approximated by $\mathcal{H}_1$ clauses. We shall follow this approach, but slightly improve it, as the relaxation operator we suggest, for clauses violating (H1.2), is more precise as compared to the approximation technique suggested in [59].

Let e be an expression. We then write $e[X \rightsquigarrow t]$ for the expression that is as e, except that the leftmost occurrence of X is replaced by t. We shall also employ the shorthand notation ω for a sequence of literals $g_1 \wedge \ldots \wedge g_l$ when convenient, and point-wise extend substitutions and $\models$ for this. We then define a function $\alpha_1(\varphi) = \bigcup\{\alpha_1(c) \mid c \in \varphi\}$ as follows:

$$\alpha_1(g \Leftarrow \omega) = \begin{cases} \alpha_1(g[X \rightsquigarrow Y] \Leftarrow \omega[X \mapsto Y] \wedge \omega) & \text{, if } Y \text{ is fresh} \quad (a) \\ \alpha_1(g \Leftarrow \omega[X \mapsto Y] \wedge \mathsf{p}(X, Z_1, \ldots, Z_k)) \\ \cup \{\mathsf{p}(X, Z_1, \ldots, Z_k) \Leftarrow \omega\} & \text{, if } Y \text{ and } \mathsf{p} \text{ are fresh} \quad (b) \\ \{g \Leftarrow \omega\} & \quad (c) \end{cases}$$

Here (a) is chosen if $g \Leftarrow \omega$ violates (H1.1), (b) if if $g \Leftarrow \omega$ satisfy (H1.1) but violates (H1.2), and otherwise (c) applies. In both (a) and (b) X is the leftmost variable that gives rise to the violation, and in (b) the siblings $Z_1, \ldots, Z_k$ of X in g are carried on to the auxiliary predicate, to retain the highest amount of precision.[7]

From the definition it is apparent that the set of clauses generated by α_1 will always be in $\mathcal{H}_1$, and that, if applied to a set of clauses already within $\mathcal{H}_1$, α_1 will be the identity function. Furthermore, as the set of variables in a clause is always finite, it should be fairly obvious that α_1 will always terminate.

Now, prior to showing that α_1 indeed constitutes a relaxation function, we shall give an auxiliary result that benefits the presentation of the proof.

Fact 2.11 *If $\rho \models c$ then $\forall X, Y : \rho \models c[X \mapsto Y]$.*

Proof The proof follows straightforwardly from the definitions:

$$\begin{array}{ll} \rho \models g \Leftarrow \omega & \\ \Leftrightarrow & \forall \theta : \rho, \theta \models \omega \Rightarrow \rho, \theta \models g \\ \Rightarrow & \forall \theta : \forall X, Y : \theta(X) = \theta(Y) \Rightarrow (\rho, \theta \models \omega \Rightarrow \rho, \theta \models g) \\ \Leftrightarrow & \forall \theta : \forall X, Y : \rho, \theta \models \omega[X \mapsto Y] \Rightarrow \rho, \theta \models g[X \mapsto Y] \\ \Leftrightarrow & \forall X, Y : \rho \models (g \Leftarrow \omega)[X \mapsto Y] \end{array}$$

This leads to the result we seek. □

Lemma 2.12 *The function α_1 is a $\mathcal{H}_1$-relaxation operator.*

Proof The proof amounts to showing that if $\rho \models \alpha_1(\varphi)$ then $\rho \models \varphi$; i.e. if $\rho \models \alpha_1(c)$ then $\rho \models c$. This is proven by induction in the derivation sequence establishing $\alpha_1(c)$:

[7] The approximation technique suggested in [59] does not include the siblings in the auxiliary predicate.

(a) remembering that $Y \notin \mathsf{Var}(g \Leftarrow \omega)$ the result follows from Fact 2.11 :

$$\begin{array}{lll} \rho \models g[X \rightsquigarrow Y] \Leftarrow \omega[X \mapsto Y] \wedge \omega & & \\ \Rightarrow & & \rho \models (g[X \rightsquigarrow Y] \Leftarrow \omega[X \mapsto Y] \wedge \omega)[Y \mapsto X] \\ \Leftrightarrow & & \rho \models g \Leftarrow \omega \wedge \omega \\ \Leftrightarrow & & \rho \models g \Leftarrow \omega \end{array}$$

(b) is shown analogously to (a):

$$\begin{array}{ll} \rho \models \{g \Leftarrow \omega[X \mapsto Y] \wedge \mathsf{p}(X, Z_1, \ldots, Z_k),\ \mathsf{p}(X, Z_1, \ldots, Z_k) \Leftarrow \omega\} \\ \Leftrightarrow & \rho \models (g \Leftarrow \omega[X \mapsto Y] \wedge \mathsf{p}(X, Z_1, \ldots, Z_k)) \wedge \rho \models (\mathsf{p}(X, Z_1, \ldots, Z_k) \Leftarrow \omega) \\ \Rightarrow & \rho \models g \Leftarrow \omega[X \mapsto Y] \wedge \omega \\ \Rightarrow & \rho \models (g \Leftarrow \omega[X \mapsto Y] \wedge \omega)[Y \mapsto X] \\ \Leftrightarrow & \rho \models g \Leftarrow \omega \wedge \omega \\ \Leftrightarrow & \rho \models g \Leftarrow \omega \end{array}$$

(c) holds vacuously. □

Example 2.13 (Example 2.10 continued) *Recall that the clause from Example 2.10 was not in $\mathcal{H}_1$, as the variables X_{enc} and X_{nb} violated rule* (H1.2). Thus, the relaxation α_1 applied to this clause will decouple these variables, resulting in the following two clauses, now both in $\mathcal{H}_1$.

$$\mathsf{net}(\langle X_{enc}, \{X_{nb}\}_{X_{kab}}\rangle) \Leftarrow \mathsf{net}(\langle\{\langle b, X_{kab}, na, X_{nb}\rangle\}_{ka}, X'\rangle) \wedge \mathsf{p}(X_{enc})$$
$$\mathsf{p}(X_{enc}) \Leftarrow \mathsf{net}(\langle\{\langle b, X_{kab}, na, X_{nb}\rangle\}_{ka}, X_{enc}\rangle)$$

2.4.3 $\mathcal{H}_1$-Normalisation

Normalising a set of $\mathcal{H}_1$-clauses amounts to bringing the clauses onto the normal form given in Table 2.3 in §2.1. This is a well-known procedure that was initially described in [96] in a direct manner and later using resolution techniques in [59]. Basically the procedure consists of iteratively extending the set of normal clauses by simplifying non-normal clauses using the current set of normal clauses, this procedure is continued until no further simplification can be performed; then the non-normal clauses are redundant and can be removed.

In Table 2.5 we presents the normalisation procedure as an inference system.[8] Assuming that preconditions may be ordered freely to match the rules, each rule should be read as follows:

[8] We follow the convention of omitting the $\wedge$'s between multiple preconditions or conclusions of the inference rules, to ease readability.

(Norm1) $(\mathsf{p}_H(f(X_1,\ldots,X_n)) \Leftarrow \mathsf{p}_H(X_1)\wedge\cdots\wedge\mathsf{p}_H(X_n)) \in \varphi \qquad \text{if } f \in \mathsf{Con}(\varphi)$

(Norm2) $$\frac{(g \Leftarrow \omega)\in\varphi \qquad X\in\mathsf{Var}(g) \qquad X\notin\mathsf{Var}(\omega)}{(g \Leftarrow \mathsf{p}_H(X)\wedge\omega)\in\varphi}$$

(Norm3) $$\frac{(\mathsf{p}(t_1,\ldots,t_m)\Leftarrow\omega)\in\varphi \qquad t_i = f'(t'_1,\ldots,t'_{n'})}{\begin{array}{c}(\mathsf{p}(t_1,\ldots,X_i,\ldots,t_m)\Leftarrow\mathsf{aux}(X_i)\wedge\omega)\in\varphi\\ (\mathsf{aux}(t_i)\Leftarrow\omega)\in\varphi\end{array}} \quad \begin{array}{l}\text{if } X_i\notin\mathsf{Var}(\varphi)\\ \mathsf{aux}\notin\mathsf{Pred}(\varphi)\end{array}$$

(Norm4) $$\frac{(\mathsf{p}(f(t_1,\ldots,t_n))\Leftarrow\omega)\in\varphi \qquad t_i = f'(t'_1,\ldots,t'_{n'})}{\begin{array}{c}(\mathsf{p}(f(t_1,\ldots,X_i,\ldots,t_n))\Leftarrow\mathsf{aux}(X_i)\wedge\omega)\in\varphi\\ (\mathsf{aux}(t_i)\Leftarrow\omega)\in\varphi\end{array}} \quad \begin{array}{l}\text{if } X_i\notin\mathsf{Var}(\varphi)\\ \mathsf{aux}\notin\mathsf{Pred}(\varphi)\end{array}$$

(Norm5) $$\frac{(g\Leftarrow\mathsf{p}(t_1,\ldots,t_m)\wedge\omega)\in\varphi \qquad (\mathsf{p}(X_1,\ldots,X_m)\Leftarrow\omega')\in\widehat{\varphi} \qquad m\geq 2}{(g\Leftarrow\omega'[X_1\mapsto t_1,\ldots,X_m\mapsto t_m]\wedge\omega)\in\varphi}$$

(Norm6) $$\frac{(g\Leftarrow\mathsf{p}(f(t_1,\ldots,t_n))\wedge\omega)\in\varphi \qquad (\mathsf{p}(f(X_1,\ldots,X_n))\Leftarrow\omega')\in\widehat{\varphi}}{(g\Leftarrow\omega'[X_1\mapsto t_1,\ldots,X_n\mapsto t_n]\wedge\omega)\in\varphi}$$

(Norm7) $$\frac{(\mathsf{p}(X)\Leftarrow\mathsf{q}(X))\in\varphi \qquad (\mathsf{q}(t)\Leftarrow\omega)\in\widehat{\varphi}}{(\mathsf{p}(t)\Leftarrow\omega)\in\widehat{\varphi}}$$

(Norm8) $$\frac{(g\Leftarrow\mathsf{q}_1(X)\wedge\mathsf{q}_2(X)\wedge\omega)\in\varphi}{(g\Leftarrow(\mathsf{q}_1\cup\mathsf{q}_2)(X)\wedge\omega)\in\varphi}$$

(Norm9) $$\frac{\begin{array}{c}(\mathsf{p}(f(X_1,\ldots,X_n))\Leftarrow\mathsf{p}_1(X_1)\wedge\cdots\wedge\mathsf{p}_n(X_n))\in\widehat{\varphi}\\ (\mathsf{q}(f(Y_1,\ldots,Y_n))\Leftarrow\mathsf{q}_1(Y_1)\wedge\cdots\wedge\mathsf{q}_n(Y_n))\in\widehat{\varphi}\end{array}}{((\mathsf{p}\cup\mathsf{q})(f(X_1,\ldots,X_n))\Leftarrow(\mathsf{p}_1\cup\mathsf{q}_1)(X_1)\wedge\cdots\wedge(\mathsf{p}_n\cup\mathsf{q}_n)(X_n))\in\widehat{\varphi}}$$

(Norm10) $$\frac{\begin{array}{c}(\mathsf{p}(X_1,\ldots,X_m)\Leftarrow\mathsf{q}_1(X_1)\wedge\cdots\wedge\mathsf{q}_k(X_k))\in\varphi\\ \bigwedge_{i=1}^{k}\rho_{\widehat{\varphi}}(\mathsf{q}_i)\neq\emptyset \qquad k\geq m\geq 2\end{array}}{(\mathsf{p}(X_1,\ldots,X_m)\Leftarrow\mathsf{q}_1(X_1)\wedge\cdots\wedge\mathsf{q}_m(X_m))\in\widehat{\varphi}}$$

(Norm11) $$\frac{\begin{array}{c}(\mathsf{p}(f(X_1,\ldots,X_n))\Leftarrow\mathsf{q}_1(X_1),\ldots,\mathsf{q}_k(X_k))\in\varphi\\ \bigwedge_{i=1}^{k}\rho_{\widehat{\varphi}}(\mathsf{q}_i)\neq\emptyset \qquad k\geq n\end{array}}{(\mathsf{p}(f(X_1,\ldots,X_n))\Leftarrow\mathsf{q}_1(X_1)\wedge\cdots\wedge\mathsf{q}_n(X_n))\in\widehat{\varphi}}$$

Table 2.5: Normalisation of an $\mathcal{H}_1$-formula.

(Norm1) introduces a *universal predicate* p_H, mapped to the entire Herbrand universe in the least model, i.e. $\rho_\varphi(\mathsf{p}_H) = H_\varphi$;

(Norm2) takes a clause that contains an *unrestricted variable* X, i.e. a variable only present in the conclusion, and produces an equivalent clauses where X is restricted, using the universal predicate p_H;

(Norm3) and (Norm4) decomposes conclusions that do not adhere to the normal form through introduction of auxiliary predicates;

(Norm5) and (Norm6) simplifies non-normal literals in preconditions by using clauses already in normal form;

(Norm7) ensures that clauses of the form $\mathsf{p}(X) \Leftarrow \mathsf{q}(X)$ becomes superfluous, by simply replicating all clauses where q occurs in the conclusion, replaced by p;

(Norm8) takes clauses that requires an intersection of unary predicates in the precondition, introduces an auxiliary predicate that denotes the intersection, and add a new clause for the intersection;

(Norm9) computes intersections by introducing a clause for every constructor, for which a clause exists in the normal form for each literal in the precondition; and[9]

(Norm10) and (Norm11) simply add all clauses in normal form, for which the precondition is satisfiable.

We may add a few comments to the normalisation procedure. The technique used in the rules (Norm5) and (Norm6) is related to *ordered resolution* [49], although in a restricted manner as we only apply it to preconditions. In general, unrestricted ordered resolution does not terminate on most interesting clauses, as it allows for composing arbitrarily large conclusions. However, when restricting it to preconditions it will always terminate, as it will only introduce strictly simpler clauses.

Rule (Norm8) introduces clauses for intersections, to allow the intersections to be calculated in (Norm9). Using this approach, we adhere to the normal form of Table 2.3 in §2.1 and furthermore we only add satisfiable clauses, thus obtaining the most succinct representation of the least model. Observe that satisfiability of a predicate in the least model, e.g. $\rho_{\widehat{\varphi}}(\mathsf{p}) \neq \emptyset$, in this case simply amounts to determining if any clause in $\widehat{\varphi}$ contain p in conclusion; as only satisfiable

[9] We assume here that variables in clauses are α-renamed to match this rule. A more rigorous definition would include a total substitution of each precondition, but this would unnecessarily complicate the definition.

predicates are added to $\widehat{\varphi}$ such a clause would imply non-emptiness. Alternatively we could have allowed the clauses denoting intersections in the normal form instead, thus following the customs of classical resolution theory. This would have resulted in a quadratic time normalisation, but an exponential time membership lookup in the least model. Instead, by precalculating all intersections, we have an exponential time normalisation, but linear time membership lookup. For us, this option is more appealing, as the membership lookup time is, as we shall see, crucial for the specialisation because it requires a thorough investigation of the least model.

We then let η_1 denote a function that performs the normalisation of Table 2.5; i.e. a function that given a formula $\varphi \in \mathcal{H}_1$, finds the least $\widehat{\varphi}$ that satisfies the rules. We then have the following result from [96]:

Lemma 2.14 *The function η_1 is an $\mathcal{H}_1$-normalisation operator.*

Note that to ensure that this function terminates, it is, of course, necessary that the rules (NORM3), (NORM4) can be applied at most once for each compound term in each clause, to avoid a continuous introduction of auxiliary predicates. Similarly, (NORM8) should only abide a unique naming scheme for intersections, to avoid that the same intersections may be computed an unbounded number of times. For the benefit of the practically inclined reader, we have included the pseudo-code for a succinct, yet efficient, implementation of the normalisation function (also known as a solver) in Appendix B.

2.4.4 Specialisation Operator

The specialisation is supposed to utilise information in a given normal form approximation $\widehat{\varphi}$ of a formula φ, to produce a new formula φ' that maintains the set of models smaller than or equal to $\rho_{\widehat{\varphi}}$, but may yield smaller approximations. One approach to define such an operator, is to define a function that safely eliminates variables from the formula.

First we shall require a formulation of which ground terms a variable may be substituted by in a clause.

Definition 2.15 (Substitution set) *The substitution set for $X \in \mathsf{Var}(\varphi)$ is given by $\mathcal{T}_\varphi(X) = \{X\theta \mid \rho_\varphi, \theta \models \omega \wedge (g \Leftarrow \omega) \in \varphi \wedge X \in \mathsf{Var}(g \Leftarrow \omega)\}$.*

Or plainly: the substitution set $\mathcal{T}_\varphi(X)$ denotes the set of ground terms that the

variable X can be replaced by, while still allowing the precondition of the clause it occurs in to be satisfiable in the least model ρ_φ.

Some of these substitution sets may be infinite and we are therefore particularly interested in the set $\mathcal{F}_\varphi$ of variables $X \in \mathsf{Var}(\varphi)$ for which $\mathcal{T}_\varphi(X)$ is finite. However, it may not always be feasible to determine the complete set $\mathcal{F}_\varphi$, or the corresponding substitution sets, and thus we shall say that a *finiteness substitution map* I_φ is permissable if $dom(I_\varphi) \subseteq \mathcal{F}_\varphi$ and $\forall X \in dom(I_\varphi) : I_\varphi(X) \supseteq \mathcal{T}_\varphi(X)$.

As neither the relaxation nor the normalisation renames variables, and as $\rho_{\widehat{\varphi}} \models \varphi$, it follows that for all $X \in \mathsf{Var}(\varphi)$ then $\mathcal{T}_\varphi(X) \subseteq \mathcal{T}_{\widehat{\varphi}}(X)$. Thus a permissable $I_{\widehat{\varphi}}$ is also a permissable I_φ, and the idea is then to use the former to perform a *complete expansion* of the formula. This is done by the function, $\beta_1 : \widehat{\mathbf{HC}} \to \mathbf{HC} \to \mathbf{HC}$, defined as follows:

$$\beta_1[\widehat{\varphi}](\varphi) = \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall X \in dom(\theta) : X\theta \in I_{\widehat{\varphi}}(X)\}$$

Note that $\beta_1[\widehat{\varphi}]$ is the identity function if there are no variables in φ with finite substitution sets in $I_{\widehat{\varphi}}$.

Lemma 2.16 *The function β_1 is a specialisation function.*

Proof Assume $\widehat{\varphi}$, φ and ρ, such that $\rho_{\widehat{\varphi}} \models \varphi$ and $\rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}}$. The proof burden then amounts to showing that $\rho \models \beta_1[\widehat{\varphi}](\varphi) \Leftrightarrow \rho \models \varphi$. The proof is twofold, first showing $\rho \models \beta_1[\widehat{\varphi}](\varphi) \Rightarrow \rho \models \varphi$:

$$\begin{array}{ll}
\rho \models \beta_1[\widehat{\varphi}](\varphi) & \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall X \in dom(\theta) : X\theta \in I_{\widehat{\varphi}}(X)\} \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall X \in dom(\theta) : X\theta \in \mathcal{T}_{\widehat{\varphi}}(X)\} \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall (g \Leftarrow \omega) \in \varphi : \ \rho_{\widehat{\varphi}}, \theta \models \omega\} \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall (g \Leftarrow \omega) \in \varphi : \ \rho, \theta \models \omega\} \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \rho \models \varphi\theta\} \\
\Rightarrow & \rho \models \varphi
\end{array}$$

He we use that $\forall X \in dom(\theta) : I_{\widehat{\varphi}}(X) \supseteq \mathcal{T}_{\widehat{\varphi}}(X)$ in the second step, and in the fourth step we use the assumption that $\rho \sqsubseteq_\varphi \rho_{\widehat{\varphi}}$. It then remains to show that $\rho \models \varphi \Rightarrow \rho \models \beta_1[\widehat{\varphi}](\varphi)$, which follows trivially:

$$\begin{array}{ll}
\rho \models \varphi & \\
\Rightarrow & \forall \theta : \ \rho \models \varphi\theta \\
\Rightarrow & \rho \models \bigcup \{\varphi\theta \mid dom(\theta) = dom(I_{\widehat{\varphi}}) \ \wedge \ \forall X \in dom(\theta) : X\theta \in I_{\widehat{\varphi}}(X)\} \\
\Rightarrow & \rho \models \beta_1[\widehat{\varphi}](\varphi)
\end{array}$$

That concludes the proof. □

$$\frac{\forall(\mathsf{p}(t) \Leftarrow \omega) \in \widehat{\varphi} : \ \mathsf{Pred}(\omega) \subseteq dom(F)}{\begin{array}{c}\forall(\mathsf{p}(f(X_1,\ldots,X_n)) \Leftarrow \mathsf{q}_1(X_1) \wedge \cdots \wedge \mathsf{q}_n(X_n)) \in \widehat{\varphi} : \\ \forall\langle t_1,\ldots,t_n\rangle \in F(\mathsf{q}_1) \times \cdots \times F(\mathsf{q}_n) : \\ f(t_1,\ldots,t_n) \in F(\mathsf{p})\end{array}}$$

$$\frac{(g_0 \Leftarrow g_1 \wedge \cdots \wedge g_l) \in \widehat{\varphi} \qquad \mathsf{q}(X) \in \{g_1,\ldots,g_l\} \qquad \mathsf{q} \in dom(F)}{I_{\widehat{\varphi}}(X) = F(\mathsf{q})}$$

Table 2.6: Computing a finiteness substitution map; $I_{\widehat{\varphi}}$.

It then only remains to show how $I_{\widehat{\varphi}}$ can be determined for $\mathcal{H}_1$, or more specifically, for the normal form in Table 2.3. Such a map can be obtained by the inductively defined procedure in Table 2.6.

The map is built by first finding an auxiliary map F of the unary predicates with a finite map in $\widehat{\varphi}$; i.e. $dom(F) \subseteq \mathcal{F}_\varphi$ and $\forall \mathsf{p} \in dom(F) : F(\mathsf{p}) = \rho_{\widehat{\varphi}}(\mathsf{p})$. A predicate is simply finite if all predicates in all preconditions of clauses defining it are finite. Once F is determined, it can be used to build $I_{\widehat{\varphi}}$ straightforwardly.

Example 2.17 (Example 2.10 continued) *Assume the finiteness substitution map* $I_{\widehat{\varphi}} = [X_{nb} \mapsto \{nb\}, X_{kab} \mapsto \{kab, m_\bullet\}]$*, then a complete expansion of the clause from Example 2.10 is:*

$$\begin{array}{l}\mathsf{net}(\langle X_{enc}, \{nb\}_{kab}\rangle) \Leftarrow \mathsf{net}(\langle\{\langle b, kab, na, nb\rangle\}_{ka}, X_{enc}\rangle) \\ \mathsf{net}(\langle X_{enc}, \{nb\}_{m_\bullet}\rangle) \Leftarrow \mathsf{net}(\langle\{\langle b, m_\bullet, na, nb\rangle\}_{ka}, X_{enc}\rangle)\end{array}$$

2.4.5 Simplification ordering

As the specialisation expands the clauses of the formula, an obvious choice of simplification ordering is an ordering where clauses are greater than their expanded counterpart.

$$\forall\varphi : \forall\varphi' : \quad \varphi \preceq_1 \varphi' \Leftrightarrow (\exists\widehat{\varphi} : \varphi = \beta_1[\widehat{\varphi}](\varphi'))$$

It follows that equivalence resolves to equality, and thus the check $\varphi' \prec \varphi$ in the iterative framework will amount to the test for identity. We then have the result:

Lemma 2.18 *The operator* $\preceq_1$ *is a simplification ordering.*

Proof We have trivially that $\preceq_1$ is a partial order. That it is also well-founded follows from the fact that any formula φ can only hold a finite set of variables:

a complete expansion (modulo identity) expands each clause into a finite set of new clauses holding a strictly lower number of variables, hence any formula can only be completely expanded a finite number of times. □

2.4.6 $\mathcal{H}_1$-Refinement Scheme

Finally, to show that the different ingredients work together as intended, we must show that they form an $\mathcal{H}_1$-refinement scheme.

Lemma 2.19 *The quadruple* $(\alpha_1, \eta_1, \beta_1, \preceq_1)$ *forms an* $\mathcal{H}_1$*-refinement scheme.*

Proof To prove this, we must show that for any φ and $\widehat{\varphi}$ then $\rho_{\alpha_1(\beta_1[\widehat{\varphi}](\varphi))} \sqsubseteq_\varphi \rho_{\alpha_1(\varphi)}$; i.e. that for all $t \in H_\varphi$ and $\mathsf{p} \in \mathsf{Pred}(\varphi)$ then $t \in \rho_{\alpha_1(\beta_1[\widehat{\varphi}](\varphi))}(\mathsf{p})$ implies $t \in \rho_{\alpha_1(\varphi)}(\mathsf{p})$. This is shown by contradiction.

Assume that $t \in \rho_{\alpha_1(\beta_1[\widehat{\varphi}](\varphi))}(\mathsf{p})$ and $t \notin \rho_{\alpha_1(\varphi)}(\mathsf{p})$. As $\rho_{\beta_1[\widehat{\varphi}](\varphi)} \sqsubseteq_\varphi \rho_\varphi$ (in particular $\rho_{\beta_1[\widehat{\varphi}](\varphi)} = \rho_\varphi$ if $\rho_{\widehat{\varphi}} \models \varphi$) and $\rho_\varphi \sqsubseteq_\varphi \rho_{\alpha_1(\varphi)}$, the latter implies that $t \notin \rho_{\beta_1[\widehat{\varphi}](\varphi)}(\mathsf{p})$. Thus, $t \in \rho_{\alpha_1(\beta_1[\widehat{\varphi}](\varphi))}(\mathsf{p})$ is an over-approximation introduced by the relaxation α_1 on $\beta_1[\widehat{\varphi}](\varphi)$, but not on φ. However, we then easily arrive at a contradiction, using induction in the length of the derivation sequence establishing $\beta_1[\widehat{\varphi}](\varphi)$; α_1 only introduces over-approximation when decoupling variables violating $H1$, but as $\beta_1[\widehat{\varphi}]$ only eliminates variables, then any variable pair present in $\beta_1[\widehat{\varphi}]$ is also present in φ. This establishes the result we seek. □

Thus the refinement scheme $(\alpha_1, \eta_1, \beta_1, \preceq_1)$ is suitable for the iterative framework of §2.3.2. We then give a slightly larger example, showing the inter-operability of the framework and the need for the iterative behaviour.

Example 2.20 *Assume that* φ *consists of the clauses:*

$$
\begin{array}{lllll}
(1) & \mathsf{q}(a) & & & \\
(2) & \mathsf{q}(b) & & & \\
(3) & \mathsf{p}(X, Y) & \Leftarrow & \mathsf{q}(X) \wedge \mathsf{q}(Y) \\
(4) & \mathsf{p}(b, f(Y)) & \Leftarrow & \mathsf{p}(b, Y) \\
(5) & \mathsf{r}(X, g(Y, Y)) & \Leftarrow & \mathsf{p}(X, Y)
\end{array}
$$

We see that only (5) *violates* $H1$*, and we get that applying* α_1 *to this clause yields:*

$$
\begin{array}{lllll}
(5a) & \mathsf{r}(X, g(Y', Y)) & \Leftarrow & \mathsf{p}(X', Y') \wedge \mathsf{p}(X', Y) \wedge \mathsf{aux}(X) \\
(5b) & \mathsf{aux}(X) & \Leftarrow & \mathsf{p}(X, Y') \wedge \mathsf{p}(X, Y)
\end{array}
$$

Introducing the fresh variables X' *and* Y'*, and the auxiliary predicate* aux.[10] *Using the normalisation,* η_1*, we can then find an equivalent formula of the clauses* (1)*,* (2)*,* (3)*,* (4)*,* (5a)*, and* (5b) *in normal form, denoted* $\widehat{\varphi}_1$*. Indeed, we find:*

$$\begin{array}{lcl}
\rho_{\widehat{\varphi}_1}(\mathsf{q}) & = & \{a, b\} \\
\rho_{\widehat{\varphi}_1}(\mathsf{p}) & = & \{\langle X, Y\rangle \mid X, Y \in \{a, b\}\} \cup \\
 & & \{\langle b, f^n(Y)\rangle \mid Y \in \{a, b\} \wedge n \geq 1\} \\
\rho_{\widehat{\varphi}_1}(\mathsf{r}) & = & \{\langle X, g(Y', Y)\rangle \mid X \in \{a, b\} \wedge Y', Y \in \{f^n(a) \vee f^n(b) \mid n \geq 0\}\}
\end{array}$$

Where we write $f^n(t)$ *for the term* t *enclosed by* n *applications of the constructor* f*. By inspection, we see that the result is precise for* p *and* q*, whereas for* r*, it is imprecise. However, the model reveals that the variable* X *in the clause* (5) *has the finite substitution set* $\{a, b\}$*, and thus we may perform a complete expansion of this clause*[11] *and get:*

$$\begin{array}{lcl}
(5.1) \quad \mathsf{r}(a, g(Y, Y)) & \Leftarrow & \mathsf{p}(a, Y) \\
(5.2) \quad \mathsf{r}(b, g(Y, Y)) & \Leftarrow & \mathsf{p}(b, Y)
\end{array}$$

Analogously to above, these clauses may be relaxed (now requiring no auxiliary predicates), and then normalised together with (1)*,* (2)*,* (3)*, and* (4)*. This produces a new normal form,* $\widehat{\varphi}_2$*, which is a more precise approximation of* r*:*

$$\begin{array}{lcl}
\rho_{\widehat{\varphi}_2}(\mathsf{r}) & = & \{\langle a, g(Y', Y)\rangle \mid Y', Y \in \{a, b\}\} \cup \\
 & & \{\langle b, g(Y', Y)\rangle \mid Y', Y \in \{f^n(a) \vee f^n(b) \mid n \geq 0\}\}
\end{array}$$

Again, the model reveals a variable with finite substitution set; namely Y *in* (5.1) *which has the finite substitution set* $\{a, b\}$*. A complete expansion of this clause then gives:*

$$\begin{array}{lcl}
(5.1.1) \quad \mathsf{r}(a, g(a, a)) & \Leftarrow & \mathsf{p}(a, a) \\
(5.1.2) \quad \mathsf{r}(a, g(b, b)) & \Leftarrow & \mathsf{p}(a, b)
\end{array}$$

Which are both within $\mathcal{H}_1$ *and thus they can be normalised together with* (1)*,* (2)*,* (3)*,* (4)*, and the relaxation of* (5.2)*. This, finally, produces the normal form,* $\widehat{\varphi}_3$*, which is the most precise approximation of* r *we can obtain:*

$$\begin{array}{lcl}
\rho_{\widehat{\varphi}_3}(\mathsf{r}) & = & \{\langle a, g(Y, Y)\rangle \mid Y \in \{a, b\}\} \cup \\
 & & \{\langle b, g(Y', Y)\rangle \mid Y', Y \in \{f^n(a) \vee f^n(b) \mid n \geq 0\}\}
\end{array}$$

Hence, to find the most precise model for φ*, three iterations of the algorithm was required.*

We should remark that $\widehat{\varphi}_3$ *still does not represent the least model of* φ*, as the two occurrences of* Y *in the conclusion of* (5.2) *remains decoupled. However, a finite description of the least model of* φ*, in the normal form of Table 2.3 in §2.1, does not exist, and thus we cannot hope to obtain it.*

[10] The precondition of (5a) contains some superfluous literals. There are, of course, engineering tricks to avoid this overhead, some of which are mentioned in §B.2 of Appendix B.

[11] Both variables in (3) also have finite substitution sets, but as this clause is already in $\mathcal{H}_1$ the specialisation will not improve precision, and thus is less interesting.

1. $\mathsf{net}(\langle a, na\rangle)$
2. $\mathsf{net}(\langle b, \{\langle X_a, X_{na}, nb\rangle\}_{kb}\rangle) \Leftarrow \mathsf{net}(\langle X_a, X_{na}\rangle)$
3. $\mathsf{net}(\langle\{\langle b, kab, X_{na}, nb\rangle\}_{ka}, \{\langle a, kab\rangle\}_{kb}\rangle) \Leftarrow \mathsf{net}(\langle b, \{\langle a, X_{na}, nb\rangle\}_{kb}\rangle)$
4. $\mathsf{net}(\langle X_{enc}, \{nb\}_{kab}\rangle) \wedge \mathsf{a\_key}(kab) \Leftarrow \mathsf{net}(\langle\{\langle b, kab, na, nb\rangle\}_{ka}, X_{enc}\rangle)$
5. $\mathsf{b\_key}(kab) \Leftarrow \mathsf{net}(\langle\{\langle a, kab\rangle\}_{kb}, \{nb\}_{kab}\rangle)$

Figure 2.5: Specialised clauses of single instance Yahalom.

2.5 Verifying Protocols

Having presented the iterative framework, as well as an instance of it using $\mathcal{H}_1$, we now return to protocol analysis. In §2.2 we gave an intuitive presentation of a translation scheme from protocol narrations into Horn clauses, exemplified by the key-distribution protocol Yahalom, both when modelling a single instance of the protocol in §2.2.1 as well as multiple instances of the protocol in §2.2.2. We also showed how a generic network attacker could be formulated in Horn clauses, such that the protocol descriptions could be used for analysing for various security properties. In this section, we shall show how the $\mathcal{H}_1$-instance of the iterative framework can be used to find suitable models of these protocol analysis problems; in particular we shall validate the confidentiality and integrity of the Yahalom protocol.

2.5.1 A Single Instance of Yahalom

The conjunction of protocol description of a single instance of the Yahalom protocol, Figure 2.1 in §2.2.1, and the network attacker, Figure 2.3 in §2.2.3, constitutes an instance of an analysis problem. This problem falls, as we discussed in §2.3.1, outside the known decidable classes of Horn clauses, and thus we must rely to the iterative framework to deliver us a suitable model of it.

The $\mathcal{H}_1$-instance of the iterative framework requires only a single iteration to produce such a model; after the first iteration the algorithm produces the specialised clauses shown in Figure 2.5. We omitted the clauses for the attacker, as all variables occurring in them have trivially infinite substitution sets and, thus, these clauses remain unchanged.

The first observation is that all the clauses are within $\mathcal{H}_1$, thus the algorithm has found the least model of the analysis problem; i.e. the analysis estimate is complete. Inspecting the analysis estimate, $\widehat{\varphi}$, also reveals that both a_key and b_key may only be mapped to kab (i.e. the constant kab has been substituted for the variable X_{ab}). This proves that one instance of the protocol will always successfully establish the correct shared key between the intended principals, hence we have certified that the protocol satisfies integrity. We also find that $ka, kb, kab \notin \rho_{\widehat{\varphi}}(\mathsf{dy})$, which certifies that confidentiality of all the shared keys is preserved. Both of these results were expected as Yahalom is considered a secure protocol (cf. [31, 105]).

The analysis estimate provides us additional interesting insight. Namely, that all variables except X_a and X_{na} in step 2, X_{na} in step 3, and X_{enc} in step 4 can be completely expanded into their intended values, which is also apparent from the specialised clauses in Figure 2.5. Hence, these four variables represent the only elements of the protocol that the attacker can possibly interfere with; it can in fact get them substituted for arbitrary ground terms. However, the result shows that such an invalid substitution will result in a rejection of the key establishment at a later point and, thus, it will at most pose an annoyance to the principals attempting to establish a key. Such an attack is called a *denial-of-service* attack, and although it may be considered a vulnerability in the protocol, it is not a flaw.

2.5.2 Multiple instances of Yahalom

Turning to the more interesting analysis problem with multiple instances and multiple sessions, namely the conjunction of the protocol description in Figure 2.2 in §2.2.2 and the network attacker in Figure 2.3 in §2.2.3, a single iteration of the algorithm suffices for certifying confidentiality: $\forall i, j : \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor a_j \rfloor) \notin \rho_{\widehat{\varphi}}(\mathsf{dy})$.

Integrity, on the contrary, requires more than one iteration of the iterative framework. Indeed, the first iteration yields the approximation that the predicates a_key and b_key may be mapped to tuples of the form $\langle \lfloor a_i \rfloor, \lfloor a_j \rfloor, \lfloor kab \rfloor(\lfloor a_{i'} \rfloor, \lfloor a_{j'} \rfloor) \rangle$ for all $i, i', j, j' \in \{1, 2, 3\}$. In other words, any keys generated as part of a protocol run can be distributed to any combination of principals $\lfloor a_i \rfloor$ and $\lfloor a_j \rfloor$. This, obviously, violates integrity, but at least it shows that the mappings of a_key and b_key are finite. Hence the clauses can be specialised, and already after the second iteration we arrive at the specialised clauses presented in Figure 2.6. The description is slightly abbreviated in that each clause is quantified over $i, j \in \{1, 2, 3\}$.

We notice first that the iterative framework was unable to produce clauses

0. $\mathsf{a}(\lfloor a_i \rfloor), \mathsf{b}(\lfloor b_j \rfloor)$
1. $\mathsf{net}(\langle a_i, \lfloor na \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)\rangle) \Leftarrow \mathsf{a}(\lfloor a_i \rfloor) \wedge \mathsf{b}(\lfloor b_j \rfloor)$
2. $\mathsf{net}(\langle \lfloor b_j \rfloor, \{\langle X_a, X_{na}, \lfloor nb \rfloor(X_a, \lfloor b_j \rfloor)\rangle\}_{k(\lfloor b_j \rfloor, s)}\rangle) \Leftarrow \mathsf{net}(\langle X_a, X_{na}\rangle) \wedge \mathsf{b}(\lfloor b_j \rfloor)$
3. $\mathsf{net}(\langle \{\langle \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor), X_{na}, X_{nb}\rangle\}_{k(\lfloor a_i \rfloor, s)}, \{\langle a_i, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)\rangle\}_{k(\lfloor b_j \rfloor, s)}\rangle)$
   $\Leftarrow \mathsf{net}(\langle \lfloor b_j \rfloor, \{\langle \lfloor a_i \rfloor, X_{na}, X_{nb}\rangle\}_{k(\lfloor b_j \rfloor, s)}\rangle) \wedge \mathsf{a}(\lfloor a_i \rfloor) \wedge \mathsf{b}(\lfloor b_j \rfloor)$
4. $\mathsf{net}(\langle X_{enc}, \{X_{nb}\}_{\lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)}\rangle) \wedge \mathsf{a\_key}(\lfloor a_i \rfloor, \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor))$
   $\Leftarrow \mathsf{net}(\langle \{\langle \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor), \lfloor na \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor), X_{nb}\rangle\}_{k(\lfloor a_i \rfloor, \lfloor b_j \rfloor)}, X_{enc}\rangle) \wedge$
   $\mathsf{a}(\lfloor a_i \rfloor) \wedge \mathsf{b}(\lfloor b_j \rfloor)$
5. $\mathsf{b\_key}(\lfloor a_i \rfloor, \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor))$
   $\Leftarrow \mathsf{net}(\langle \{\}_{k(\lfloor b_j \rfloor, s)\langle \lfloor a_i \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)\rangle}, \{\lfloor nb \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)\}_{\lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)}\rangle) \wedge$
   $\mathsf{b}(\lfloor b_j \rfloor)$

Figure 2.6: Specialised clauses of multiple instances Yahalom; $i, j \in \{1, 2, 3\}$.

within $\mathcal{H}_1$. The generic formulation for nonces, using initiator and responder names to generate new nonces, forces step 2 beyond $\mathcal{H}_1$, as the variable X_a has infinite substitution set and occurs multiple times in the conclusion. This over-approximation introduced by the algorithm carries over to step 3 and 4, in which the variables representing b's nonce, X_{nb}, are found to have infinite substitution set.

Fortunately, the analysis estimate is still precise enough to certify the integrity of Yahalom with multiple instances and sessions; i.e. a_key and b_key may only be mapped to tuples of the form $\langle \lfloor a_i \rfloor, \lfloor b_j \rfloor, \lfloor kab \rfloor(\lfloor a_i \rfloor, \lfloor b_j \rfloor)\rangle$ for $i, j \in \{1, 2, 3\}$.

It is noteworthy that merely applying the relaxation and then the solver, equivalent to a single iteration, in the style of [59], would yield the unsatisfactory and incorrect result that all keys generated by the solver could be accepted at any b_j as coming from any a_i. This is the false positive that any context-independent analysis (see §5.6) will face when analysing Yahalom. In fact, even the approach of manually expanding the analysis with the information about sender and recipient of each message, results in this false positive for Yahalom [24]. Interestingly, the iterative algorithm performs this expansion, along with other expansions, but automatically, thus allowing a clearer protocol description and leaving less room for human errors.

2.6 Discussion

In §2.2 we sketched a methodology for specifying static analyses of cryptographic protocols; in particular we showed how to model and analyse the non-trivial key-distribution protocol Yahalom. To avoid too much digression in the presentation and to adhere to the topic of the chapter, we omitted some insights behind some of the choices we made. These are discussed in detail below.

2.6.1 Alternative Modelling Choices

The modelling reflects important design choices. First we see that all principals are divided into the disjoint sets of initiators, responders and servers, which is the classical modelling approach. But more general alternatives, letting principals act in more than one role, could be chosen. These could easily be accommodated by Horn clauses as well, by simply using the same set of ground terms to represent different roles; e.g. principals may act as both initiators and responders in Figure 2.2 if $\rho(\mathsf{a}) = \rho(\mathsf{b}) = \{i_1, i_2, i_3\}$. Interestingly, this would introduce yet another decision, namely whether the principals should use the same key for acting in both roles or use different keys. Again both options are translatable to Horn clauses.

Secondly, there is the choice of modelling the pre-existing key between the principals and the server with the constructor k and the generated key with the constructor kab. The different constructors reflect the typical distinction between long term keys and short terms keys, and that only long term keys can be used for generating short term keys. In the current scenario, this distinction does not make a difference. But it would, if the server could also act in the responder role, as the generated short term key might then be used for generating other short term keys.

All of the above variations require loosening some of the assumptions, and it is important to be conscious that modelling choices as more general scenarios may also introduce new attacks. In this chapter we have only considered the usual setting, but later, when verifying protocols with our developed analysis in Chapter 6, we will consider other alternatives as well.

2.6.2 Security Properties

A conjunction of one of the formulae in Figure 2.1 or Figure 2.2 with the attacker in Figure 2.3 constitutes an instance of an analysis problem, and the least model of such a problem allows us to investigate certain properties of the protocol.

When we are interested in guaranteeing that the protocol has some important properties, we must ensure that the analysis captures all possible violations. In other words, it must be sound. Hence, we must review the two abstraction techniques, introduced in §2.2.2, for modelling an unbounded number of instances and sessions.

In [39] it is shown that it is always sufficient to consider only a bounded number of principals. In particular, it is only necessary to consider two principals for verifying confidentiality, and in general, for any property, $(k + 1)$ is enough, where k is the number of principal variables used in the specification of that property.[12] Thus, for example, three principals are enough for verifying integrity, as we validated Yahalom for through the predicates a_key and b_key.

However, instead of restricting the number of principals, and thus the semantics of the protocol, we follow the approach of [29] and project the semantic behaviour of all principals onto this limited number of principals. This projection is onto the canonical values in the analysis estimate, rather than onto the semantic behaviour as in [39], and thus a verification of this type covers the general case of an unlimited number of principals.

The other abstraction, coalescing the generated nonces in each instance of the protocol into one equivalence class for each syntactic occurrence, is also safe with respect to most security properties. The abstraction means that the analysis is unable to distinguish two messages generated from different sessions but at the exact same point of the protocol. This, again, means that the analysis can capture all but a specific type of replay attacks, namely the type where a message from a previous session is reused at the same step of the protocol. This specific kind of replay attacks are usually of little interest and easily caught by the implementation. Nevertheless, in §7.2 we shall discuss how the analysis can be extended to capture also this kind of attack.

Returning to the properties, the analysis obviously allows for verifying confidentiality of a particular ground term t, as this merely resolves to investigating whether t belongs to the possible bindings of the predicate dy.

[12] An interesting observation is that k seems to enough for protocols where principals have unique roles (e.g. $A \neq B$). However, as this is an unproven conjecture, we have chosen to abide by the rule of $k + 1$ throughout this dissertation.

Integrity is the dual of confidentiality, and with respect to protocols, this means that the message received is the one intended. This can also be verified through the analysis, as one may inspect that all possible variable bindings within each step correspond to those intended. Usually only specific parts are interesting in a protocol, and for Yahalom these are the bindings of predicates a_key and b_key; i.e. that the key distributed to A and B after a successful run of the protocol can only be Kab.

Finally, one may enrich all encryptions with information about the sender and the intended recipient in the style of [22, 24], allowing for verifying message authentication. For the remainder of this dissertation we shall only be concerned with the verification of protocols up to integrity and confidentiality, but in §7.2 we shall revisit this subject and discuss how other properties could be verified as well, by the simple use of annotations.

2.7 Concluding Remarks

In this chapter we have shown how Horn clauses allow for a clear and intuitive specification of protocol analysis problems. Unfortunately, as it often is with clean and simple analyses, the resulting instances of analysis problems are outside any of the known decidable fragments of Horn clauses, threatening the analysis itself with being a mere theoretical exercise.

Contrary to the traditional approach of modifying the analysis and introduce abstractions that would force it into a decidable fragment, we proposed an iterative framework for solving unrestricted Horn clauses, guaranteeing termination and soundness, and in some cases achieving even completeness. The suggested algorithm uses a known decidable class of Horn clauses to produce a sound approximation method; the latter being achieved by iteratively specialising the analysis problem and improving the analysis estimate, until either no further improvements can be made or the analysis estimate is complete. We also presented an instance of the iterative framework, using the decidable class $\mathcal{H}_1$, and with it we successfully validated the Yahalom key-distribution protocol, a protocol infamous for being particularly difficult to analyse.

So it seems that modelling and analysing cryptographic protocols using Horn clauses is both reasonable and viable. However, the manual method used in this chapter, although the most common and typical approach, is both tedious and error-prone. These issues do not become any less serious when we enlarge the scope from a single protocol to the simulation of an entire system, where several protocols may be applied – in this case the manual method is simply not

tractable, nor trustworthy.

In the remainder of this dissertation, we will work on this issue and develop a process calculus specifically designed to bridge the gap between the analysis formulation and a human comprehensible model.

Chapter 3

Cryptographic Components

The Horn clause approach gave us some insights on a succinct, clear model for cryptographic protocols. Predicates were used for modelling the communication medium and terms were used to model the cryptographically composed messages.

In a process calculus we would now expect constructs such as $a.P$ to denote the sequential composition of some action, a, and the continuation process, P. We would also expect constructs for composition of concurrent processes, $P_1 \mid P_2$, where the processes would communicate using actions for sending and receiving messages.

Imagine then that a process calculus employs two operations, $\mathsf{out}\, e$ and $\mathsf{in}\, p$, for transmission and reception, respectively, of cryptographically composed messages. Output involves some encoded message, e, and input uses a pattern, p, specifying which messages that are accepted. Coming from Horn clauses, it seems natural to construct messages with a term language and allow expressions such as $\mathsf{out}\, \{m\}_k$ for the transmission of the message m encrypted with the key k. Likewise, we would expect that the input pattern specifies the structure of the messages accepted and allows introduction of new variables. For example, we could write $\mathsf{in}\, \{x\}_k$ for the input which matches any message encoded with key k and binds contents of the encryption to a variable x.

Those are typical constructs in process calculi with cryptography; e.g. LySa

[22, 24] and the Spi Calculus [6]. In fact, from these languages we see that the modelling of cryptography and the underlying communication model may be developed independently, as both LySa and the Spi Calculus are based on the π-calculus, which itself has no support for cryptography. This motivates an independent development of the cryptographic components, which is the topic of this chapter.

Contrary to the above-mentioned calculi, we will allow composite terms on input. For example, writing $\mathsf{in}\,\langle x, \{x\}_k\rangle$ should be allowed – clearly this pattern would match any value of the form $\langle\langle v, \{v\}_k\rangle\rangle$. This seems a natural choice, this is basically how we modelled it in Horn clauses and they are clearly more intuitive and succinct than an tedious encoding of how the pattern should be decomposed in a step-wise manner.

In the remainder of this chapter we develop the syntax and semantics of the cryptographic components for the calculus. This is done in two steps: (1) we create the intuitive model that directly reflects the encoding in Horn clauses, and (2) we extend the simple model with asymmetric cryptography, which introduces some non-trivial problems. After the development, we shall justify the components, by showing that they satisfy some attractive properties and show they may be used to express protocol narrations in an unambiguous way.

3.1 A Mundane Approach

3.1.1 Syntax

The basic building blocks used for modelling cryptography are *encodings* and *patterns*. These are defined as term languages and their abstract syntax is listed in Table 3.1. Patterns require introduction of yet another syntactic category, *decodings*, denoted d. We also introduce the countably infinite set of names, $\mathcal{N}$, ranged over by $k, n, m, \ldots$, and the countably infinite set of variables $\mathcal{X}$, ranged over by $x, y, z, \ldots$, and assume that these sets are disjoint.

Encodings are built up from names, representing all basic elements such as text, nonces, symmetric keys or time stamps, all coalesced into the same syntactic class, and variables, which are placeholders that may be substituted for names or composite encodings during execution. The tuple construct $\langle e_1, \ldots, e_k\rangle$ is used for concatenation of encodings and finally the construct $\{e_1\}_{e_0}$ represents symmetric encryption of an encoding e_1 using the key e_0. In this setting a *value* is a closed encoding, that is an encoding with no variables, and we denote it

e	::=	n	Name, $n \in \mathcal{N}$
	|	x	Variable, $x \in \mathcal{X}$
	|	$\langle e_1, \ldots, e_k \rangle$	Tuple
	|	$\{\!\{e_1\}\!\}_{e_0}$	Symmetric encryption
d	::=	e	Comparison
	|	$\langle d_1, \ldots, d_k \rangle$	Decompose tuple
	|	$\{\!\{d\}\!\}_e$	Symmetric decryption
p	::=	$d \triangleleft X$	Pattern, $X \subseteq \mathcal{X}$

Table 3.1: Syntax for encodings, decodings and patterns.

$v \in \mathsf{Val}$.

Upon receipt of a value, the recipient party may perform certain actions in order to decompose the encoding and obtain the intended information. These actions are modelled as decodings and there are three main actions the recipient may perform: (1) compose a new value from known data and bit-wise compare it to the received message, (2) decompose a tuple, and (3) decrypt encrypted value using a known key. It is important to note that a key may itself be a complex encoding, but must not contain any decomposition constructors, as keys should be known prior to use.

The decompositions combined with the set of variables, X, the recipient expects to bind to a value within the matching, establishes a pattern. The intent of these components is best illustrated through a small example.

Example 3.1 *Assume the existence of a matching operator* as *for matching a value* v *against a pattern* p*, written* $v \,\mathsf{as}\, p$*, and an operator* $.$ *for sequential composition. The former could be used for decomposition or decryption in a process calculus, and the general idea is then that a process* $(v \,\mathsf{as}\, e \triangleleft X).P$ *would first verify that* v *structurally matches* e*; if this is the case then the continuation process* P *is updated with any newly bound variables in* X*, and executed; otherwise the execution will be blocked.*

The matching $(\langle n, m \rangle \,\mathsf{as}\, \langle y, m \rangle \triangleleft \{y\}).P$ *should succeed because the tuple* $\langle n, m \rangle$ *matches* $\langle y, m \rangle$*. This results in the substitution of the name* n *for the variable* y *in the continuation process* P*. The matching* $(\langle n, m \rangle \,\mathsf{as}\, \langle y, y \rangle \triangleleft \{y\}).P$ *on the other hand would not succeed, as the distinct names* n *and* m *cannot simultaneously match the pattern* y*, and thus further execution is garbled.*

Following the process calculus tradition (as introduced by the λ-calculus [35, 12]), we shall handle names and variables based on their *scope*. Scopes are introduced by a *defining occurrence*, whereafter all uses of the name or variable is linked to that occurrence, and therefore said to be within its scope. This allows for reusing the same syntactic identifier for different names or variables, while maintaining their distinction in the semantics.[1] In patterns, $d \triangleleft X$, we explicitly list the variables that are defined, X, whereas we leave the defining occurrence of names to the surrounding calculus.

Whenever a variable or name is contained by a scope we say that it is *bound*. The opposite of bound, that is when a name or variable occurs outside any scope, is usually referred to as *free*. In that respect, we introduce a function for finding free names, fn, and free variables, fv, in an encoding, decoding or pattern. Their definitions are straightforward and given in Table 3.2 and Table 3.3; all names and variables occur free in encodings and decodings, as these components contain no defining occurrences of either, whereas only the free variables in d not included in the set X occurs free in the pattern $d \triangleleft X$. Sometimes we may also refer to the bound variables in a pattern, written $\mathsf{bv}(p)$, which is directly as indicated by the syntax; namely $\mathsf{bv}(d \triangleleft X) = X$.

Remark 3.2 *A name or variable is always either free or bound in an encoding or pattern, and never both.*

$$
\begin{aligned}
\mathsf{fn}(n) &\stackrel{\text{def}}{=} \{n\} \\
\mathsf{fn}(x) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fn}(\langle e_1, \ldots, e_k \rangle) &\stackrel{\text{def}}{=} \mathsf{fn}(e_1) \cup \cdots \cup \mathsf{fn}(e_k) \\
\mathsf{fn}(\{\!|e_1|\!\}_{e_0}) &\stackrel{\text{def}}{=} \mathsf{fn}(e_0) \cup \mathsf{fn}(e_1) \\
\mathsf{fn}(\langle d_1, \ldots, d_k \rangle) &\stackrel{\text{def}}{=} \mathsf{fn}(d_1) \cup \cdots \cup \mathsf{fn}(d_k) \\
\mathsf{fn}(\{\!|d|\!\}_e) &\stackrel{\text{def}}{=} \mathsf{fn}(e) \cup \mathsf{fn}(d) \\
\mathsf{fn}(d \triangleleft X) &\stackrel{\text{def}}{=} \mathsf{fn}(d)
\end{aligned}
$$

Table 3.2: Free names; fn.

[1] This feature is widely used in programming languages; e.g. in `for` loops where the same iterator name may be reused throughout the entire program.

$$
\begin{array}{rcl}
\mathsf{fv}(n) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(x) & \stackrel{\text{def}}{=} & \{x\} \\
\mathsf{fv}(\langle e_1, \ldots, e_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_1) \cup \cdots \cup \mathsf{fv}(e_k) \\
\mathsf{fv}(\{e_1\}_{e_0}) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_0) \cup \mathsf{fv}(e_1) \\
\mathsf{fv}(\langle d_1, \ldots, d_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fv}(d_1) \cup \cdots \cup \mathsf{fv}(d_k) \\
\mathsf{fv}(\{\!|d|\!\}_e) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup \mathsf{fv}(d) \\
\mathsf{fv}(d \triangleleft X) & \stackrel{\text{def}}{=} & \mathsf{fv}(d) \backslash X
\end{array}
$$

Table 3.3: Free variables; fv.

3.1.2 Semantics

Having presented the syntax of the components, we now extend them with a formal meaning in the form of a semantics. For this we choose to rely on a *substitution-based semantics*; i.e. a semantics that uses substitutions to collect bindings to variables, as they occur, and then replace all future uses of the variables with their corresponding value binding. Substitution-based semantics has the advantage of being, arguably, the most comprehensible type of semantics, and it also appears to be the most common semantics used for expressing process calculi. The semantics choice implies that we also require the calculus to adopt the cryptographic components to use a similar semantics, although it should be fairly easy to convert the semantics given in this chapter to other types of semantics.

Substitution-based semantics are only concerned with *closed* processes; i.e. processes that contain no free names or variables. Thus it follows that the semantics of encodings are merely values, as the semantics will ensure that any variable within an encoding will be replaced by a value before that encoding is used. The semantics of pattern matching, on the other hand, must record any new binding of variables. Formally, the semantics needs to specify the criterion for a value v to successfully match a pattern p, as well as produce a partial map

$$\theta \quad : \quad \mathcal{X} \to \mathsf{Val}$$

mapping the bound variables in the matching to their respective value.[2]

As for Horn clauses, we shall write $[\,]$ for the empty map and $\theta[x \mapsto v]$ for the substitution that is like θ except it maps x to v. Furthermore, we define an

[2]We slightly abuse notation here, as we use the same symbol for both substitutions on Horn clauses and on the cryptographic components. This is chosen because the substitutions will be used analogously analysis-wise later and thus the notation eases this transition.

equivalence relation for substitutions, $=_s$, as

$$\theta_1 =_s \theta_2 \qquad \Leftrightarrow \qquad \forall x \in (dom(\theta_1) \cap dom(\theta_2)) : \theta_1(x) = \theta_2(x)$$

The equality operator $=_s$ ensures that the substitutions agree on the mapping of the variables they share; i.e. if a variable is in the domain, indicated by the operator $dom(\cdot)$, of both substitutions then it must be mapped to the same value in both substitutions. Thus, we may unambiguously unite two substitutions, $\theta_1 \cup \theta_2$, whenever $\theta_1 =_s \theta_2$.

The judgement for the semantics of pattern matching takes the form $v \text{ as } p : \theta$ and states that the value v correctly matches the pattern p, giving rise to the variable mappings in θ. As usual, this is defined as a set of inference rules, as listed in Table 3.4. Whenever a value v is matched against a variable x we use the rule (Bind), specifying that the resulting substitution is the mapping of x to v. We know that all variables that have been bound prior to the pattern matching are already replaced by their corresponding values, and thus the rule (Bind) simply requires the variable x to belong to the list of variables intended to be bound in the matching X. Symmetric decryption (SDec) requires both the key and the content to match the pattern, and tuples, (ETup) and (DTup), require each sub-pattern to match. The semantics assumes that all sub-patterns are matched simultaneously and ensures that possible, multiple mappings of variables are identical using the $=_s$ relation.

(Name)	$n \text{ as } n \lhd X : [\,]$	
(Bind)	$\dfrac{x \in X}{v \text{ as } x \lhd X : [x \mapsto v]}$	
(ETup)	$\dfrac{\bigwedge_{i=1}^{k} \; v_i \text{ as } e_i \lhd X : \theta_i}{\langle v_1, \ldots, v_k \rangle \text{ as } \langle e_1, \ldots, e_k \rangle \lhd X : \theta_1 \cup \cdots \cup \theta_k}$	if $\forall i, j : \theta_i =_s \theta_j$
(SEnc)	$\dfrac{v_0 \text{ as } e_0 \lhd X : \theta_0 \qquad v_1 \text{ as } e_1 \lhd X : \theta_1}{\{v_1\}_{v_0} \text{ as } \{e_1\}_{e_0} \lhd X : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$
(DTup)	$\dfrac{\bigwedge_{i=1}^{k} \; v_i \text{ as } d_i \lhd X : \theta_i}{\langle v_1, \ldots, v_k \rangle \text{ as } \langle d_1, \ldots, d_k \rangle \lhd X : \theta_1 \cup \cdots \cup \theta_k}$	if $\forall i, j : \theta_i =_s \theta_j$
(SDec)	$\dfrac{v_0 \text{ as } e \lhd X : \theta_0 \qquad v_1 \text{ as } d \lhd X : \theta_1}{\{v_1\}_{v_0} \text{ as } \{\!|d|\!\}_e \lhd X : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$

Table 3.4: Pattern matching; $v \text{ as } p : \theta$.

The resulting substitution θ of a pattern matching is supposed to be used to update any continuation process with the new bindings to variables, that the matching resulted in. Therefore, in addition to the semantics, we need a formal

definition of substitution of values for variables in the cryptographic components. This is given in Table 3.5 and includes no surprises. Substitution on encodings is straightforward, whereas on patterns it must respect possible rebindings of the variables.

Remark 3.3 *Later, in Chapter 4 we shall also allow free α-conversion of names; i.e. renaming bound names. This requires an additional definition of replacing a name by another name in encodings and patterns, but as these components contain no defining occurrences of names, the definitions are analogous to that of substitution in the case that a variable is never redefined.*

$$
\begin{array}{rcl}
n[x \mapsto v] & \stackrel{\text{def}}{=} & n \\
y[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} v & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
\langle e_1, \ldots, e_k \rangle [x \mapsto v] & \stackrel{\text{def}}{=} & \langle e_1[x \mapsto v], \ldots, e_k[x \mapsto v] \rangle \\
(\{\!|e_1|\!\}_{e_0})[x \mapsto v] & \stackrel{\text{def}}{=} & \{\!|e_1[x \mapsto v]|\!\}_{e_0[x \mapsto v]} \\
\langle d_1, \ldots, d_k \rangle [x \mapsto v] & \stackrel{\text{def}}{=} & \langle d_1[x \mapsto v], \ldots, d_k[x \mapsto v] \rangle \\
(\{\!|d|\!\}_e)[x \mapsto v] & \stackrel{\text{def}}{=} & \{\!|d[x \mapsto v]|\!\}_{e[x \mapsto v]} \\
(d \triangleleft X)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} d \triangleleft X & \text{if } x \in X \\ d[x \mapsto v] \triangleleft X & \text{if } x \notin X \end{cases}
\end{array}
$$

Table 3.5: Substitution; $e[x \mapsto v]$, $d[x \mapsto v]$, and $p[x \mapsto v]$.

3.1.3 Well-formedness

A well-formedness condition is an (often simple) analysis deciding whether a specific analysis component, Θ, is an acceptable estimate for specific phrase, $\mathbf{p}$, written with the judgement $\Theta \vdash \mathbf{p}$. Hence, if we assume that components belong to the domains $\Theta \in \mathsf{A}$ and $\mathbf{p} \in \mathsf{P}$, we may interpret the judgement relation as a function and type it

$$\vdash\colon\ (\mathsf{A} \times \mathsf{P}) \to \{\mathsf{true}, \mathsf{false}\}$$

Sometimes we may use subscripts or superscripts on $\vdash$ to denote static components.[3]

Usually the relation $\vdash$ is defined by a finite number of inference rules specifying which pairs that are well-formed, i.e. mapped to true, and then components

[3]Formally, this is equivalent to make the relation higher-order, where the sub-scripts and/or superscripts are used for instantiating the relation.

not satisfying the rules are mapped to false. We say that the definition is *compositional* whenever the validity of the relation between complex expressions may always be determined based on the validity of the constituent expressions and the rules used to combine them. Conversely, relations that are not compositional are usually referred to as *abstract*.

Languages are typically defined in a general and non-restrictive manner, and specific unwanted behaviour is instead avoided with the well-formedness condition. This allows a, possibly, more generic language design, but it also requires a strict enforcement of the well-formedness condition. In other words, we are only interested in components that definitely satisfy the well-formedness condition, and all other components should be rejected; also, in the case of an abstract well-formedness condition, components leading to an infinite derivation tree. This is called an *inductive interpretation* of the definition.

The syntax and semantics given in the previous sections are examples of such non-restrictive language definitions. There are no guarantees that names and variables are bound before they are used and whenever v as $p : \theta$ then the semantics allows $dom(\theta) \subseteq \mathsf{bv}(p)$ (as opposed to the equality that our intuition dictates) and that the inclusion might be strict. In fact, a more rigorous definition of $\mathsf{bv}(d \triangleleft X)$, based on the semantics alone, would be $\mathsf{fv}(d) \cap X$, but that is much less intuitive, and most likely not the original intention of the protocol designer. Therefore, we need a well-formedness condition that guarantees that the language meets the expectations.

Another consideration is that the semantics does not enforce the usual assumption of perfect cryptography; that is, the assumption that a cipher text cannot be decrypted without knowledge of the decryption key and that the encryption or decryption key can never be derived from a cipher text. A pattern like $\{\!|y|\!\}_x \triangleleft \{x, y\}$ obviously violates this assumption, as it allows for deriving both the key, x, and the plain text, y, from a cipher text without any prior knowledge. This flexibility supports scenarios where certain keys or parts of the plain text are guessed, such as when modelling guessing attacks or similar vulnerabilities. Still, perfect cryptography is the usual assumption in protocol analysis, and, thus, we shall formulate the well-formedness condition such that perfect cryptography is guaranteed by it, hereby allowing the protocol designer or analyst to choose if the criterion should apply in each case.

The well-formedness condition will assume the existence of the sets of previously bound names N and previously bound variables X, scoped around the cryptographic component. It follows, that for an encoding, we shall simply require all names and variables used within the encoding to belong to the sets N and X, respectively. The resulting well-formedness condition for encodings is written as the judgement $\langle \mathsf{N}, \mathsf{X} \rangle \vdash e$ in Table 3.6.

$$\text{(WE)}\quad \frac{\mathsf{fn}(e) \subseteq \mathsf{N} \quad \mathsf{fv}(e) \subseteq \mathsf{X}}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e}$$

Table 3.6: Well-formedness of encodings; $\langle \mathsf{N}, \mathsf{X} \rangle \vdash e$.

The requirement for patterns is more subtle, as we must ensure that the patterns enforce the rules of perfect cryptography. Our approach to this challenge is based on the following observation:

Observation 3.4 *A pattern, $d \triangleleft X$, satisfies the rules of perfect cryptography, whenever there exists an ordered sequence, $x_1, x_2, \ldots$, of the variables in X, such that x_1 may be derived from d according to the rules of perfect cryptography and without knowledge of the binding of the remaining variables; deriving x_2 requires at most knowledge of the binding of x_1; etc.*

To exemplify this observation, consider the pattern $\langle \{\!|x|\!\}_y, \{\!|y|\!\}_x \rangle \triangleleft \{x, y\}$; here a cyclic dependency exists as neither x may be derived without knowledge of y, nor y without knowledge of x. However, if we modify the pattern to $\langle \{\!|x|\!\}_y, \{\!|y|\!\}_z, z \rangle \triangleleft \{x, y, z\}$ then suddenly a derivation sequence exists, namely the sequence z, y, x. Thus, the well-formedness condition must accept any pattern for which the variables may be derived in an orderly fashion, and disallow all other patterns.

That a variable, x, is derivable from a decoding d, according to the rules of perfect cryptography and assuming the knowledge $\langle \mathsf{N}, \mathsf{X} \rangle$, is captured by the auxiliary judgement $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x d$ as defined in Table 3.7. Obviously, the variable x is derived legally from the decoding x. If the decoding is composite, a recursive search is initiated; in a tuple at least one of the sub-decodings must provide knowledge of x, and a cryptographic construction should only allow x to be derived from the contents, if the key (i.e. the encoding e) consists of purely known names and variables. This is sufficient as knowledge of all names in a value is enough to recreate the value, given that everybody is assumed access to the same cryptographic operations. Note that there are, of course, no rules allowing new variable bindings to be derived from encryptions.

The auxiliary judgement allows for specifying the well-formedness condition for patterns, $\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft X$, as defined by the two rules (WDef) and (WEmp) in Table 3.7. If X is not empty the rule (WDef) applies, where an element $x \in X$, i.e. the next variable to be derived, is required not to be previously bound, $x \notin \mathsf{X}$, and legally derivable from the decoding given the current knowledge, $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x d$. If this is the case, the variable x can safely be added to the current

set of known variables, X, and removed from the set of variables yet to be derived, X. The inductive nature of the definition ensures that at some point X will either consist exclusively of mutual dependent variables, or the set will be empty. In the case of the latter, the rule (WEMP) applies, which establishes that d is now a well-formed decoding including only known variables, $\mathsf{fv}(e) \subseteq \mathsf{X}$, and names, $\mathsf{fn}(e) \subseteq \mathsf{N}$.

Remark 3.5 *The constraint $x \notin \mathsf{X}$ in* (WDEF) *ensures that variables have a uniquely binding occurrence, such that patterns like $\{x\}_x \triangleleft \{x\}$ are never accepted; not even when x is also previously bound. If multiple definitions of variables within a pattern is allowed, then patterns, such as the above-mentioned, becomes ambiguous; i.e. it is unclear which x that the protocol designer is referring to. Note that this does not force the calculus employing the cryptographic components to disallow redefinitions of variables altogether, but only that variable definitions should be unique within each pattern.*

$$\text{(WDEF)}\quad \frac{x \notin \mathsf{X} \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x d \quad \langle \mathsf{N}, \mathsf{X} \cup \{x\} \rangle \vdash d \triangleleft X}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft X \cup \{x\}}$$

$$\text{(WEMP)}\quad \frac{\mathsf{fn}(d) \subseteq \mathsf{N} \quad \mathsf{fv}(d) \subseteq \mathsf{X}}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft \emptyset}$$

$$\text{(WH1)}\quad \frac{\bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x e_i}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x \langle e_1, \ldots, e_k \rangle}$$

$$\text{(WH2)}\quad \frac{x = y}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x y} \qquad \text{(WH3)}\quad \frac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x d}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x \{\!|d|\!\}_e}$$

$$\text{(WH4)}\quad \frac{\bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x d_i}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x \langle d_1, \ldots, d_k \rangle}$$

Table 3.7: Well-formedness of patterns; $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$.

To fully understand the need for, and the workings of, the well-formedness condition, consider the following example.

Example 3.6 *The following three patterns are all entirely legal:*

(1) $\{\!|x|\!\}_k \triangleleft \{x\}$ where $k \in \mathsf{N}$
(2) $\{\!|y|\!\}_x \triangleleft \{y\}$ where $x \in \mathsf{X}$
(3) $\langle \{\!|x|\!\}_y, \{\!|y|\!\}_z, z \rangle \triangleleft \{x, y, z\}$

In (1) *the new variable x is derived through decryption with k, an already known name. Similarly, in* (2)*, y is derived using the prior knowledge of x to decrypt*

the encryption. In (3) z *may be derived from the third element of the tuple, then the second part of the tuple can be decrypted using* z *hereby deriving* y*, and finally from the first part of the tuple* x *can be found using* y*.*

All of the above patterns follow the cryptographic rules and are correctly accepted by our well-formedness condition. However, slightly altering these patterns, change them into patterns with unwanted properties – all correctly disqualified by the well-formedness condition.

$$
\begin{array}{ll}
(1') & \{\!|k|\!\}_x \lhd \{x\} \quad \text{where } k \in \mathsf{N} \\
(1'') & \{\!|k|\!\}_k \lhd \{x\} \quad \text{where } k \in \mathsf{N} \\
(2') & \{\!|y|\!\}_x \lhd \{x, y\} \\
(3') & \langle \{\!|x|\!\}_y, \{\!|y|\!\}_x \rangle \lhd \{x, y\}
\end{array}
$$

The pattern (1′) *is illegal as either* $x \in \mathsf{X}$ *and the pattern renames* x*, which is not allowed, otherwise* x *is not in scope, meaning that* x *is derived from a key, thus violating perfect cryptography and not satisfying* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x \{\!|k|\!\}_x$*. In pattern* (1″) x *is simply never bound, and in* (2′) x *is again only derived from a key. Finally, in* (3′)*,* x *cannot be derived without prior knowledge of* y *and* y *cannot be derived without knowledge of* x*, resulting in that neither* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x \langle \{\!|x|\!\}_y, \{\!|y|\!\}_x \rangle$ *nor* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_y \langle \{\!|x|\!\}_y, \{\!|y|\!\}_x \rangle$ *can be satisfied when* $x, y \notin \mathsf{X}$*.*

3.2 Revising the Model

The modelling of cryptographic components presented in §3.1, hereafter referred to as the basic model, captures our intuition of cryptography in a succinct manner. It does, however, only support symmetric cryptography and most modern systems also rely on other cryptographic operations such as asymmetric encryption, digital signatures, hashing, etc. We now show how the basic model may be extended with additional cryptographic operations. Specifically, we will add support for asymmetric encryption and digital signatures, as this extension allows for modelling most of the classical cryptographic protocols, and it introduces some interesting new aspects that the model must cater for.

We adopt the usual abstraction for asymmetric cryptography and rely on key pairs, $\langle k^+, k^- \rangle$, consisting of a public key, k^+, and a private key, k^-, each identified by their superscripts. The central idea is then that any party, who knows the public available half of the key pair, may encrypt messages. Decryption, on the other hand, requires knowledge of the corresponding private key, which is usually reserved for the owner of the key pair. We shall write $[\![e]\!]_{k^+}$ for expressing the cipher text result from encrypting a message e using the public key k^+. An assumption of perfect cryptography would then dictate that obtaining e from $[\![e]\!]_{k^+}$ would require knowledge of k^-.

Typically asymmetric cryptography is also used for modelling digital signatures, simply by reversing the order of the keys. An asymmetrical encryption of a message e using a private key k^-, written $\lfloor e \rfloor_{k^-}$, represents therefore e digitally signed by the owner of the key pair $\langle k^+, k^- \rangle$, and may be verified by everybody that knows the public key k^+. [4] In the remainder of this section we shall show how the basic model may be adapted to support these operations.

3.2.1 Syntax

The introduction of asymmetric cryptography requires us to reevaluate the model from §3.1. Recall that in the basic model we distinguish between matching incoming values against values composed from the current knowledge (i.e. encodings), or decomposing the incoming values using the current knowledge and then match on the constituent parts. For symmetric encryption these two approaches will have the same result because of the symmetry, and the difference is only interesting for analyses that takes computation time into account. But when we add asymmetric components to the model, the approaches are no longer equivalent.

Assume, for example, that a principal A wants to encrypt a message, m, with principal B's public key, Kb^+, and send the cipher text to B. Now, if a future confirmation message from B includes the cipher text again, A will not be able to decrypt it and confirm the content, as A does not know the corresponding decryption key Kb^- (only B knows this). However, A can easily recreate the cipher text and bitwise compare the result to the received message.

It follows from the discussion above that the model requires both asymmetric decryption and encryption in patterns, similarly to the basic model. Conversely, it is easily seen that asymmetric cryptography also opens up scenarios where a principal can decompose a value but cannot recreate this value afterwards. Consider, for example, digital signatures: a principal who receives a signed value may be able to learn the signed content (through decryption with the signer's public key), but cannot reproduce the signed value afterwards, because it does not know signer's private key. Again, this introduces complications when modelling protocols, where a principal receives a value, verifies that it is signed by the expected principal, and forwards it to a third party.[5] To cater for this, we shall also extend our model with a capability for enforcing restrictions on

[4] This modelling is arguably slightly misleading, because digital signatures in general add no confidentiality to messages. However, as decryption only requires knowledge of the key k^+, which is assumed publicly available, the model is still sound.

[5] An example of this behaviour would be the administrator role in many voting protocols (e.g. [14, 40, 50]).

the values a variable may be mapped to. With such an extension, the example mentioned above could be modelled by letting the principal bind the incoming signed value to a variable x, if and only if it was signed by the expected principal.

The resulting revised syntax for the cryptographic components is given in Table 3.8. Names are now annotated with a tag; the tags $+$ and $-$ are used for public and private keys in a key pair, respectively, and ϵ represents all other names, equivalent to the basic model. Asymmetric encryption is modelled by the operator $\lfloor\cdot\rfloor_\cdot$, and for decodings we add the operator $\lfloor\!\lfloor\cdot\rfloor\!\rfloor_\cdot$ for public key decryption.

Instead of the set X, representing variables intended to be bound in a pattern in the basic model, we now associate each variable intended with a decoding that any variable binding must match; i.e. we write $x \mapsto d$ for defining the variable x which is allowed to be mapped to any value v that matches the decoding d.

Remark 3.7 *Usually, encryption can be replaced by an equivalent decryption in decodings. Thus, to avoid confusion, we shall only use decryption in decodings and encryption in encodings in the remainder of this dissertation. However, it is imperative to know the difference, and that in some, albeit rare, cases encryption is needed in a decoding.*

e	$::=$	n^τ	Name, $n^\tau \in \mathcal{N}, \tau \in \{+,-,\epsilon\}$
	$\mid$	x	Variable, $x \in \mathcal{X}$
	$\mid$	$\langle e_1, \ldots, e_k \rangle$	Tuple
	$\mid$	$\{e_1\}_{e_0}$	Symmetric encryption
	$\mid$	$\lfloor e_1 \rfloor_{e_0}$	Asymmetric encryption
d	$::=$	$*$	Wildcard
	$\mid$	e	Comparison
	$\mid$	$\langle d_1, \ldots, d_k \rangle$	Decompose tuple
	$\mid$	$\{\!\{d\}\!\}_e$	Symmetric decryption
	$\mid$	$\lfloor\!\lfloor d \rfloor\!\rfloor_e$	Asymmetric decryption
p	$::=$	$d \triangleleft [x_1 \mapsto d_1, \cdots, x_k \mapsto d_k]$	Pattern, $\{x_1, \ldots, x_k\} \subseteq \mathcal{X}$

Table 3.8: Revised syntax.

As restrictions on variables are also enforced through decodings, we extend decodings with a wildcard, $*$, that simply matches any value. Hence, variables defined with the decoding $*$ may be bound unconditionally, similarly to the variables of the basic model, and we shall refer to such variables as *unrestricted variables* and all other variables as *restricted variables*.

Convention 3.8 *Each variable must have a corresponding decoding, and thus the set of variables in the pattern may be regarded as a partial map, denoted* Γ*, mapping each defined variable to its decoding.*

Notice that decodings allow for both recursive variable definitions as well as binding variables within the value binding of other variables. Recursive variable definitions, such as in the pattern $x \triangleleft [x \mapsto \langle x, x \rangle]$, would in this pattern matching context never match any values (as values are always of finite size). Thus such decodings should be prohibited by a well-formedness condition. Defining variables based on other variables, however, provides a useful tool when describing protocols; e.g. the pattern $x \triangleleft [x \mapsto [\![y]\!]_{k^+}, y \mapsto *]$ allows for both binding a signed value and the value itself to variables in one go.

As in the basic model, we shall rely on the usual definitions of free names and variables; for completeness these are given in Table 3.9 and Table 3.10. We here use the operator $rg(\cdot)$ to denote the range of a function or map. Additionally, we may refer to the bound variables in a pattern, $\mathsf{bv}(d \triangleleft \Gamma)$, which in the extended design is simply $dom(\Gamma)$.

$$
\begin{array}{rcl}
\mathsf{fn}(n^\tau) & \stackrel{\text{def}}{=} & \{n^\tau\} \\
\mathsf{fn}(x) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fn}(\langle e_1, \ldots, e_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fn}(e_1) \cup \cdots \cup \mathsf{fn}(e_k) \\
\mathsf{fn}(\{\!|e_1|\!\}_{e_0}) & \stackrel{\text{def}}{=} & \mathsf{fn}(e_0) \cup \mathsf{fn}(e_1) \\
\mathsf{fn}([\![e_1]\!]_{e_0}) & \stackrel{\text{def}}{=} & \mathsf{fn}(e_0) \cup \mathsf{fn}(e_1) \\
\mathsf{fn}(*) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fn}(\langle d_1, \ldots, d_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fn}(d_1) \cup \cdots \cup \mathsf{fn}(d_k) \\
\mathsf{fn}(\{\!|d|\!\}_e) & \stackrel{\text{def}}{=} & \mathsf{fn}(e) \cup \mathsf{fn}(d) \\
\mathsf{fn}([\![d]\!]_e) & \stackrel{\text{def}}{=} & \mathsf{fn}(e) \cup \mathsf{fn}(d) \\
\mathsf{fn}(d \triangleleft \Gamma) & \stackrel{\text{def}}{=} & \mathsf{fn}(d) \cup \left(\bigcup_{d' \in rg(\Gamma)} \mathsf{fn}(d') \right)
\end{array}
$$

Table 3.9: Revised free names; fn.

Notation 3.9 *We may, when convenient, omit the* ϵ *on simple names. Likewise, we may simply write* x *instead of* $x \mapsto *$ *for unrestricted variables.*

$$
\begin{array}{rcl}
\mathsf{fv}(n^{\tau}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(x) & \stackrel{\text{def}}{=} & \{x\} \\
\mathsf{fv}(\langle e_1, \ldots, e_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_1) \cup \cdots \cup \mathsf{fv}(e_k) \\
\mathsf{fv}(\{e_1\}_{e_0}) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_0) \cup \mathsf{fv}(e_1) \\
\mathsf{fv}([e_1]_{e_0}) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_0) \cup \mathsf{fv}(e_1) \\
\mathsf{fv}(*) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(\langle d_1, \ldots, d_k \rangle) & \stackrel{\text{def}}{=} & \mathsf{fv}(d_1) \cup \cdots \cup \mathsf{fv}(d_k) \\
\mathsf{fv}(\{\!|d|\!\}_e) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup \mathsf{fv}(d) \\
\mathsf{fv}([\![d]\!]_e) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup \mathsf{fv}(d) \\
\mathsf{fv}(d \triangleleft \Gamma) & \stackrel{\text{def}}{=} & \left(\mathsf{fv}(d) \cup \left(\bigcup_{d' \in rg(\Gamma)} \mathsf{fv}(d')\right)\right) \setminus dom(\Gamma)
\end{array}
$$

Table 3.10: Revised free variables; fv.

3.2.2 Semantics

The judgement for the semantics still takes the form $v \ \mathsf{as} \ p : \theta$ and is defined in Table 3.11. It is closely related to the semantics for the basic model, but requires a few more rules. The rule (RPEnc) is used whenever the pattern is a comparison between an incoming value and an asymmetrically encrypted value. The rules (RPDec) and (RSign) are used for asymmetric decryption and validation of digital signatures, respectively, and are similar to symmetric decryption, except we ensure that the keys are tagged as required and that they belong to a key pair. When matching a value v against a variable x the rule (RBind) applies, requiring the value v to match the decoding of x and update the resulting substitution of this matching with the binding of x to v.

(RName)	n^{τ} as $n^{\tau} \lhd \Gamma : [\,]$	
(RBind)	$\dfrac{x \in dom(\Gamma) \qquad v \text{ as } \Gamma(x) \lhd \Gamma : \theta}{v \text{ as } x \lhd \Gamma : \theta[x \mapsto v]}$	
(RETup)	$\dfrac{\bigwedge_{i=1}^{k} v_i \text{ as } e_i \lhd \Gamma : \theta_i}{\langle v_1, \ldots, v_k \rangle \text{ as } \langle e_1, \ldots, e_k \rangle \lhd \Gamma : \theta_1 \cup \cdots \cup \theta_k}$	if $\forall i, j : \theta_i =_s \theta_j$
(RSEnc)	$\dfrac{v_0 \text{ as } e_0 \lhd \Gamma : \theta_0 \qquad v_1 \text{ as } e_1 \lhd \Gamma : \theta_1}{\{v_1\}_{v_0} \text{ as } \{e_1\}_{e_0} \lhd \Gamma : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$
(RPEnc)	$\dfrac{v_0 \text{ as } e_0 \lhd \Gamma : \theta_0 \qquad v_1 \text{ as } e_1 \lhd \Gamma : \theta_1}{[v_1]_{v_0} \text{ as } [e_1]_{e_0} \lhd \Gamma : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$
(RWild)	v as $* \lhd \Gamma : [\,]$	
(RDTup)	$\dfrac{\bigwedge_{i=1}^{k} v_i \text{ as } d_i \lhd \Gamma : \theta_i}{\langle v_1, \ldots, v_k \rangle \text{ as } \langle d_1, \ldots, d_k \rangle \lhd \Gamma : \theta_1 \cup \cdots \cup \theta_k}$	if $\forall i, j : \theta_i =_s \theta_j$
(RSDec)	$\dfrac{v_0 \text{ as } e \lhd \Gamma : \theta_0 \qquad v_1 \text{ as } d \lhd \Gamma : \theta_1}{\{v_1\}_{v_0} \text{ as } \{\!|d|\!\}_e \lhd \Gamma : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$
(RPDec)	$\dfrac{n^{-} \text{ as } e \lhd \Gamma : \theta_0 \qquad v \text{ as } d \lhd \Gamma : \theta_1}{[v]_{n^{+}} \text{ as } [\![d]\!]_e \lhd \Gamma : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$
(RSign)	$\dfrac{n^{+} \text{ as } e \lhd \Gamma : \theta_0 \qquad v \text{ as } d \lhd \Gamma : \theta_1}{[v]_{n^{-}} \text{ as } [\![d]\!]_e \lhd \Gamma : \theta_0 \cup \theta_1}$	if $\theta_0 =_s \theta_1$

Table 3.11: Revised pattern matching; v as $p : \theta$.

As in the basic model, we also need a definition of substitution on encodings and patterns. These are defined straightforwardly and listed in Table 3.12.

3.2.3 Well-formedness

The well-formedness condition must also cater for the language extension with asymmetric cryptography. The well-formedness condition for encodings is unchanged and listed in Table 3.13.

Well-formedness for patterns, listed in Table 3.14, still requires an auxiliary judgement for determining whether a variable may be legally derived from a decoding, d, given the current known names N and previously bound variables X. This is captured by the judgement $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^{\Gamma} d$ that now, in addition to the decoding and the current knowledge, also requires the map Γ between variables and their corresponding decoding. The auxiliary judgement is analogous to the auxiliary judgement in the basic model, but it requires an additional rule,

$$
\begin{array}{rcl}
n^{\tau}[x \mapsto v] & \stackrel{\text{def}}{=} & n^{\tau} \\
y[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} v & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
\langle e_1, \ldots, e_k \rangle [x \mapsto v] & \stackrel{\text{def}}{=} & \langle e_1[x \mapsto v], \ldots, e_k[x \mapsto v] \rangle \\
(\{\!|e_1|\!\}_{e_0})[x \mapsto v] & \stackrel{\text{def}}{=} & \{\!|e_1[x \mapsto v]|\!\}_{e_0[x \mapsto v]} \\
([\![e_1]\!]_{e_0})[x \mapsto v] & \stackrel{\text{def}}{=} & [\![e_1[x \mapsto v]]\!]_{e_0[x \mapsto v]} \\
*[x \mapsto v] & \stackrel{\text{def}}{=} & * \\
\langle d_1, \ldots, d_k \rangle [x \mapsto v] & \stackrel{\text{def}}{=} & \langle d_1[x \mapsto v], \ldots, d_k[x \mapsto v] \rangle \\
(\{\!|d|\!\}_e)[x \mapsto v] & \stackrel{\text{def}}{=} & \{\!|d[x \mapsto v]|\!\}_{e[x \mapsto v]} \\
([\![d]\!]_e)[x \mapsto v] & \stackrel{\text{def}}{=} & [\![d[x \mapsto v]]\!]_{e[x \mapsto v]} \\
(d \triangleleft \Gamma)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} d \triangleleft \Gamma & \text{if } x \in dom(\Gamma) \\ d[x \mapsto v] \triangleleft \{y \mapsto d'[x \mapsto v] \mid (y \mapsto d') \in \Gamma\} & \text{if } x \notin dom(\Gamma) \end{cases}
\end{array}
$$

Table 3.12: Revised substitution; $e[x \mapsto v]$, $d[x \mapsto v]$, and $p[x \mapsto v]$.

(RWH6), for the asymmetric decryption construct, as well as the rule (RWH5), which allows variables to be derived from decodings of other variables. The latter is necessary whenever the binding of one variable is a sub-value of another variable binding; e.g. in the pattern $x \triangleleft [x \mapsto [\![y]\!]_{k^+}, y \mapsto *]$ the variable y is bound within the decoding of x. Notice that new variable bindings are, of course, still not allowed to be found within encryptions, and thus there is no rule for the asymmetric encryption construct.

$$
\text{(RWT)} \quad \frac{\mathsf{fn}(e) \subseteq \mathsf{N} \quad \mathsf{fv}(e) \subseteq \mathsf{X}}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e}
$$

Table 3.13: Revised well-formedness of encodings; $\langle \mathsf{N}, \mathsf{X} \rangle \vdash e$.

The auxiliary judgement allows for defining the well-formedness condition for patterns and closely follows that of the basic model. We still require the variables to be derivable in an orderly fashion, as well as the decodings for each variable to include at most already known names and variables. The latter is a reasonable requirement, as if it was not satisfied and, for example, there existed two restrictive definitions $(x_i \mapsto d_i), (x_j \mapsto d_j) \in \Gamma$ such that $x_i \in \mathsf{fv}(d_j)$ and $x_j \in \mathsf{fv}(d_i)$, then either these variables would never be bound in a matching, or the pattern would only match values of infinite depth (where the semantics would go into an infinite loop). As the former is unacceptable behaviour and the latter could never happen, such patterns should be prohibited.

$$\text{(RWDef)}\quad \frac{x \notin \mathsf{X} \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma d \quad \mathsf{fn}(d') \subseteq \mathsf{N} \quad \mathsf{fv}(d') \subseteq \mathsf{X} \quad \langle \mathsf{N}, \mathsf{X} \cup \{x\} \rangle \vdash d \triangleleft \Gamma}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft \Gamma[x \mapsto d']}$$

$$\text{(RWEmp)}\quad \frac{\mathsf{fn}(d) \subseteq \mathsf{N} \quad \mathsf{fv}(d) \subseteq \mathsf{X}}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft [\,]}$$

$$\text{(RWH1)}\quad \frac{\bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma e_i}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma \langle e_1, \ldots, e_k \rangle}$$

$$\text{(RWH2)}\quad \frac{x = y}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma y} \qquad \text{(RWH3)}\quad \frac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma d}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma \{\!|d|\!\}_e}$$

$$\text{(RWH4)}\quad \frac{\bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma d_i}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma \langle d_1, \ldots, d_k \rangle}$$

$$\text{(RWH5)}\quad \frac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma \Gamma(y)}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma y} \qquad \text{(RWH6)}\quad \frac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma d}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma [\![d]\!]_e}$$

Table 3.14: Revised well-formedness of patterns; $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$.

Remark 3.10 *It is important to note that the definition does not explain how the order of the derivation sequence is found, but only that if it exists, then well-formedness is satisfied. In fact, the rule* (RWH5)*, for the auxiliary judgement, renders the definition of well-formedness coinductive, and thus it must be determined through fix-point iteration.*

We showed in Example 3.6 how perfect cryptography is enforced through well-formedness in the basic model. These examples naturally also apply to the extended model, but the addition of asymmetric cryptography introduces additional, interesting ways to circumvent perfect cryptography, and the following example shows how the revised well-formedness condition, presented above, also captures these.

Example 3.11 *The following patterns are all legal:*

(1) $y \triangleleft [x \mapsto *, y \mapsto [\![x]\!]_{m^-}]$ where $m^- \in \mathsf{N}$
(2) $\langle [\![x]\!]_x, y \rangle \triangleleft [x \mapsto *, y \mapsto x]$
(3) $\langle x, y \rangle \triangleleft [x \mapsto [\![y]\!]_y, y \mapsto *]$

In both (1) *and* (2) x *may be found through the decoding of* y*, whereafter* y *may be found directly. In* (3) *we must first find* y*, whereafter finding* x *is straightforward. Notice, that both* (2) *and* (3) *will only match values of the form* $\langle [\![m^+]\!]_{m^-}, m^+ \rangle$ *or* $\langle [\![m^-]\!]_{m^+}, m^- \rangle$*.*

Again we can render these patterns illegal by modifying them slightly. The

following patterns are all correctly disqualified by the well-formedness condition:

$$
\begin{array}{ll}
(1') & y \lhd [x \mapsto *, y \mapsto [\![x]\!]_{m^-}] \quad \text{where } m^- \notin \mathsf{N} \\
(2') & \langle [\![x]\!]_y, x \rangle \lhd [x \mapsto *, y \mapsto x] \\
(3') & \langle x, y \rangle \lhd [x \mapsto [\![y]\!]_y, y \mapsto x]
\end{array}
$$

The pattern $(1')$ *is illegal as it includes an unknown name,* $m^- \notin \mathsf{N}$. *In* $(2')$ *the variable* y *is moved, so that it is read through an encryption key instead. This violates well-formedness, as although* x *may easily be found, the variable* y *can never be derived from the pattern. This illustrates well the intuition behind patterns and decodings; the decoding of a variable is used once the possible value binding of the variable is established, to ensure that this value matches the intention, and not to establish the binding itself. The pattern* $(3')$ *simply gives circular definitions for* x *and* y*, and thus neither variable can be derived first, as their decodings are not well-formed without knowledge of the other variable.*

3.3 Properties of the Components

Whenever new concepts are being introduced, we are usually interested in determining whether they adhere to the usual rules and satisfy the expected properties. In this section, we shall show that the models for cryptographic components satisfy the most important properties.

Instead of giving analogous proofs for the basic model and the revised model, we draw attention to the fact that the basic design is included in the revised model. Any encoding, e, in the basic model, is also legal in the revised model, and any pattern $e \lhd X$ can be converted into an equivalent pattern in the revised model; i.e. $e \lhd \Gamma$ where $\Gamma = \{x \mapsto * \mid x \in X\}$. For encodings and patterns, committed to such a conversion, we see that the semantics of the two models are also equivalent, since the rules (RPEnc), (RPDec), and (RSign) would never be applied. This same is the case for well-formedness, and hence it is sufficient to prove that the properties of interest hold for the revised model, whereafter, directly and by construction, we may conclude that they also hold for the basic model.

Some of the proofs rely heavily upon knowledge of set and order theory, and the reader may refer to Appendix A for an introduction to the applied techniques.

3.3.1 Semantics

We are interested in two specific properties of the component semantics, namely *termination*, i.e. it is always computable whether a specific value and pattern match, and, in the case they match, the substitution the matching gives rise to; and *determinism*, i.e. the resulting substitution of a matching is always unique. These are shown below.

Proposition 3.12 (Termination) *There exists a terminating function which for all v and p returns a θ such that v as $p : \theta$, if one such exists, and otherwise terminates unsuccessfully.*

Proof The result is trivial as v as $p : \theta$ is compositional. Thus, defining the function analogously promises that the matching of any finite elements will be decided in a finite number of recursive calls. □

Proposition 3.13 (Determinism) *For all v and p there exists at most one θ such that v as $p : \theta$.*

Proof This is shown by contradiction. Assume θ and θ' such that $\theta \neq \theta'$, v as $p : \theta$, and v as $p : \theta'$. The result then follows by induction on the structure of the inference tree establishing v as $p : \theta$ where each case results in a contradiction. □

3.3.2 Well-formedness

We shall also show some properties of the well-formedness conditions. In particular, the properties we are looking for are:

- *Well-definedness*; that the definition is unambiguous, i.e. under assumption of sets N and X, then an encoding or a pattern is always either well-formed or not.
- *Termination*; that well-formedness is a decidable relation.
- *Type safety*; if a value is well-formed under assumption of the known set of names N, then so is any variable binding that it may give rise to through pattern matching.

- *Perfect cryptography guarantee*; well-formed components will never violate the perfect cryptography assumption.

These properties are shown below.

Proposition 3.14 (Well-definedness) *The relations* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash e$ *and* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$ *are well-defined.*

Proof The proof amounts to show that the judgement relations constitute total functions. These relations are both compositional and the result follows therefore by simple straightforward structural induction, although the proof for $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$ relies on that the auxiliary relation, $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$, is also well-defined.

Unfortunately, the relation $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$ is not compositional, because of the rule (RWH5), so instead we must show well-definedness by showing the existence of a fixed point; more precisely, the existence of a least fixed point as we require an inductive interpretation of the rules. Basing an analysis functional on the clauses then gives us

$$
\begin{aligned}
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, \langle e_1, \cdots, e_k \rangle) &\stackrel{\text{def}}{=} \bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} e_i \\
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, y) &\stackrel{\text{def}}{=} x = y \\
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, \{\!|d|\!\}_e) &\stackrel{\text{def}}{=} \langle \mathsf{N}, \mathsf{X} \rangle \vdash e \wedge \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d \\
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, \langle d_1, \cdots, d_k \rangle) &\stackrel{\text{def}}{=} \bigvee_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d_i \\
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, y) &\stackrel{\text{def}}{=} \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} \Gamma(y) \\
\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, [\![d]\!]_e) &\stackrel{\text{def}}{=} \langle \mathsf{N}, \mathsf{X} \rangle \vdash e \wedge \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d
\end{aligned}
$$

and where $\mathcal{F}[\vdash^{\Gamma}_{x}]$ returns false in all other cases. It follows that $\mathcal{F}[\vdash^{\Gamma}_{x}](\langle \mathsf{N}, \mathsf{X} \rangle, d)$ is true if and only if $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$ is. Clearly $\mathcal{F}[\vdash^{\Gamma}_{x}]$ is monotonic on a complete lattice (with the obvious choice of order), and, thus, by Tarski's fixed point theorem, the functional possesses a lattice of fixed points; in particular a least fixed point. □

Proposition 3.15 (Termination) *The functions* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash e$ *and* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$ *are terminating.*

Proof We must prove that the relations represent terminating functions. As in the proof of Proposition 3.14, this result is straightforward as both functions are compositional and thus recursively defined on a recursive structure, but, again, the proof relies on that the auxiliary function, $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$, is also terminating.

Hence, the proof amounts to defining a terminating function $f_{\Gamma,x}$, equivalent to an inductive interpretation of the rules defining $\langle \mathsf{N}, \mathsf{X} \rangle \vdash_x^\Gamma d$; i.e. the least fixed point of the functional $\mathcal{F}[\vdash_x^\Gamma]$ as defined in the proof for Proposition 3.14.

Recall, first, that an inductive interpretation implies that tuples $\langle \langle \mathsf{N}, \mathsf{X} \rangle, d \rangle$, which cannot be decided within a finite derivation tree, should be mapped to false. This implies that we need only traverse through the decoding of any variable at most once in each path of the derivation tree, as repeated traversals only lead to non-termination and not any additional information. In other words, we may obtain a well-founded recursion by using a simple memoisation technique as follows:

$$
\begin{aligned}
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, \langle e_1, \cdots, e_k \rangle) &\stackrel{\text{def}}{=} \bigvee_{i=1}^{k} f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, e_i) \\
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, y) &\stackrel{\text{def}}{=} x = y \\
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, \{\!|d|\!\}_e) &\stackrel{\text{def}}{=} \langle \mathsf{N}, \mathsf{X} \rangle \vdash e \wedge f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, d) \\
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, \langle d_1, \cdots, d_k \rangle) &\stackrel{\text{def}}{=} \bigvee_{i=1}^{k} f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, d_i) \\
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, y) &\stackrel{\text{def}}{=} \begin{cases} f^{\mathcal{S} \cup \{y\}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, \Gamma(y)) & \text{if } y \notin \mathcal{S} \\ \mathsf{false} & \text{if } y \in \mathcal{S} \end{cases} \\
f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, [\![d]\!]_e) &\stackrel{\text{def}}{=} \langle \mathsf{N}, \mathsf{X} \rangle \vdash e \wedge f^{\mathcal{S}}_{\Gamma,x}(\langle \mathsf{N}, \mathsf{X} \rangle, d)
\end{aligned}
$$

and where $f^{\mathcal{S}}_{\Gamma,x}$ returns false in all other cases. Defining $f_{\Gamma,x}$ as $f^{\emptyset}_{\Gamma,x}$ then obviously results in a terminating function, as any decoding, d, holds only a finite number of variables, $\mathsf{fv}(d)$. □

Proposition 3.16 (Type Safety) *If* $\langle \mathsf{N}, \emptyset \rangle \vdash v$ *and* v as $p : \theta$ *then* $\langle \mathsf{N}, \emptyset \rangle \vdash v'$ *for all* $v' \in rg(\theta)$.

Proof The result is found by structural induction in the definition of v as $p : \theta$ where it follows directly from the fact that $\langle \mathsf{N}, \emptyset \rangle \vdash v$ implies $\langle \mathsf{N}, \emptyset \rangle \vdash v'$ for any sub-value v' of v. □

Notice that type safety does *not* require the patterns to be well-formed, but only the values, as well-formed patterns merely enforce the assumption of perfect cryptography and that all variables are correctly mapped. This leads us to the last and less trivial property, namely that well-formed patterns enforce the assumption of perfect cryptography.

Perfect Cryptography. Before we can discuss whether a pattern adheres to the assumption of perfect cryptography, we must give a formal definition of the

concept. Naturally, patterns or encodings are malformed if they contain unbound variables or names, but this may trivially be caught by the well-formedness condition.

Fact 3.17 *If* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$ *then* $\mathsf{fn}(p) \subseteq \mathsf{N}$ *and* $\mathsf{fv}(p) \subseteq \mathsf{X}$.

Proof Follows by induction in the structure of the inference tree establishing $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$. □

And, by definition, an analogous result holds for encodings. In other words, unbound entities can be trivially avoided by equipping the surrounding language with a similar well-formedness condition, ensuring that all free occurrences in encodings and patterns are previously bound. Hence, the main concern is not for the unbound names and variables, but for the variables bound within the pattern matching; we want the well-formedness condition to ensure that a well-formed pattern can never be used for abusing perfect cryptography.

To show such an abstract result, we shall first introduce a principal I that plays according to the rules of perfect cryptography and will always attempt to maximise its knowledge. Formally, we define I's accumulated knowledge $I(\mathcal{K})$, assuming its initial knowledge $\mathcal{K}$, as listed in Table 3.15.

$$\text{(I1)}\quad \frac{v \in \mathcal{K}}{v \in I(\mathcal{K})}$$

$$\text{(I2)}\quad \frac{\langle v_1, \ldots, v_k \rangle \in I(\mathcal{K})}{v_1, \ldots, v_k \in I(\mathcal{K})} \qquad \text{(I3)}\quad \frac{v_1, \ldots, v_k \in I(\mathcal{K})}{\langle v_1, \ldots, v_k \rangle \in I(\mathcal{K})}$$

$$\text{(I4)}\quad \frac{v_0, v \in I(\mathcal{K})}{\{v\}_{v_0} \in I(\mathcal{K})} \qquad \text{(I5)}\quad \frac{\{v\}_{v_0} \in I(\mathcal{K}) \quad v_0 \in I(\mathcal{K})}{v \in I(\mathcal{K})}$$

$$\text{(I6)}\quad \frac{[v]_{n^+} \in I(\mathcal{K}) \quad n^- \in I(\mathcal{K})}{v \in I(\mathcal{K})} \qquad \text{(I7)}\quad \frac{[v]_{n^-} \in I(\mathcal{K}) \quad n^+ \in I(\mathcal{K})}{v \in I(\mathcal{K})}$$

$$\text{(I8)}\quad \frac{v_0, v \in I(\mathcal{K})}{[v]_{v_0} \in I(\mathcal{K})}$$

Table 3.15: The knowledge of I; $I(\mathcal{K})$.

Rule (I1) shows that I has some initial knowledge $\mathcal{K}$. It can then extend its knowledge by decomposing and composing tuples (rules (I2) and (I3)), and creating new encryptions and decrypting existing encryptions with known keys (rules (I4-I8)). Given these ingredients we now define that a pattern p guarantees

perfect cryptography if and only if allowing I to use pattern matching of any value $v \in I(\mathcal{K})$ against p would never produce new variable bindings, not already in $I(\mathcal{K})$. More formally we have:

Definition 3.18 *The pattern p guarantees perfect cryptography if and only if for all v and θ where v as $p : \theta$ then $\{v\} \cup \mathsf{fn}(p) \subseteq I(\mathcal{K})$ implies $rg(\theta) \subseteq I(\mathcal{K})$.*

Intuitively, if matching against p could violate this requirement, $I(\mathcal{K})$ would gain knowledge that he did not previously have – as I fully utilises perfect cryptography, this would imply breaking perfect cryptography. This provides us with a concrete requirement for patterns to guarantee perfect cryptography, which we now must prove is ensured by the well-formedness condition.

Facts 3.19
(1) *If v as $p : \theta$ then $dom(\theta) \subseteq \mathsf{bv}(p)$*
(2) *If v as $e \triangleleft \Gamma : \theta$ then $v = e\theta$*
(3) *If $\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft \Gamma'$ and $\Gamma' \subseteq \Gamma$ then $\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft \Gamma$*

Proof Each result is trivial and follow by straightforward induction. □

We then ease presentation of the main result by first giving an auxiliary result showing that $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$ is only satisfied if x can be derived from d without violating perfect cryptography.

Lemma 3.20 *If v as $d \triangleleft \Gamma : \theta$ and $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d$ where $\{v\} \cup \mathsf{fn}(d) \cup \{\theta(y) \mid y \in \mathsf{X} \cap dom(\theta)\} \subseteq I(\mathcal{K})$ then $\theta(x) \in I(\mathcal{K})$.*

Proof The proof proceeds by induction on the structure of d, where we show the interesting cases:

Case y is trivially as $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} y$ implies $x = y$ by (RWH2), and thus the result follows by the assumption $v \in I(\mathcal{K})$ and rule (RBIND).

Case $\{\!|d'|\!\}_e$. According to the semantics v must then be of the form $\{v_1\}_{v_0}$. Thus, we have:

$$
\begin{array}{lll}
\{v_1\}_{v_0} \text{ as } \{\!|d'|\!\}_e \triangleleft \Gamma : \theta \ \wedge\ \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} \{\!|d'|\!\}_e & & \\
\Rightarrow \quad \{v_1\}_{v_0} \text{ as } \{\!|d'|\!\}_e \triangleleft \Gamma : \theta \wedge & & \text{(by RWH3,RWT)} \\
\quad \mathsf{fv}(e) \subseteq \mathsf{X} \ \wedge\ \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d' & & \\
\Rightarrow \quad v_0 \text{ as } e \triangleleft \Gamma : \theta_0 \ \wedge\ v_1 \text{ as } d' \triangleleft \Gamma : \theta_1 \wedge & & \text{(by RSDEC)} \\
\quad \mathsf{fv}(e) \subseteq \mathsf{X} \ \wedge\ \theta_0 \cup \theta_1 = \theta \ \wedge\ \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d' & & \\
\Rightarrow \quad v_0 = e\theta_0 \ \wedge\ v_1 \text{ as } d' \triangleleft \Gamma : \theta_1 \wedge & & \text{(by Fact 3.19 (2))} \\
\quad \mathsf{fv}(e) \subseteq \mathsf{X} \ \wedge\ \theta_0 \cup \theta_1 = \theta \ \wedge\ \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\Gamma}_{x} d' & &
\end{array}
$$

It follows that $v_0 \in I(\mathcal{K})$ because $rg(\theta_0) \cup \mathsf{fn}(e) \subseteq \{\theta(y) \mid y \in \mathsf{X} \cap dom(\theta)\} \cup \mathsf{fn}(d) \subseteq I(\mathcal{K})$. Hence, by (I5), also $v_1 \in I(\mathcal{K})$, and the result then follows by the induction hypothesis.

Case $[\![d']\!]_e$. The matching must be due to one of two rules, (RPDec) or (RPSign), where both follow analogously to case of symmetric decryption.

All remaining cases are trivial and follow by the assumptions and the induction hypothesis. □

Theorem 3.21 (Perfect cryptography) *If $\langle \mathsf{N}, \emptyset \rangle \vdash p$ then p guarantees perfect cryptography.*

Proof Assume $d \triangleleft \Gamma$, v and θ such that v as $d \triangleleft \Gamma : \theta$ where $\{v\} \cup \mathsf{fn}(d \triangleleft \Gamma) \subseteq I(\mathcal{K})$. The proof then proceeds by induction on the structure of the inference tree establishing $\langle \mathsf{N}, \mathsf{X} \rangle \vdash d \triangleleft \Gamma'$ showing that if $\Gamma' \subseteq \Gamma$ and $\{\theta(x) \mid x \in \mathsf{X} \cap dom(\Gamma)\} \subseteq I(\mathcal{K})$ then $\{\theta(x) \mid x \in dom(\Gamma')\} \subseteq I(\mathcal{K})$:

Case (RWEmp) holds vacuously.

Case (RWDef) The result follows directly by Lemma 3.20, Fact 3.19 (3), and the induction hypothesis. □

Finally, we shall point out an interesting corollary of this result.

Corollary 3.22 *The judgement $\langle \mathsf{N}, \emptyset \rangle \vdash p$ ensures that for all v and θ where v as $p : \theta$ then $dom(\theta) = \mathsf{bv}(p)$.*

Proof Using Fact 3.19 (1), the result follows from the proof for Theorem 3.21. □

Remark 3.23 *The last two results state nothing about patterns containing free variables. However, it is, in fact, easy to extend the perfect cryptography result to show that: if $\langle \mathsf{N}, \mathsf{X} \rangle \vdash p$ then $p\theta$ guarantees perfect cryptography for all θ. But this result is not required to guarantee perfect cryptography for programs in the language employing the components, which we shall see in Chapter 4, and, thus, it is omitted.*

3.4 Extended Protocol Narrations

In this section we will give some examples on how to use the proposed language extension to model security protocols. As the cryptographic components are supposed to be useable in a large variety of process calculi, we shall not assume anything about the underlying semantics of the communication model. Instead, we shall formalise the protocol in an extended protocol narration (equivalent to that of [22]), where we distinguish between outputs and corresponding inputs, as well as between encryptions and corresponding decryptions.

For convenience and readability we shall employ the naming convention that variables bound at the principal A are named x, usually with a superscript indicating its expected value, and similarly we use the names y and z for the principals B and S, respectively.

We begin with a good example of a protocol that could be modelled using the basic model.

Example 3.24 (Wide Mouthed Frog) *The Wide Mouthed Frog protocol [31] is a symmetric key protocol used for establishing a short term key Kab between two principals A and B, who both trust a server S. In our modelling we shall consider the following version [6]:*

$$\begin{array}{lllll} 1. & A \to S & : & A, \{B, Kab\}_{Ka} \\ 2. & S \to B & : & \{A, Kab\}_{Kb} \\ 3. & A \to B & : & \{M\}_{Kab} \end{array}$$

Here Ka and Kb are master keys that A and B, respectively, are assumed to share with the server S. The key Kab is the session key that A and B share after completion of the protocol, such that they in the last step can communicate the message M encrypted.

This protocol is a commonly used example because of its simplicity and it is also, since it only uses symmetric encryption, a well-suited choice for the basic model. As described above, we shall formulate the protocol in an extended protocol narration and for this we use the encodings and patterns, as well as assume a matching operator as*, similar to that of the semantics. A direct translation into an extended protocol narration in the style of [22] is given in Figure 3.1. Each line in the simple Alice-Bob protocol narration, has been translated into three lines in the extended narration. The first line describes the actions of the sender, the second line then describe how the recipient inputs the message, and the third line describes how the recipient decrypts some of the received elements using pattern matching. Notice that this encoding assumes that the distributed key Kab is fresh, and that S knows the master key of A and B, Ka and Kb respectively.*

$$
\begin{array}{llllll}
1. & A & \rightarrow & & : & \langle A, \{\!|\langle B, Kab\rangle|\!\}_{Ka}\rangle \\
1'. & & \rightarrow & S & : & \langle A, z\rangle \lhd \{z\} \\
1''. & & & S & : & z \text{ as } \{\!|\langle B, z^{Kab}\rangle|\!\}_{Ka} \lhd \{z^{Kab}\} \\
2. & S & \rightarrow & & : & \{\!|\langle A, z^{Kab}\rangle|\!\}_{Kb} \\
2'. & & \rightarrow & B & : & y \lhd \{y\} \\
2''. & & & B & : & y \text{ as } \{\!|\langle A, y^{Kab}\rangle|\!\}_{Kb} \lhd \{y^{Kab}\} \\
3. & A & \rightarrow & & : & \{\!|M|\!\}_{Kab} \\
3'. & & \rightarrow & B & : & y' \lhd \{y'\} \\
3''. & & & B & : & y' \text{ as } \{\!|y^{M}|\!\}_{y^{Kab}} \lhd \{y^{M}\}
\end{array}
$$

Figure 3.1: Wide Mouthed Frog extended narration.

$$
\begin{array}{llllll}
1. & A & \rightarrow & & : & \langle A, \{\!|\langle B, Kab\rangle|\!\}_{Ka}\rangle \\
1'. & & \rightarrow & S & : & \langle A, \{\!|\langle B, z^{Kab}\rangle|\!\}_{Ka}\rangle \lhd \{z^{Kab}\} \\
2. & S & \rightarrow & & : & \{\!|\langle A, z^{Kab}\rangle|\!\}_{Kb} \\
2'. & & \rightarrow & B & : & \{\!|\langle A, y^{Kab}\rangle|\!\}_{Kb} \lhd \{y^{Kab}\} \\
3. & A & \rightarrow & & : & \{\!|M|\!\}_{Kab} \\
3'. & & \rightarrow & B & : & \{\!|y^{M}|\!\}_{y^{Kab}} \lhd \{y^{M}\}
\end{array}
$$

Figure 3.2: Wide Mouthed Frog optimised narration.

This encoding, however, can be optimised as all the decryptions can be incorporated into the input patterns directly; the resulting narration is given in Figure 3.2.This optimised encoding is as detailed about the actions of the participants as the former encoding, but the readability is much higher. It is also noticeable that this encoding uses less variables, as it is not necessary to store temporary values in temporary variables. Naturally, however, if the temporary variables were still needed for some reason, they could easily be incorporated; either through the more tedious narration above or through the sub-decodings of variables in the extended model.

The next example protocol uses asymmetric cryptography, making it ideal for the revised model.

Example 3.25 (Needham-Schroeder Public Key) *The Needham-Schroeder public key protocol [84] is an authentication protocol, with the purpose of mutually authenticating two principals A and B. In our modelling we shall consider the*

$$
\begin{array}{llllll}
1. & A & \rightarrow & & : & [\langle Na, A\rangle]_{Kb+} \\
1'. & & \rightarrow & B & : & [\![\langle y^{Na}, A\rangle]\!]_{Kb-} \triangleleft [y^{Na}] \\
2. & B & \rightarrow & & : & [\langle y^{Na}, Nb\rangle]_{Ka+} \\
2'. & & \rightarrow & A & : & [\![\langle Na, x^{Nb}\rangle]\!]_{Ka-} \triangleleft [x^{Nb}] \\
3. & A & \rightarrow & & : & [x^{Nb}]_{Kb+} \\
3'. & & \rightarrow & B & : & [\![Nb]\!]_{Kb-} \triangleleft [\,]
\end{array}
$$

Figure 3.3: Needham-Schroeder Public Key extended narration.

usual version:

$$
\begin{array}{lll}
1. & A \rightarrow B & : [Na, A]_{Kb+} \\
2. & B \rightarrow A & : [Na, Nb]_{Ka+} \\
3. & A \rightarrow B & : [Nb]_{Kb+}
\end{array}
$$

In brief, the principals are authenticated by exchanging the nonces Na and Nb. Note that the protocol assumes that A and B know each other's public key prior to the protocol run. The original version of the protocol (in [84]) includes steps for retrieval of the public keys from a trusted third party, but these steps are usually omitted for simplicity and readability.

Again we may easily translate this into an extended protocol narration using the language components. The result of this translation is given in Figure 3.3. The translation is straightforward, although it is noteworthy that no variables are bound in the third step, as this is a simple confirmation message for B.

The above two protocols are chosen as running examples in the remainder of this dissertation, not only because they are relatively simple and well-known to the community, but also because they are infamous for the attacks that exist on them. Especially the Needham-Schroeder Public Key has gained notoriety after it took more than one and a half decades from the protocol was first published in [84] till the attack was found in [70]. An example of why automated protocol analysis and verification is necessary.

3.5 Discussion

When designing a language or language components, there are always several considerations and choices one must take. Below we shall discuss some of the most interesting aspects of our development.

3.5.1 Binding Auxiliary Variables

The semantics does not allow introduction of auxiliary variables which do not correspond to a structural sub-term of the pattern matching; e.g. the pattern $(x \lhd \{x \mapsto *, y \mapsto \mathbf{\{}x\mathbf{\}}_k\})$ would never result in the binding of y and it is also correctly discarded by the well-formedness condition. But, although it is clearly obvious what the intention of the pattern above is, it should be considered an example of bad programming style, instead of changing the semantics. The program should always explicitly reflect that the introduced y represents a newly composed value (as opposed to a sub-value of an existing value), and a much clearer formulation uses a two-step approach by first binding x and afterwards matching $\mathbf{\{}x\mathbf{\}}_k$ to a binding occurrence of y.

3.5.2 Hashing and Other Cryptographic Primitives

We mentioned in §3.2 that many modern protocols rely on more than just symmetric cryptography, and that was the motivation for introducing asymmetric encryption and digital signatures. But although asymmetric and symmetric cryptography accounts for most common protocols, there are also other primitives that one could find interesting.

A cryptographic hash function, or just *hashing*, is the transformation of input to a fixed-size string, called the *message digest*. The perfect cryptography assumption for hashing dictates two properties: (1) it is a one-way function (i.e. it is impossible to recreate the original input from the message digest); (2) it is collision free (i.e. different inputs give different message digests).

Obviously such a construct could easily be added to the language components, but, in fact, the revised model can already simulate a function with the above properties. Notice that an asymmetrically encrypted value, encrypted with a key that is not part of a key pair, can never be decrypted. Thus, any asymmetric encryption with such a key represents a one-way function. Indeed, if we introduced a unique name H and assumed it publicly available, then asymmetric encryption using this name would correctly model hashing; e.g. a hashed value of the tuple $\langle A, B\rangle$ would be modelled as $\mathbf{[}\langle A, B\rangle\mathbf{]}_H$.[6]

Apart from hashing, there are several other cryptographic operations that are used in protocols. Some may be easily added to the cryptographic constructs, whereas others may not be easily translated to an algebraic abstraction. For the

[6]This approach could also be used for modelling *message authentication codes (MAC)*, simply by substituting the secret key for H.

interested reader we refer to [89] where we show how the cryptographic function *blinding*, which is used in electronic voting systems and for anonymous digital payment, can be modelled and analysed; this work also led to the discovery of a flaw in a well-known voting protocol (see [86]).

3.5.3 Distinguishing Between Encoding and Decoding

In [30] a calculus is suggest modelling cryptography with the same operator for encryption and decryption, and, based on this work, we introduced a stand-alone model for cryptographic components in [87] that took a similar approach. This approach results in an elegant and persuasive model, which needs only one term language, instead of both encodings and decodings, and if the language is restricted to symmetric encryption, it is also theoretically sound.

But, in addition to the obvious drawback that the abstraction makes the it impossible to perform analyses where the difference between encryption and decryption is important (e.g. timing analyses), this modelling approach yields complications when dealing with asymmetric cryptography.

This is easily illustrated. Assume that the symmetric operator $\{\cdot\}$. and the asymmetric operator $[\cdot]$. are used for modelling both encryption and decryption, which would result in a semantics with rules along the lines of

$$(1)\quad \frac{v_0 \text{ as } e_0 : \theta_0 \qquad v_1 \text{ as } e_1 \lhd \Gamma : \theta_1}{\{v_1\}_{v_0} \text{ as } \{e_1\}_{e_0} \lhd \Gamma : \theta_0 \cup \theta_1} \qquad (2)\quad \frac{n^- \text{ as } e_0 : \theta_0 \qquad v \text{ as } e_1 \lhd \Gamma : \theta_1}{[v]_{n^+} \text{ as } [e_1]_{e_0} : \theta_0 \cup \theta_1}$$

$$(3)\quad \frac{n^+ \text{ as } e_0 \lhd \Gamma : \theta_0 \qquad v \text{ as } e_1 \lhd \Gamma : \theta_1}{[v]_{n^-} \text{ as } [e_1]_{e_0} \lhd \Gamma : \theta_0 \cup \theta_1}$$

where each rule is also guarded by $\theta_0 =_s \theta_1$.

Intuitively this may seem correct, but there is a caveat. By definition, symmetric encryption and decryption requires the keys to match, but by abusing the rules for asymmetric decryption, we may now violate this rule; e.g. the encoding $\{m_1\}_{[m_2]_{n^+}}$ matches $\{m_1\}_{[m_2]_{n^-}}$, and not itself, according to the semantics above. This problem is not easily dealt with, as a key property of the cryptographic components is that keys may also be composite values, and thus the distinction between encodings and decodings is required.

3.6 Concluding Remarks

This chapter developed an expressive syntax, general semantics and a notion of well-formedness for capturing the assumptions of perfect cryptography. The syntax builds on and refines ideas found in the Spi-calculus [6], in LySa [22, 24], and in LySa^{NS} [30], in order to express patterns that allows derivation of a number of secrets from a single transmitted message, as is common in security protocols.

The formal development has taken the form of defining a semantics that can also deal with imperfect cryptography (as when secret keys can be broken by brute force attack or successful guessing) which has been supplemented by general well-formedness conditions for ruling out those behaviours not allowed when assuming perfect cryptography.

The theorems established in §3.3 aim at showing that the well-formedness conditions are sufficiently restrictive that no improper behaviour is admitted; conversely the example protocols modelled in §3.4 aim at showing that the well-formedness conditions are sufficiently flexible to be of widespread interest.

In Chapter 4 we will show how the cryptographic components can be incorporated into a process calculus, but they are, by no means, restricted to this particular language. In fact, we believe that any process calculus with a notion of communication, could be extended with the components, thereby allowing the language designer to choose the communication model that fits.

Chapter 4

The Language

4.1 Introduction

Having built the cryptographic primitives, we now turn to the design of the process calculus. The calculus should include operators for sending and receiving messages, but apart from that we have few requirements for the language.

In the Horn clause model we used a predicate net to model all communication. This is in line with many language-based approaches, everything happens on a single, global communication medium. In that respect, an obvious choice of calculus would simply have an operation for sending an encoding and receiving with a pattern, apart from the usual components for parallel composition, sequential composition, etc.

Modern IT-systems, however, rarely have such a simple structure. With respect to security, it is safe to assume all communication as global, but it is often much too coarse an assumption to be of any practical use. Most systems rely heavily on infrastructure to obtain the required security properties and that cannot be modelled in a calculus where all communication is visible to everyone. Instead we shall adopt ideas from languages with distribution and build our language around the process calculus Klaim.

KLAIM [17] is a language specifically designed for describing distributed systems with structured communication. It allows programmers to distribute processes and specify the scope of communication. Informally we may say that KLAIM allows several net's, as opposed to just the one we used in the Horn clause modelling. KLAIM processes are sequential processes which communicate across the so-called *tuple spaces*. These tuple space are multisets of tuples, corresponding to our values, and interaction is established by placing and retrieving tuples via pattern matching mechanisms.

But in order to correctly simulate cryptographic protocols, the language must be extended with the cryptographic constructs of Chapter 3. We shall formally specify such an extension of KLAIM and name it CryptoKlaim.

The remainder of this chapter is structured as follows. In §4.2 we present the syntax of CryptoKlaim and give an informal description of how each construct in the language behaves. This is properly formalised in §4.3 in the form of a reduction semantics for the language. We then extend the well-formedness condition of the components to the language setting in §4.4 and define the notion of a program. In §4.5 we show that programs remain well-formed under evaluation. Finally, in §4.6, we show how protocols can be formulated in the language, before we in §4.7 discuss some theoretical aspects of the development.

4.2 Syntax

We now introduce CryptoKlaim. It is defined via three syntactic categories as presented in Table 4.1: actions, processes, and nets. The syntax relies on the cryptographic components, as defined in Chapter 3. The main entity of the calculus, nets, consists of parallel compositions of located processes or values. A net, thus, is the finite collection of *nodes*, representing executable processes or sent data, each associated with a *locality* identifying the tuple space it is placed on. Localities are used as addresses (i.e. network references) of nodes, and are the syntactic ingredient used to expressing the idea of administrative domain: computations at a given locality is under the control of a specific authority. We assume the existence of a countable infinite set of such localities, $\mathcal{L}$, ranged over by $l, l', l_0, l_1, \ldots,$, and disjoint from the sets of names and variables.

To allow recursion and infinite behaviour the language supports process invocation. Thus, execution requires a process environment, Δ, which is a (static, finite, partial) map from process identifiers to processes; we write[1] $A[x_1, \ldots, x_k] \triangleq P$

[1] We follow the process calculus tradition of denoting process definition with square brackets and invocation with parenthesis.

a	::=	$\mathsf{out}\, e$	Local Output
	|	$\mathsf{out}\, e@l$	Output, $l \in \mathcal{L}$
	|	$\mathsf{in}\, p$	Local Input
	|	$\mathsf{in}\, p@l$	Input, $l \in \mathcal{L}$
	|	$e\ \mathsf{as}\ p$	Pattern matching
P	::=	0	Terminated process
	|	$a.P$	Action prefixing
	|	$P_1 \mid P_2$	Parallel composition
	|	$A(e_1, \ldots, e_k)$	Invocation
	|	$(\nu^T\ n)\ P$	Local name restriction ($T \in \{\pm, \epsilon\}$)
N	::=	$l :: P$	Single node, $l \in \mathcal{L}$
	|	$l :: \langle v \rangle$	Located value, $l \in \mathcal{L}$
	|	$N_1 \parallel N_2$	Net composition
	|	$(\nu^T\ n)\ N$	Global name restriction ($T \in \{\pm, \epsilon\}$)

Table 4.1: Language syntax.

whenever the identifier $A[x_1, \ldots, x_k]$ is mapped to a process P in Δ.

Below we informally discuss the meanings of the constructs in each of the syntactic categories.

Actions. The behaviour of a process is described through its actions upon the tuple spaces. These actions are defined in Table 4.1 and depend syntactically on the cryptographic components, encodings and patterns, as introduced in Chapter 3.

(1) Local output, written $\mathsf{out}\, e$, outputs a value to the tuple space the action is executed in. The encoding being output is ensured to be a value (i.e., closed) as we are assuming a substitution-based semantics and all variables in e will be replaced by their corresponding mapping prior to execution of the action. Local output models communication between concurrent processes controlled by the same authority; e.g. communication between a server and a database.

(2) Output, written $\mathsf{out}\, e@l$, is as (1) except that the output value will be placed in the locality l instead of locally. This allows communication across tuple spaces and between computations controlled by different authorities (or principals).

(3) Local input, written $\mathsf{in}\, p$, inputs a value from the tuple space the action is executed from. Input works according to the semantics of the cryptographic components as defined in Chapter 3: (*i*) first a value on the tuple space that matches the pattern is chosen, if one such exists; (*ii*) the value is matched against the pattern according to the semantics of pattern matching; and (*iii*) the resulting substitution of the matching is used for updating the continuation process. In the case that multiple values match the pattern, one will be chosen non-deterministically, such that execution of the process *may* use any of the eligible values to proceed. Naturally, further execution is blocked until a successful input has been recorded.

(4) Input, written $\mathsf{in}\, p@l$, is as (3) except that the value is chosen from l instead of locally.

(5) In addition to input and output, a process may also use an in-lined, explicit pattern matching action, by directly invoking the matching construct $e\, \mathsf{as}\, p$. The consequence of matching is the same as outputting e (or rather, the value it corresponds to when all variables are substituted away), whereafter the value is again input and matched against p. The advantage, however, of the explicit matching is obvious, as it avoids any unintended, unnecessary interaction with concurrent processes, while allowing introduction of auxiliary variables. Clearly the matching action blocks further execution if e and p do not match.

Processes. Processes are then inductively defined in Table 4.1 and depend syntactically on actions and names.

(1) The nil process, 0, denotes a process that can do nothing. Technically, this is the terminal process that is the end of all things.

(2) Action prefixing, $a.P$, denotes the sequential process that first executes the action a, awaits termination, whereafter the continuation process P is updated with any variable bindings from a and executed.

(3) Parallel composition, $P_1 \mid P_2$, where P_1 and P_2 may proceed independently of each other and communication is asynchronous, using the tuple spaces as communication medium. Consider, for example, the concurrent processes:

$$\mathsf{out}\, v@l.P_1 \mid \mathsf{in}\, p@l.P_2$$

Communication between these processes requires two steps: (*i*) the leftmost process will place the value v on the tuple space indicated by the locality l; whereafter, (*ii*) the rightmost process will input v from l, if it matches the pattern p, update the continuation process P_2 with any variable bindings

the matching resulted in, and execute the resulting process. Sometimes we may consider a parallel composition of a family of processes P_i with indices from set I, and abbreviate it with the notation $|_{i\in I}\ P_i$.

(4) Process invocation, $A(e_1, \ldots, e_k)$, assumes that a process definition exist $A[x_1, \ldots, x_k] \triangleq P$; if this is the case, then all occurrences of the variables x_1 to x_k in P are replaced by the values corresponding to, e_1 to e_k, respectively, and the resulting process is then executed.

(5) The constructor, $(\nu^T\ n)\ P$, is simply a scoping operator for names, syntactically indicating the defining occurrence. The superscript T may either be: ϵ, binding the name $n^\epsilon \in \mathcal{N}$ within the scope of the process P; or $\pm$, binding the names $n^+, n^- \in \mathcal{N}$ within the scope of P, as well as indicating that $\langle n^+, n^-\rangle$ is a key pair. Naming scopes may be expanded or contracted to allow communication; e.g. two concurrent processes P_1 and P_2 will only agree on a name n^ϵ that occurs free in both processes, if the occurrences belong to the same scope, such as in $(\nu^\epsilon\ n)\ (P_1 \mid P_2)$. As for parallel composition, we may sometimes need to create a family of scopes $(\nu^T\ n_1)\ \cdots (\nu^T\ n_k)\ P$ over a process P and which we shall abbreviate with the notation $(\nu^T_{i\in\{1,\ldots,k\}}\ n_i)\ P$.

In the following, we shall impose a notational convention for the scoping constructors:

Notation 4.1 *The superscripts in scoping constructors, $\pm$ and ϵ, are shorthand notations for the sets $\{+,-\}$ and $\{\epsilon\}$, respectively.*

This gives a more succinct definition of various relations. We may also use the term T as a placeholder for either $\pm$ or ϵ, whenever both are applicable.

Nets. Nets are inductively defined in Table 4.1. They are the main entities of the language and depend syntactically on processes and values (i.e. encodings without variables).

(1) The node $l :: P$ is a located process P at the tuple space l. The locality of the process determines where locally output values are placed and which values that are available for locale inputs. In this way, the locality decides the local resources of the process, and, thus, two syntactically equivalent processes may behave differently in different localities. Note that the process may still access resources from other localities as well, by explicitly specifying this.

(2) The node $l :: v$ is a located value v at the tuple space l. Located values are used for communication between processes and the locality decides which processes the value can be accessed by.

(3, 4) Parallel composition and name scoping works similarly to their process counterparts.

In §4.4 we shall formally define the notion of a *program*. Until then, it may be convenient to think of a program as the combination of: (i) a tuple space environment, $\mathsf{L} \subseteq \mathcal{L}$; (ii) a net, N; and (iii) an associated process environment, Δ, mapping each process identifier occurring in N to a process.

Free names and variables. Finally, the notions free names and variables are extended to the language in Table 4.2 and Table 4.3, respectively. Again the definitions are straightforward, relying on their equivalent counterparts for the cryptographic components, and respecting the naming scopes of the language.

$$
\begin{array}{rcl}
\mathsf{fn}(l :: P) & \stackrel{\text{def}}{=} & \mathsf{fn}(P) \\
\mathsf{fn}(l :: \langle v \rangle) & \stackrel{\text{def}}{=} & \mathsf{fn}(v) \\
\mathsf{fn}(N_1 \parallel N_2) & \stackrel{\text{def}}{=} & \mathsf{fn}(N_1) \cup \mathsf{fn}(N_2) \\
\mathsf{fn}((\nu^T\ n)\ N) & \stackrel{\text{def}}{=} & \mathsf{fn}(N) \backslash \{n^\tau \mid \tau \in T\} \\
\mathsf{fn}(0) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fn}(\mathsf{out}\, e.P) & \stackrel{\text{def}}{=} & \mathsf{fn}(e) \cup \mathsf{fn}(P) \\
\mathsf{fn}(\mathsf{out}\, e@l.P) & \stackrel{\text{def}}{=} & \mathsf{fn}(e) \cup \mathsf{fn}(P) \\
\mathsf{fn}(\mathsf{in}\, p.P) & \stackrel{\text{def}}{=} & \mathsf{fn}(p) \cup \mathsf{fn}(P) \\
\mathsf{fn}(\mathsf{in}\, p@l.P) & \stackrel{\text{def}}{=} & \mathsf{fn}(p) \cup \mathsf{fn}(P) \\
\mathsf{fn}(e\ \mathsf{as}\ p.P) & \stackrel{\text{def}}{=} & \mathsf{fn}(e) \cup \mathsf{fn}(p) \cup \mathsf{fn}(P) \\
\mathsf{fn}(P_1 \mid P_2) & \stackrel{\text{def}}{=} & \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) \\
\mathsf{fn}(A(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \mathsf{fn}(e_1) \cup \cdots \cup \mathsf{fn}(e_k) \\
\mathsf{fn}((\nu^T\ n)\ P) & \stackrel{\text{def}}{=} & \mathsf{fn}(P) \backslash \{n^\tau \mid \tau \in T\}
\end{array}
$$

Table 4.2: Free names; fn.

Remark 4.2 *Note that we assume that whenever* $A[x_1, \ldots, x_k] \triangleq P$ *then* $\mathsf{fn}(P) = \emptyset$ *and* $\mathsf{fv}(P) \subseteq \{x_1, \ldots, x_k\}$. *This is due to technical reasons (variable name capture, see §4.7), and will be enforced by the well-formedness condition introduced in §4.4.*

$$
\begin{array}{rcl}
\mathsf{fv}(l :: P) & \stackrel{\text{def}}{=} & \mathsf{fv}(P) \\
\mathsf{fv}(l :: \langle v \rangle) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(N_1 \parallel N_2) & \stackrel{\text{def}}{=} & \mathsf{fv}(N_1) \cup \mathsf{fv}(N_2) \\
\mathsf{fv}((\nu^T\ n)\ N) & \stackrel{\text{def}}{=} & \mathsf{fv}(N) \\
\\
\mathsf{fv}(0) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(\mathsf{out}\, e.P) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup \mathsf{fv}(P) \\
\mathsf{fv}(\mathsf{out}\, e@l.P) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup \mathsf{fv}(P) \\
\mathsf{fv}(\mathsf{in}\, p.P) & \stackrel{\text{def}}{=} & (\mathsf{fv}(p) \cup \mathsf{fv}(P)) \backslash \mathsf{bv}(p) \\
\mathsf{fv}(\mathsf{in}\, p@l.P) & \stackrel{\text{def}}{=} & (\mathsf{fv}(p) \cup \mathsf{fv}(P)) \backslash \mathsf{bv}(p) \\
\mathsf{fv}(e\ \mathsf{as}\, p.P) & \stackrel{\text{def}}{=} & \mathsf{fv}(e) \cup (\mathsf{fv}(p) \cup \mathsf{fv}(P)) \backslash \mathsf{bv}(p) \\
\mathsf{fv}(P_1 \mid P_2) & \stackrel{\text{def}}{=} & \mathsf{fv}(P_1) \cup \mathsf{fv}(P_2) \\
\mathsf{fv}(A(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \mathsf{fv}(e_1) \cup \cdots \cup \mathsf{fv}(e_k) \\
\mathsf{fv}((\nu^T\ n)\ P) & \stackrel{\text{def}}{=} & \mathsf{fv}(P)
\end{array}
$$

Table 4.3: Free variables; fv.

4.3 Semantics

In this section, the informal description of the behaviour of CryptoKlaim as given in §4.2 is made precise by a concrete semantics. Following the process calculus tradition, the semantics shall define how nets evolve in a step-by-step fashion. Specifically we shall give an operational semantics, formalised by a binary relation over nets called the *reduction relation*. The reduction relation holds between a pair of nets, written $N \rightarrow N'$, precisely when N can evolve into N'.

An aim of a reduction semantics is that the definition of the reduction relation should be kept simple and only focus on central behavioural aspects, so it will usually require the components to be on a specific form to match the rules. Hence, in order to loosen up these rigid requirements, some syntactic manipulations of nets are introduced before moving to the reduction relation itself.

4.3.1 Structural Congruence

The concept of a structural congruence was first introduced in Berry and Boudol's Chemical Abstract Machine [16] and later transferred to process calculi by Milner [80]. The idea is that two nets are considered equal when they only differ in

syntactic aspects of no importance to the way nets evolve. The *structural congruence relation*, $N \equiv N'$ is defined as the smallest relation satisfying the rules in Table 4.4, and the definition of structural congruence is mostly trivial:

(Reflex) through (Assoc) establish reflexivity, symmetry, and transitivity of the structural equivalence and provide standard definitions of commutativity and associativity for parallel compositions of nets;

(Comp1) and (Comp2) show how structural changes in parallel branches or within scopes imply structural equivalence of the composed system;

(Clone) and (Abs) show that parallel process nodes with the same locality behave equivalently regardlessly of whether they are represented in one or two nodes, and that parallel composition have the nil process, 0, as neutral element; and

(Res1) through (Res4) allow *scope extrusion*, i.e. they define how naming scopes may be expanded or contracted without changing the meaning of the net.

The rule (Alpha), however, is noteworthy. This rule says that two nets, N_1 and N_2, are considered structural congruent, if they are α-*equivalent*, written $N_1 =_\alpha N_2$. That two processes are α-equivalent means that they are identical, except that they possibly differ in the naming of bound names.[2]

The binary relation α-equivalence is defined in Table 4.6 and relies on the procedure of replacing all occurrences of a bound name in a process with another name, called α-*renaming* (or sometimes α-conversion), which is defined in Table 4.5. The definition of α-renaming is relatively straightforward and it is only interesting to point out that the substitution must respect the naming scopes; i.e. when a free occurrence of a name m^τ is substituted into a process where m^τ is already bound, the free m^τ becomes spuriously bound or captured, called *name capture*. This is unintended behaviour, as it changes the binding occurrence of m^τ, thereby its meaning, and thus the binding occurrence must be renamed to avoid the name capture. As already mentioned in Remark 3.3, α-renaming on the cryptographic components is straightforward, as they contain no defining occurrences of names and thus no name capture can take place.[3]

[2] For readers with programming experience it may be convenient to think of α-equivalent nets as identical programs, except some syntactic parts (such as function or variable names, datatypes, etc.) are named differently; indeed the programs will still behave equivalently.

[3] It is interesting to note here that we define only α-equivalence on bound names and not variables. Naturally, α-equivalence could also be defined for bound variables, but this is not necessary for the theoretical development and is therefore left out for the sake of simplicity.

(Reflex)	$N \equiv N$
(Sym)	$\dfrac{N \equiv N'}{N' \equiv N}$
(Trans)	$\dfrac{N \equiv N'' \quad N'' \equiv N'}{N \equiv N'}$
(Comm)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$
(Assoc)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(Comp1)	$\dfrac{N_1 \equiv N_1'}{N_1 \parallel N_2 \equiv N_1' \parallel N_2}$
(Comp2)	$\dfrac{N \equiv N'}{(\nu^T\ n)\ N \equiv (\nu^T\ n)\ N'}$
(Clone)	$l :: (P_1 \mid P_2) \equiv l :: P_1 \parallel l :: P_2$
(Abs)	$l :: P \equiv l :: (P \mid 0)$
(Res1)	$l :: (\nu^T\ n)\ P \equiv (\nu^T\ n)\ (l :: P)$
(Res2)	$(\nu^{T_1}\ n_1)\ (\nu^{T_2}\ n_2)\ N \equiv (\nu^{T_2}\ n_2)\ (\nu^{T_1}\ n_1)\ N$
(Res3)	$l :: 0 \equiv (\nu^T\ n)\ (l :: 0)$
(Res4)	$\dfrac{\forall \tau \in T : n^\tau \notin \mathsf{fn}(N_1)}{N_1 \parallel (\nu^T\ n)\ N_2 \equiv (\nu^T\ n)\ (N_1 \parallel N_2)}$
(Alpha)	$\dfrac{N_1 =_\alpha N_2}{N_1 \equiv N_2}$

Table 4.4: Structural congruence; $N \equiv N'$.

$$
\begin{array}{rcl}
0[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & 0 \\
(\mathsf{out}\, e.P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \mathsf{out}\, e[n^\tau \mapsto m^\tau].P[n^\tau \mapsto m^\tau] \\
(\mathsf{out}\, e@l.P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \mathsf{out}\, e[n^\tau \mapsto m^\tau]@l.P[n^\tau \mapsto m^\tau] \\
(\mathsf{in}\, p.P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \mathsf{in}\, p[n^\tau \mapsto m^\tau].P[n^\tau \mapsto m^\tau] \\
(\mathsf{in}\, p@l.P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \mathsf{in}\, p[n^\tau \mapsto m^\tau]@l.P[n^\tau \mapsto m^\tau] \\
(e\, \mathsf{as}\, p.P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & e[n^\tau \mapsto m^\tau]\, \mathsf{as}\, p[n^\tau \mapsto m^\tau].P[n^\tau \mapsto m^\tau] \\
(P_1 \mid P_2)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & P_1[n^\tau \mapsto m^\tau] \mid P_2[n^\tau \mapsto m^\tau] \\
A(e_1, \ldots, e_k)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & A(e_1[n^\tau \mapsto m^\tau], \ldots, e_k[n^\tau \mapsto m^\tau]) \\
((\nu^T\ k)\ P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \begin{cases} (\nu^T\ k)\ P[n^\tau \mapsto m^\tau] & \text{if } \forall \tau \in T : k^\tau \notin \{n^\tau, m^\tau\} \\ (\nu^T\ k)\ P & \text{if } \exists \tau \in T : k^\tau = n^\tau \\ (\nu^T\ k_2)\ (P\theta)[n^\tau \mapsto m^\tau] & \text{if } \exists \tau \in T : k^\tau = m^\tau \\ \quad \text{and } \forall \tau \in T : k_2^\tau \notin (\mathsf{fn}(P) \cup \{m^\tau\}) & \\ \quad \text{and } \theta = \{k^\tau \mapsto k_2^\tau \mid \tau \in T\} & \end{cases}
\end{array}
$$

Table 4.5: α-renaming of processes; $P[n^\tau \mapsto m^\tau]$.

$$
\begin{array}{rcl}
(l :: P)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & l :: P[n^\tau \mapsto m^\tau] \\
(l :: \langle v \rangle)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & l :: \langle v[n^\tau \mapsto m^\tau] \rangle \\
(N_1 \parallel N_2)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & N_1[n^\tau \mapsto m^\tau] \parallel N_2[n^\tau \mapsto m^\tau] \\
((\nu^T\ k)\ N)[n^\tau \mapsto m^\tau] & \stackrel{\text{def}}{=} & \begin{cases} (\nu^T\ k)\ N[n^\tau \mapsto m^\tau] & \text{if } \forall \tau \in T : k^\tau \notin \{n^\tau, m^\tau\} \\ (\nu^T\ k)\ N & \text{if } \exists \tau \in T : k^\tau = n^\tau \\ (\nu^T\ k_2)\ (N\theta)[n^\tau \mapsto m^\tau] & \text{if } \exists \tau \in T : k^\tau = m^\tau \\ \quad \text{and } \forall \tau \in T : k_2^\tau \notin (\mathsf{fn}(P) \cup \{m^\tau\}) & \\ \quad \text{and } \theta = \{k^\tau \mapsto k_2^\tau \mid \tau \in T\} & \end{cases}
\end{array}
$$

Table 4.6: α-renaming of nets; $N[n^\tau \mapsto m^\tau]$.

$$
\begin{array}{lll}
\text{(AEQ)} \quad (\nu^T\ n)\ N =_\alpha (\nu^T\ m)\ N\theta & \text{if } \forall \tau \in T : m^\tau \notin \mathsf{fn}(N) \\
& \text{and } \theta = \{n^\tau \mapsto m^\tau \mid \tau \in T\}
\end{array}
$$

Table 4.7: α-equivalence; $N_1 =_\alpha N_2$.

Remark 4.3 *The structural congruence rule* (Res3) *should be conceived as for garbage collection purposes only; e.g. assume* $(\nu^{\epsilon}\ n)\ (l :: P)$ *where* $n^{\epsilon} \notin \mathsf{fn}(l :: P)$*, then the restriction can be removed as follows:*

$$
\begin{array}{lll}
& (\nu^{\epsilon}\ n)\ (l :: P) & \\
\equiv & (\nu^{\epsilon}\ n)\ (l :: (P \mid 0)) & (\text{by Abs}) \\
\equiv & (\nu^{\epsilon}\ n)\ (l :: P \parallel l :: 0) & (\text{by Comp2,Clone}) \\
\equiv & l :: P \parallel (\nu^{\epsilon}\ n)\ (l :: 0) & (\text{by Res4}) \\
\equiv & l :: P \parallel l :: 0 & (\text{by Comp1,Res3}) \\
\equiv & l :: P & (\text{by Clone,Abs})
\end{array}
$$

This is its only purpose, because if rule (Res3) *instead is used to introduce a fresh name scope then, due to the nature of* (Res4)*, any process or net within that scope is not allowed to contain a free occurrence of the fresh name.*

Finally, we require the notion of substitution to be extended to cover processes as well. This is done in Table 4.8 in a straightforward manner, relying on substitution on the cryptographic components as defined in Chapter 3 and α-renaming of processes. Note that, as substitutions are always from variables to values and we assume that the set of names, $\mathcal{N}$, and variables, $\mathcal{X}$, are disjoint, a substitution-based semantics will not lead to variable name capture, and thus we do not need to define α-equivalence of variables.

$$
\begin{array}{rcl}
0[x \mapsto v] & \stackrel{\text{def}}{=} & 0 \\
(\mathsf{out}\ e.P)[x \mapsto v] & \stackrel{\text{def}}{=} & \mathsf{out}\ e[x \mapsto v].P[x \mapsto v] \\
(\mathsf{out}\ e@l.P)[x \mapsto v] & \stackrel{\text{def}}{=} & \mathsf{out}\ e[x \mapsto v]@l.P[x \mapsto v] \\
(\mathsf{in}\ p.P)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{in}\ p[x \mapsto v].P[x \mapsto v] & \text{if } x \notin \mathsf{bv}(p) \\ \mathsf{in}\ p.P & \text{if } x \in \mathsf{bv}(p) \end{cases} \\
(\mathsf{in}\ p@l.P)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{in}\ p[x \mapsto v]@l.P[x \mapsto v] & \text{if } x \notin \mathsf{bv}(p) \\ \mathsf{in}\ p@l.P & \text{if } x \in \mathsf{bv}(p) \end{cases} \\
(e\ \mathsf{as}\ p.P)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} e[x \mapsto v]\ \mathsf{as}\ p[x \mapsto v].P[x \mapsto v] & \text{if } x \notin \mathsf{bv}(p) \\ e[x \mapsto v]\ \mathsf{as}\ p.P & \text{if } x \in \mathsf{bv}(p) \end{cases} \\
(P_1 \mid P_2)[x \mapsto v] & \stackrel{\text{def}}{=} & P_1[x \mapsto v] \mid P_2[x \mapsto v] \\
A(e_1, \ldots, e_k)[x \mapsto v] & \stackrel{\text{def}}{=} & A([e_1[x \mapsto v], \ldots, e_k[x \mapsto v]) \\
((\nu^{T}\ n)\ P)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} (\nu^{T}\ n)\ P[x \mapsto v] & \text{if } \forall \tau \in T : n^{\tau} \notin \mathsf{fn}(v) \\ (\nu^{T}\ m)\ (P\theta)[x \mapsto v] & \text{if } \exists \tau \in T : n^{\tau} \in \mathsf{fn}(v) \end{cases} \\
& & \quad \text{and } \forall \tau \in T : m^{\tau} \notin (\mathsf{fn}(v) \cup \mathsf{fn}(P)) \\
& & \quad \text{and } \theta = \{n^{\tau} \mapsto m^{\tau} \mid \tau \in T\}
\end{array}
$$

Table 4.8: Substitution; $P[x \mapsto v]$.

4.3.2 Reduction Relation

We are now ready to present the reduction relation itself. The reduction relation, $\rightarrow$, is the least relation induced by the rules in Table 4.9. The semantics follows

(L-Out)	$l :: \mathsf{out}\, v.P \rightarrow l :: P \parallel l :: \langle v \rangle$
(Out)	$l :: \mathsf{out}\, v@l'.P \rightarrow l :: P \parallel l' :: \langle v \rangle$
(L-In)	$\dfrac{v \ \mathsf{as}\ p : \theta}{l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle \rightarrow l :: P\theta}$
(In)	$\dfrac{v \ \mathsf{as}\ p : \theta}{l :: \mathsf{in}\, p@l'.P \parallel l' :: \langle v \rangle \rightarrow l :: P\theta}$
(Match)	$\dfrac{v \ \mathsf{as}\ p : \theta}{l :: v \ \mathsf{as}\, p.P \rightarrow l :: P\theta}$
(Par)	$\dfrac{N_1 \rightarrow N_1'}{N_1 \parallel N_2 \rightarrow N_1' \parallel N_2}$
(Invoc)	$\dfrac{A[x_1, \ldots, x_k] \triangleq P}{l :: A(v_1, \ldots, v_k) \rightarrow l :: P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]}$
(New)	$\dfrac{N \rightarrow N'}{(\nu^T\ n)\ N \rightarrow (\nu^T\ n)\ N'}$
(Struct)	$\dfrac{N \equiv N'' \quad N'' \rightarrow N''' \quad N''' \equiv N'}{N \rightarrow N'}$

Table 4.9: Reduction semantics of nets; $N \rightarrow N'$.

the intuitive definition given in §4.2, and consists of eight rules:

(L-Out) and (Out) simply output values to the designated localities;

(L-In) and (In) shows how input behaves, by invoking the semantics of pattern matching;

(Match) directly invokes the semantics of pattern matching;

(Invoc) says that the process invocation evaluates into its invoked process counterpart.

(New) and (Par) shows that reductions are allowed within naming scopes or in parallel sub-branches; and

(Struct) loosens some of the structural requirements of the relation by allowing structural changes on the nets in accordance with the structural equivalence.

As usual, the semantics is best understood through a couple of examples.

Example 4.4 (α-renaming) *Consider the net:*

$$l :: \left((\nu^{\epsilon}\ n)\ \mathsf{out}\ n.P_1' \quad | \quad (\nu^{\epsilon}\ n)\ \mathsf{in}\ x \triangleleft [x].P_2'\right)$$

Now, obviously, the processes should communicate locally, resulting in that x is mapped to n in process P_2'. However, the processes do not agree on the meaning of n, hence these processes cannot communicate as is. Therefore, as allowed by the rule (Struct)*, we shall use the structural equivalence to put the net on a form that allows the communication to happen. Assume that $m \notin \mathsf{fn}(P_1') \cup \mathsf{fn}(P_2')$ then:*

$$\begin{array}{lll}
& l :: ((\nu^{\epsilon}\ n)\ \mathsf{out}\ n.P_1' \quad | \quad (\nu^{\epsilon}\ n)\ \mathsf{in}\ x \triangleleft [x].P_2') & \\
\equiv & l :: ((\nu^{\epsilon}\ n)\ \mathsf{out}\ n.P_1') \quad \| \quad l :: ((\nu^{\epsilon}\ n)\ \mathsf{in}\ x \triangleleft [x].P_2') & \text{(by Clone)} \\
\equiv & (\nu^{\epsilon}\ n)\ l :: (\mathsf{out}\ n.P_1') \quad \| \quad (\nu^{\epsilon}\ n)\ l :: (\mathsf{in}\ x \triangleleft [x].P_2') & \text{(by Res1)} \\
\equiv & (\nu^{\epsilon}\ n)\ l :: (\mathsf{out}\ n.P_1') \quad \| \quad (\nu^{\epsilon}\ m)\ l :: (\mathsf{in}\ x \triangleleft [x].P_2'[n \mapsto m]) & \text{(by Aeq)} \\
\equiv & (\nu^{\epsilon}\ n)\ (\nu^{\epsilon}\ m)\ (l :: (\mathsf{out}\ n.P_1') \quad \| \quad l :: (\mathsf{in}\ x \triangleleft [x].P_2'[n \mapsto m])) & \text{(by Res4)}
\end{array}$$

The processes now agree on the meaning of n, and if we let $P_2'' = P_2'[n \mapsto m]$ then we proceed:

$$\begin{array}{lll}
& (\nu^{\epsilon}\ n)\ (\nu^{\epsilon}\ m)\ (l :: (\mathsf{out}\ n.P_1') \ \| \ l :: (\mathsf{in}\ x \triangleleft [x].P_2'')) & \\
\rightarrow & (\nu^{\epsilon}\ n)\ (\nu^{\epsilon}\ m)\ (l :: P_1' \ \| \ l :: \langle n \rangle \ \| \ l :: (\mathsf{in}\ x \triangleleft [x].P_2'')) & \text{(by New,Par,L-Out)} \\
\rightarrow & (\nu^{\epsilon}\ n)\ (\nu^{\epsilon}\ m)\ (l :: P_1' \ \| \ l :: P_2''[x \mapsto n]) & \text{(by New,Par,L-In)}
\end{array}$$

Establishing the communication we seek.

Example 4.5 (Process Invocation) *Consider the net,*

$$l_A :: A \quad \| \quad l_B :: B$$

and the associated process definitions,

$$\begin{array}{lcl}
A & \triangleq & (\nu^{\epsilon}\ n)\ \mathsf{out}\ n.A \\
B & \triangleq & \mathsf{in}\ x \triangleleft [x]@l_A.\mathsf{out}\ x.B
\end{array}$$

In short, A continuously creates new names, n, and outputs them locally, and B continuously inputs a name from the locality of A, l_A, and outputs it locally. A possible execution of this could be:

$$\begin{array}{lll}
& l_A :: A \ \| \ l_B :: B & \\
\rightarrow & l_A :: ((\nu^{\epsilon}\ n)\ \mathsf{out}\ n.A) \ \| \ l_B :: (\mathsf{in}\ x \triangleleft [x]@l_A.\mathsf{out}\ x.B) & \text{(by Par,Invoc)} \\
\equiv & (\nu^{\epsilon}\ n)\ (l_A :: (\mathsf{out}\ n.A) \ \| \ l_B :: (\mathsf{in}\ x \triangleleft [x]@l_A.\mathsf{out}\ x.B)) & \text{(by Res1,Res4)} \\
\rightarrow & (\nu^{\epsilon}\ n)\ (l_A :: A \ \| \ l_A :: \langle n \rangle \ \| \ l_B :: (\mathsf{in}\ x \triangleleft [x]@l_A.\mathsf{out}\ x.B)) & \text{(by New,Par,L-Out)} \\
\rightarrow & (\nu^{\epsilon}\ n)\ (l_A :: A \ \| \ l_B :: (\mathsf{out}\ n.B)) & \text{(by New,Par,L-In)} \\
\rightarrow & (\nu^{\epsilon}\ n)\ (l_A :: A \ \| \ l_B :: B \ \| \ l_B :: \langle n \rangle) & \text{(by New,Par,L-Out)} \\
\equiv & l_A :: A \ \| \ l_B :: B \ \| \ (\nu^{\epsilon}\ n)\ l_B :: \langle n \rangle & \text{(by Res4)}
\end{array}$$

Whereafter A and B may continue in this manner. Notice that the next time A outputs, it will be a different n, as indicated by the scopes. We should also point out that this is only one possible execution, since the processes are not synchronised, and thus A may output more than once before B inputs.

A trivial, yet important, observation is that the semantics implicitly ensures that name capture through pattern matching does not occur. In other words, if $P \rightarrow P'$ and $n^\tau \notin \mathsf{fn}(P)$ then $n^\tau \notin \mathsf{fn}(P')$. Notice also that the semantics prohibits *malicious* use of α-renaming for illegal pattern matching; e.g. the processes,

$$l :: ((\nu^\epsilon\ n)\ \mathsf{in}\ (n \lhd [\,]).P_1) \quad \| \quad l :: ((\nu^\epsilon\ n)\ \mathsf{out}\ n.P_2)$$

would never synchronise, as the rules would require at least one of the n's to be α-renamed before the scopes could be enlarged and (L-In) attempted applied.

4.4 Well-formedness and Programs

In §4.2 we informally introduced the notion of a program as the combination of a set of localities, L, a net that operates on those localities, N, and a process environment, Δ, mapping all process identifiers to processes. In this section we shall give the formal requirements to such a program, and, in particular, define what it means for a net, N, to be well-formed under the assumption of a set L and a map Δ.

4.4.1 Well-formedness Condition

The well-formedness condition for CryptoKlaim is given in Table 4.10 for processes and Table 4.11 for nets, respectively as judgements of the form $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P$ and $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$. The definitions are fairly simple, given the well-formedness conditions for the cryptographic components, and all that the relations must ensure is that each variable or name that occurs free in the components belong to a containing scope. Note that variable rebinding is allowed, as we simply remove variables being defined in a pattern from X before checking for well-formedness.

4.4.2 Programs

The well-formedness predicate gives us the means to distinguish well-behaved processes. This allows us to formally define the notion of programs.

(WNIL)	$\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} 0$
(WLOUT)	$\dfrac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{out}\, e.P}$
(WOUT)	$\dfrac{l \in \mathsf{L} \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{out}\, e@l.P}$
(WLIN)	$\dfrac{\langle \mathsf{N}, \mathsf{X} \backslash \mathsf{bv}(p) \rangle \vdash p \quad \langle \mathsf{N}, \mathsf{X} \cup \mathsf{bv}(p) \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{in}\, p.P}$
(WIN)	$\dfrac{l \in \mathsf{L} \quad \langle \mathsf{N}, \mathsf{X} \backslash \mathsf{bv}(p) \rangle \vdash p \quad \langle \mathsf{N}, \mathsf{X} \cup \mathsf{bv}(p) \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{in}\, p@l.P}$
(WMATCH)	$\dfrac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash e \quad \langle \mathsf{N}, \mathsf{X} \backslash \mathsf{bv}(p) \rangle \vdash p \quad \langle \mathsf{N}, \mathsf{X} \cup \mathsf{bv}(p) \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} e\, \mathsf{as}\, p.P}$
(WPPAR)	$\dfrac{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P_1 \quad \langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P_2}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P_1 \mid P_2}$
(WINV)	$\dfrac{A[x_1, \ldots, x_k] \triangleq P \quad \langle \emptyset, \{x_1, \ldots, x_k\} \rangle \vdash^{\mathsf{L}}_{\Delta} P \quad \bigwedge_{i=1}^{k} \langle \mathsf{N}, \mathsf{X} \rangle \vdash e_i}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} A(e_1, \ldots, e_k)}$
(WPNEW)	$\dfrac{\langle \mathsf{N} \cup \{n^{\tau} \mid \tau \in T\}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} (\nu^{T}\ n)\ P}$

Table 4.10: Well-formedness of processes; $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P$.

(WNODE)	$\dfrac{l \in \mathsf{L} \quad \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} P}{\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: P}$	(WNPAR)	$\dfrac{\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N_1 \quad \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N_2}{\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N_1 \parallel N_2}$
(WVAL)	$\dfrac{l \in \mathsf{L} \quad \langle \mathsf{N}, \emptyset \rangle \vdash v}{\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \langle v \rangle}$	(WNNEW)	$\dfrac{\mathsf{N} \cup \{n^{\tau} \mid \tau \in T\} \vdash^{\mathsf{L}}_{\Delta} N}{\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} (\nu^{T}\ n)\ N}$

Table 4.11: Well-formedness of nets; $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$.

Definition 4.6 (Programs) *A triple* $\langle \mathsf{L}, \Delta, N \rangle$ *is called a (well-formed) program, written* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$*, if it satisfies the following conditions:*

- *The set of localities,* $\mathsf{L} \subseteq \mathcal{L}$*, is finite.*
- *The process environment,* Δ*, is a finite map, such that* $\mathsf{fn}(\Delta(A)) = \emptyset$ *for each* $A \in dom(\Delta)$*.*
- *The net,* N*, is closed and well-formed:* $\emptyset \vdash^{\mathsf{L}}_{\Delta} N$*.*

Convention 4.7 *In the following we shall assume that all net expressions considered are part of a (well-formed) program.*

4.5 Properties of Programs

The concept of a program allows us to discard nets that do not conform to the assumptions and may exhibit unwanted behaviour. Thus, in the remainder of this dissertation, we shall only concern ourselves with the properties and analysis of programs. First we shall see that the concept of programs is tractable (i.e. that well-formedness is well-defined), and that programs may be composed.

Proposition 4.8 (Well-definedness) *The relation* $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$ *is well-defined.*

Proof The result holds trivially as the relation is compositional. □

It follows that also programs are compositional:

Proposition 4.9 (Compositionality) *If* $\mathsf{Prg}(\mathsf{L}_1, \Delta_1, N_1)$ *and* $\mathsf{Prg}(\mathsf{L}_2, \Delta_2, N_2)$ *where* $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$ *then* $\mathsf{Prg}(\mathsf{L}_1 \cup \mathsf{L}_2, \Delta_1 \cup \Delta_2, N_1 \parallel N_2)$

Proof The result holds vacuously as $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$ is compositional. □

Type Safety. The above results are somewhat trivial and we now turn to a far less trivial result: that the typing of a program is safe under evaluation. Later, when proving properties of the analyses, we shall rely on the well-formedness conditions of programs, and this is only allowed because programs evaluate into programs,

$$\text{If } \mathsf{Prg}(\mathsf{L}, \Delta, N) \text{ and } N \to^* N' \text{ then } \mathsf{Prg}(\mathsf{L}, \Delta, N')$$

We here use the notation $N \to^* N'$ whenever N may evolve into N' in zero or more steps according to the semantics relation $\to$.[4] To see that this is indeed correct, we shall first state some minor results before proceeding to the main result itself.

Lemma 4.10 *If* $\langle \mathsf{N}, \mathsf{X} \rangle \vdash^{\mathsf{L}}_{\Delta} P$ *and* $\langle \mathsf{N}', \emptyset \rangle \vdash v$ *then* $\langle \mathsf{N} \cup \mathsf{N}', \mathsf{X} \backslash \{x\} \rangle \vdash^{\mathsf{L}}_{\Delta} P[x \to v]$.

Proof The result is found by simple structural induction in P, relying on an analogous result for the cryptographic components, also shown straightforwardly. □

Lemma 4.11 *If* $N \equiv N'$ *then* $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$ *if and only if* $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N'$.

Proof The proof proceeds by induction in the shape of the proof tree establishing $\equiv$, where most cases follows directly from the induction hypothesis. Indeed, the rules for the congruence requirements, parallel distribution, and scopes are trivial, recalling that logical conjunction is associative, commutative, and distributive. Finally, the case of α-equivalence is trivial, as α-renaming only changes the naming of bound names. □

Theorem 4.12 (Subject Reduction) *If* $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N$ *and* $N \to N'$ *then* $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} N'$

Proof This is shown by induction on the structure of the inference tree establishing $N \to N'$:

Case (L-Out). Assume $l :: \mathsf{out}\, v.P \to l :: P \parallel l :: \langle v \rangle$ by (L-Out). We then have:

$$\begin{array}{lll}
\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \mathsf{out}\, v.P & & \\
\Leftrightarrow & l \in \mathsf{L} \wedge \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{out}\, v.P & \text{(by Wnode)} \\
\Leftrightarrow & l \in \mathsf{L} \wedge \langle \mathsf{N}, \emptyset \rangle \vdash v \wedge \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} P & \text{(by Wlout)} \\
\Leftrightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: P \parallel l :: \langle v \rangle & \text{(by Wnode,Wval,Wnpar)}
\end{array}$$

Case (Out) is analogous.

[4] Formally $\to^*$ is the reflexive, transitive closure of $\to$.

Case (L-In). Assume $l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle \to l :: P\theta$ by (L-In) because v as $p : \theta$. We then have:

$$
\begin{array}{lll}
& \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle \ \wedge\ v \text{ as } p : \theta & \\
\Leftrightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \mathsf{in}\, p.P \ \wedge\ \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \langle v \rangle \ \wedge\ v \text{ as } p : \theta & \text{(by WNPAR)} \\
\Leftrightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \mathsf{in}\, p.P \ \wedge\ l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash v \ \wedge\ v \text{ as } p : \theta & \text{(by WVAL)} \\
\Rightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: \mathsf{in}\, p.P \ \wedge\ \forall v' \in rg(\theta) : \langle \mathsf{N}, \emptyset \rangle \vdash v' & \text{(by Prop. 3.16)} \\
\Leftrightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} \mathsf{in}\, p.P \ \wedge\ \forall v' \in rg(\theta) : \langle \mathsf{N}, \emptyset \rangle \vdash v' & \text{(by WNODE)} \\
\Rightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash p \ \wedge\ \langle \mathsf{N}, \mathsf{bv}(p) \rangle \vdash^{\mathsf{L}}_{\Delta} P \ \wedge & \text{(by WLIN)} \\
& \forall v' \in rg(\theta) : \langle \mathsf{N}, \emptyset \rangle \vdash v' & \\
\Rightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, dom(\theta) \rangle \vdash^{\mathsf{L}}_{\Delta} P \ \wedge\ \forall v' \in rg(\theta) : \langle \mathsf{N}, \emptyset \rangle \vdash v' & \text{(by Cor. 3.22)} \\
\Rightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} P\theta & \text{(by Lem. 4.10)} \\
\Leftrightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: P\theta & \text{(by WNODE)}
\end{array}
$$

Case (In) and (Match) are analogous.

Case (Invoc). Assume $l :: A(v_1, \ldots, v_k) \to l :: P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ by (Invoc) because $A[x_1, \ldots, x_k] \triangleq P$. We then have:

$$
\begin{array}{lll}
& \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: A(v_1, \ldots, v_k) \ \wedge\ A[x_1, \ldots, x_k] \triangleq P & \\
\Leftrightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} A(v_1, \ldots, v_k) \ \wedge\ A[x_1, \ldots, x_k] \triangleq P & \text{(by WNODE)} \\
\Leftrightarrow & l \in \mathsf{L} \ \wedge\ \langle \emptyset, \{x_1, \ldots, x_k\} \rangle \vdash^{\mathsf{L}}_{\Delta} P \ \wedge\ \bigwedge_{i=1}^{k} \langle \mathsf{N}, \emptyset \rangle \vdash v_i & \text{(by WINV)} \\
\Leftrightarrow & l \in \mathsf{L} \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash^{\mathsf{L}}_{\Delta} P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] & \text{(by Lem. 4.10)} \\
\Leftrightarrow & \mathsf{N} \vdash^{\mathsf{L}}_{\Delta} l :: P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] & \text{(by WNODE)}
\end{array}
$$

Case (New) and (Par) follow by the induction hypothesis.

Case (Struct) is a direct consequence of Lemma 4.11 and the induction hypothesis. That concludes the proof □

Leading to the main result:

Corollary 4.13 (Type Safety) *If* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$ *and* $N \to^* N'$ *then* $\mathsf{Prg}(\mathsf{L}, \Delta, N')$

Proof The result follows by induction of the length of the derivation sequence $N \to^* N'$, where the base case holds by assumption and the inductive step is established by Theorem 4.12. □

It follows that the perfect cryptography guarantee, that we showed for the cryptographic components in §3.3, is guaranteed for any execution of a program.

Corollary 4.14 (Perfect Cryptography) *If* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$ *and* $N \to^* N' \to N''$ *where the reduction* $N' \to N''$ *is due to pattern matching on a pattern* p *then* p *guarantees perfect cryptography.*

Proof From Corollary 4.13 it follows that $\mathsf{Prg}(\mathsf{L}, \Delta, N')$ and as p must be a structural sub-component of N', then the result follows by induction on the structure of the inference tree establishing $N' \to N''$, where each case follows trivially analogous to the proof for Theorem 4.12. □

4.6 Modelling Protocols

In §3.4 we showed how classical protocol narrations can be extended with more information using the cryptographic components. In this section we shall show how these extended narrations may, again, be converted into full protocol specifications in our language.

The translation into CryptoKlaim is quite simple; transmission of a message is modelled with output and reception with input. The structure of the net, however, should be considered, as it is not mentioned at all in the protocol narration. This is important as a wrong abstraction may rule out events that could happen in reality; e.g. if we assumed that two principals were communicating through a safe, private tuple space, although they were using unsafe means in reality, then reality could allow attacks even when the model showed none. To avoid this, we shall be as conservative in our modelling as possible. Each principal is located on separate localities and all inter-locality communication happens through a single tuple space, the *ether*, denoted by the locality $\updownarrow$. This is analogous to the abstraction we suggested in §2.2.1, and is supposed to be a realistic model of the internet; all communication is by default insecure and visible to everyone, and required security must be obtained through cryptography.[5]

Example 4.15 (Wide Mouthed Frog [Example 3.24 cont.]) *Recall the extended protocol narration for the Wide Mouthed Frog protocol of Figure 3.2. To translate this into a CryptoKlaim net, we will need to split the actions up into sequential processes describing each participant. Each sent message corresponds to an output action, and each received message to an input action. The processes are defined straightforwardly as illustrated in Figure 4.1. This is a direct translation of the actions of the principals, and each process definition is recursive to allow multiple protocol runs. Notice that all names that are assumed defined prior to the protocol run, i.e. principal names and long term keys, are passed to the processes through variables to allow sharing of names between the processes.*

[5] Locating the principals on separate localities is in general good coding practice, and convenient when modelling systems where principals use databases or other means of local communication.

$$
\begin{array}{lcl}
A[x^A, x^B, x^{Ka}] & \triangleq & (\nu\ Kab)\ (\nu\ M) \\
& & \mathsf{out}\ \langle x^A, \{\!|\langle x^B, Kab\rangle|\!\}_{x^{Ka}}\rangle @\updownarrow. \\
& & \quad \mathsf{out}\ \{\!|M|\!\}_{Kab} @\updownarrow. \\
& & \quad\quad A(x^A, x^B, x^{Ka}) \\
S[z^A, z^B, z^{Ka}, z^{Kb}] & \triangleq & \mathsf{in}\ \langle z^A, \{\!|\langle z^B, z^{Kab}\rangle|\!\}_{z^{Ka}}\rangle \triangleleft \{z^{Kab}\} @\updownarrow. \\
& & \quad \mathsf{out}\ \{\!|\langle z^A, z^{Kab}\rangle|\!\}_{z^{Kb}} @\updownarrow. \\
& & \quad\quad S(z^A, z^B, z^{Ka}, z^{Kb}) \\
B[y^A, y^{Kb}] & \triangleq & \mathsf{in}\ \{\!|\langle y^A, y^{Kab}\rangle|\!\}_{y^{Kb}} \triangleleft \{y^{Kab}\} @\updownarrow. \\
& & \quad \mathsf{in}\ \{\!|y^M|\!\}_{y^{Kab}} \triangleleft \{y^M\} @\updownarrow. \\
& & \quad\quad B(y^A, y^{Kb})
\end{array}
$$

Figure 4.1: Wide Mouthed Frog processes.

We then define the corresponding net:

$$
\begin{array}{rl}
N_{WMF} = & (\nu\ A)\ (\nu\ B)\ (\nu\ Ka)\ (\nu\ Kb)\ (\\
& \quad l_A :: A(A, B, Ka) \\
& \parallel \quad l_S :: S(A, B, Ka, Kb) \\
& \parallel \quad l_B :: B(A, Kb) \\
&)
\end{array}
$$

Notice how the specification explicitly shows the assumed prior knowledge of the principals; e.g. the invocation $A(A, B, Ka)$ reflects that the principal A is assumed to know the names A and B as well as the symmetric key Ka, prior to protocol execution, and that the well-formedness of the program (with a suitable choice of L) ensures that principles do not exceed their assumed knowledge.

Example 4.16 (Needham-Schroeder Public Key [Example 3.25 cont.]) *The extended protocol narration for the Needham-Schroeder Public Key was given in Figure 3.3. As for the Wide Mouthed Frog protocol, we translate this into a process calculus by splitting the actions up into a process for each participant; one for A and one for B. This results in the following process definitions of Figure 4.2 and the corresponding net definition,*

$$
\begin{array}{rl}
N_{NS} = & \\
& (\nu\ A)\ (\nu\ B)\ (\nu^{\pm}\ Ka)\ (\nu^{\pm}\ Kb)\ (\\
& \quad\quad l_A :: A(A, B, Kb^+, Ka^-) \\
& \quad \parallel \quad l_B :: B(A, B, Kb^-, Ka^+) \\
& \quad)
\end{array}
$$

Again we see the assumptions of prior knowledge, the principals must know their own private key and each other's public key on advance.

$$
\begin{array}{lcl}
A[x^{A}, x^{B}, x^{Kb^{+}}, x^{Ka^{-}}] & \triangleq & (\nu\ Na) \\
 & & \mathsf{out}\ [\langle Na, x^{A}\rangle]_{x^{Kb+}}\ @\updownarrow. \\
 & & \quad \mathsf{in}\ [\![\langle Na, x^{Nb}\rangle]\!]_{x^{Ka-}} \triangleleft [x^{Nb}]@\updownarrow. \\
 & & \quad\quad \mathsf{out}\ [x^{Nb}]_{x^{Kb+}}\ @\updownarrow. \\
 & & \quad\quad\quad A(x^{A}, x^{B}, x^{Kb^{+}}, x^{Ka^{-}}) \\
B[y^{A}, y^{B}, y^{Kb^{-}}, y^{Ka^{+}}] & \triangleq & (\nu\ Nb) \\
 & & \mathsf{in}\ [\![\langle y^{Na}, y^{A}\rangle]\!]_{y^{Kb-}} \triangleleft [y^{Na}]@\updownarrow. \\
 & & \quad \mathsf{out}\ [\langle y^{Na}, Nb\rangle]_{y^{Ka+}}\ @\updownarrow. \\
 & & \quad\quad \mathsf{in}\ [\![Nb]\!]_{y^{Kb-}} \triangleleft [\,]@\updownarrow. \\
 & & \quad\quad\quad B(y^{A}, y^{B}, y^{Kb^{-}}, y^{Ka^{+}})
\end{array}
$$

Figure 4.2: Needham-Schroeder Public Key processes.

4.6.1 Multiple Principals

Cryptographic protocols are usually specified in a manner that only deals with a few participants, one in each role. However, in reality these protocols are applied in systems with many, theoretically infinitely many, participants, and thus any serious analyst should take multiple principals into account. To cater for this we shall assume that a countably infinite set of principals in each role, and consider the situation where any one of them may participate in a communication. Formally, we shall introduce the set of initiators $A_1, A_2, \ldots \in \mathcal{A}$ and responders $B_1, B_2, \ldots \in \mathcal{B}$.

The introduction of multiple principals raises different questions.

Do principals have unique roles in all runs of the protocol?

Assuming $\mathcal{A} = \mathcal{B}$, then principals could be allowed to act as either initiator and responder, and possibly both. Thus, to clarify that principals may not have unique roles in all scenarios, we employ the naming scheme I_i where the index i denotes the current role of the principal; i.e. I_a where $a \in \mathcal{A}$ is a principal that acts as an initiator and, likewise, I_b where $b \in \mathcal{B}$ is a principal that acts as a responder. When principals are allowed to play more than one role, the question arises whether they use the same key in both roles:

If principal I_i uses the key Ka_i for initiating and Kb_i for responding, is $Ka_i = Kb_i$?

This is also an important question, and often the answer depends on the intended use of the protocol, rather than the actual protocol design. In the following we leave our models generic so that we may assume different scenarios later when analysing them.

A naive attempt to model the communication between all these participants would then trivially lead into infinitely large models, which, obviously, is not very attractive. Fortunately, as we discussed in §2.6.2, Comon-Lundh and Cortier proved a result in [39] stating that it is enough to consider only a finite number of participants in each role when investigating for a particular security property, and for integrity and confidentiality this number is 3. Hence, we shall use a finite canonicalisation of the principal sets, $\lfloor \mathcal{A} \rfloor$ and $\lfloor \mathcal{B} \rfloor$, and only consider communication between the canonical representatives. The actual size of these sets is, as already noted, dependent on the security property in question, but for the properties considered in this dissertation 3 is sufficient.

Let us return to the running examples to see how these modifications apply.

Example 4.17 (Wide Mouthed Frog [Example 4.15 cont.]) *We wish to model the scenario where multiple initiators and responders may attempt to establish a key through the same trusted third party. More precisely, the network we are interested in is as follows:*

$$
\begin{aligned}
N^*_{WMF} = \quad & (\nu_{i \in \lfloor \mathcal{A} \rfloor \cup \lfloor \mathcal{B} \rfloor}\ I_i)\ (\nu_{a \in \lfloor \mathcal{A} \rfloor}\ Ka_a)\ (\nu_{b \in \lfloor \mathcal{B} \rfloor}\ Kb_b)\ (\\
& \quad \|_{a \in \lfloor \mathcal{A} \rfloor} \quad l_a :: \left(|_{b \in \lfloor \mathcal{B} \rfloor}\ A_{ab}(I_a, I_b, Ka_a)\right) \\
& \quad \| \qquad\quad l_S :: \left(|_{a \in \lfloor \mathcal{A} \rfloor}\ |_{b \in \lfloor \mathcal{B} \rfloor}\ S_{ab}(I_a, I_b, Ka_a, Kb_b)\right) \\
& \quad \| \ \|_{b \in \lfloor \mathcal{B} \rfloor} \quad l_b :: \left(|_{a \in \lfloor \mathcal{A} \rfloor}\ B_{ab}(I_a, Kb_b)\right) \\
&)
\end{aligned}
$$

Here multiple A's (denoted I_a) may each attempt to establish a key with multiple B's (denoted I_b), through a server S; each attempt is modelled by a process invocation A_{ab}. The server will only allow key establishment between known principals, and thus it has a unique process, S_{ab}, ready to serve any pair of principals that it shares a key with. Finally, each responder B is willing to communicate with any initiator, which is modelled by the process B_{ab}. The subscripted process definitions are as those of Figure 4.1, except that names, variables and scopes are subscripted analogously to the process.

It is important that the names and keys assumed known prior to protocol execution, are scoped around all instances of the protocol, as principals should use the same name and long term key in all protocol executions. Notice that the specification also supports assumptions like $\mathcal{A} = \mathcal{B}$ (implying $\lfloor \mathcal{A} \rfloor = \lfloor \mathcal{B} \rfloor$) or $Ka_i = Kb_i$; in the first case $\lfloor \mathcal{A} \rfloor \cup \lfloor \mathcal{B} \rfloor = \lfloor \mathcal{A} \rfloor = \lfloor \mathcal{B} \rfloor$ and in the latter the scopes $(\nu_{a \in \lfloor \mathcal{A} \rfloor}\ Ka_a)$ are superfluous as they contain no free occurrences of the names.

The observant reader notices that a more simple modelling is possible by simply reusing the same process definition for all arguments, and we should remark why this is not done:

Remark 4.18 *We ensure that the variable parameters in the process invocations are restricted to singleton bindings, by creating a unique process definition for each combination of arguments. This is needed for the context-independent analysis, which we shall present in Chapter 5, to obtain sufficient precision. A more precise and context-dependent analysis, which would not require this extensive encoding, will be discussed in Chapter 5.6.2.*

Example 4.19 (Needham-Schroeder Public Key [Example 4.16 cont.]) *Analogous to the Wide Mouthed Frog protocol, we may extend the modelling of the Needham-Schroeder Public Key protocol to cover multiple initiators and responders.*

$$
\begin{aligned}
&N_{NS^*} = \\
&\quad (\nu_{i \in \lfloor \mathcal{A} \rfloor \cup \lfloor \mathcal{B} \rfloor}\ I_i)\ (\nu^{\pm}_{a \in \lfloor \mathcal{A} \rfloor}\ Ka_a)\ (\nu^{\pm}_{b \in \lfloor \mathcal{B} \rfloor}\ Kb_b)\ (\\
&\qquad \|_{a \in \lfloor \mathcal{A} \rfloor} \quad l_a :: \left(|_{b \in \lfloor \mathcal{B} \rfloor}\ A_{ab}(I_a, I_b, Kb_b^+, Ka_a^-)\right) \\
&\qquad \| \ \|_{b \in \lfloor \mathcal{B} \rfloor} \quad l_b :: \left(|_{a \in \lfloor \mathcal{A} \rfloor}\ B_{ab}(I_a, I_b, Kb_b^-, Ka_a^+)\right) \\
&\quad)
\end{aligned}
$$

Again the model assumes process definitions similar to those of Figure 4.2, except that names, variables and scopes are subscripted analogously to the process.

4.7 Discussion

4.7.1 Differences to cKlaim

The syntax and semantics of CryptoKlaim is based on cKlaim [17], the core of the Klaim-family of calculi. As opposed to CryptoKlaim, however, cKlaim supports dynamically changing systems, and allows construction of new tuples spaces as well as remote evaluation of a process.[6] These operations are not supported by CryptoKlaim, but could pose interesting extensions to the language at a later point.

CryptoKlaim assumes a static set of localities, L, instead of allowing a dynamic creation of new localities under execution as cKlaim (with the construct newloc(l)).

[6] Interestingly, though, cKlaim does not have a notion of local communication and, thus, the execution of a process is independent of the tuple space it is executed in.

While localities are the key primitive in cKlaim, being the only communicated value, they are assumed predefined and known to all participants in CryptoKlaim. This is intended to be a more realistic modelling of communication in networks, as the structure of wired networks rarely change, and practically never during a protocol run. Possibly, however, the dynamic creation of localities could be used for modelling ad-hoc networks and wireless protocols; but it should be carefully considered whether such a modelling would be realistic and sound, before the extension was made, and that is outside the scope of this dissertation.

The evaluation action, $\mathsf{eval}(P)\,@\,l$, from cKlaim is also omitted. This action simply allows a process to be executed in another tuple space, intended to model transaction of executable code. But, obviously, this only makes sense in a security context in collaboration with some access control primitives, which is also beyond the scope of this dissertation. However, we direct the interested reader to the work of Probst et al. in [111], where cKlaim is extended with access control; an extension that should be directly compatible with CryptoKlaim.

Instead CryptoKlaim supports local communication, as denoted by the omission of $@l$ on input and output actions, making process evaluation context sensitive to the locality it is executed in; the same process may behave differently in two different tuple spaces, which is the key ingredient to allow generic process definitions. In fact, a context-dependent analysis, as the one suggested in §5.6.2, could allow the modelling to benefit from this, and result in much more succinct specifications.

4.7.2 Semantical Changes

There are also some more subtle differences between the languages of technical character. First, the output action is not guarded in the semantics by the existence of the targeted locality in CryptoKlaim as it is in cKlaim. This is because the set of localities is static and therefore this requirement can be enforced statically by the well-formedness condition.

Next, processes are invoked in the semantics instead of through the structural congruence as in cKlaim. Process invocation through the structural congruence also allows *process devocation* (i.e. using the process definition from right-to-left) because of symmetry. Thus, a strict enforcement of $\mathsf{fv}(P) = \{x_1, \ldots, x_k\}$ would be required instead of the more common $\subseteq$ for all process definitions $A[x_1, \ldots, x_k] \triangleq P$, as a devocation could otherwise yield unbound variables and cause variable name capture. Additionally, it would not allow the usual subject reduction result for control flow analyses tracking variable bindings (in particular, Lemma 5.9 from Chapter 5 would not hold).

Finally, we require processes in process definition not to contain any free names. This is also to avoid name capture under evaluation and is the simplest solution to the problem; as names may be passed as arguments then there is no reason to allow free names in the processes. Alternative solutions and a more detailed discussion of free names in calculi with recursion may be found in [95].

4.7.3 Distinguishing Names and Variables

Classically, names and variables are considered equivalent and combined into one syntactical class in process calculi. But coming from Horn clauses, where the distinction is always made, it seems an obvious choice to also keep them separate in the language design. Hence, already at this point, prior to the formal construction of an analysis for the language, it should be clear that variables and names in the Horn formulation and in the language design are related.

Apart from the presentational benefits, it also turns out that some of the required complexities surrounding names are not needed for variables; e.g. we do not need to allow free α-renaming for variables. Nevertheless, it is subjective which approach one prefers, and it is sufficient to point out that a similar development with a combined syntactic class could be made.

4.8 Concluding Remarks

This chapter presented CryptoKlaim which is the core of this dissertation. Specifically, we have given the syntax and semantics of the language and defined the requirements to well-formed programs. Using a well-formedness criterion, instead of a more restrictive syntax, allows a flexible calculus and intuitive semantics that maintains attractive properties. In particular, programs are shown to always evaluate into programs, which means that the properties of the well-formed cryptographic components are always guaranteed under execution.

CryptoKlaim is intended as a simple, intuitive language for designing cryptographic systems. Recursive infinite behaviour is encoded by parameterised process invocations. As a result, we see that protocol specifications are cleanly separated into a description of the net model and a number of process definitions. This separation is supposed to ease the design phase even further; it allows a high amount of code reusability and explicitly shows the assumptions of the model. This was demonstrated through the modelling of a few well-known cryptographic protocols.

Chapter 5

Control Flow Analysis

Control flow analysis is a technique from program analysis to statically compute approximations of the result of a program execution. In this chapter we apply this technique to CryptoKlaim to determine the sets of values that may be generated by communication or bound to variables. More precisely, the result of our analysis for a net N yields an over-approximation of

(1) the set of values which each variable may be bound to during evaluation of N, and

(2) the set of values which may be placed on each locality during evaluation of N.

In other words, if the analysis estimate for N states something will not happen then it is guaranteed never to happen during any evaluation of N. Such an analysis is beneficial for establishing security properties of the system. Used in conjunction with a model of possible malicious activity (we shall present one such in §6.1), it provides a static safe approximation of all events that may happen.

The following sections are divided as follows. In §5.1 we present Flow Logic, a notational style for program analysis, which we use in §5.2 to define a control flow analysis of CryptoKlaim. We then show the correctness of the analysis in

§5.3 and how the analysis may be implemented in Horn clauses in §5.4. Finally, in §5.5 we return to our two running examples, and see how the analysis is applied to protocols, before we in §5.6 discuss aspects of the analysis.

5.1 Flow Logic

Flow Logic, as introduced by Nielson, Nielson [91, 98, 100], and with Hankin [93], is a notational style for specifying analyses across programming paradigms. In the Flow Logic framework a clear distinction between the specification of an analysis and the computation of corresponding analysis estimates is made. Essentially, Flow Logic is concerned with the former. By abstracting from domain specific formalisms and instead using standard mathematical notations, the Flow Logic constitutes a meta-language that can present an analysis without requiring additional knowledge about particular formalisms. Deriving an analysis estimate from the resulting analysis specification is then left as a separate activity, usually involving orthogonal considerations and tools.

This approach allows the designer to focus on the specification of analyses without making compromises dictated by implementation considerations. Similarly, implementation is simplified and improved, as the implementor is always free to choose the best available tool.

Acceptability Judgement. A Flow Logic specification defines the relationship between programs, P, and *analysis estimates*, A. This is given by a binary relation,

$$\models:\ (\mathsf{A} \times \mathsf{P}) \to \{\mathsf{true}, \mathsf{false}\}$$

called the acceptability judgement. For a specific program component, $\mathbf{p} \in \mathsf{P}$, and analysis estimate, $\Theta \in \mathsf{A}$, we write $\Theta \models \mathbf{p}$ whenever $\langle \Theta, \mathbf{p} \rangle$ is mapped to true in $\models$. In this case we say that Θ is an acceptable analysis estimate for $\mathbf{p}$. As for the well-formedness relation introduced in §3.1, we may sometimes use subscripts or superscripts on $\models$ to denote static components.

Flow Logic. The judgement is defined by a set of clauses – usually this definition is *syntax-directed* and consists of one clause for each syntactic construct $\diamond$ of P resulting in a compositional judgement relation:

$$\Theta \models \diamond(\cdots \mathbf{p}' \cdots) \quad \textit{iff} \quad \begin{array}{l}\text{some formula } \varphi \text{ containing } \Theta \models \mathbf{p}' \\ \text{for various sub-components } \mathbf{p}'\end{array}$$

Here φ may be a formula of (any fragment of) First Order Logic.

Generally, however, the definition may not be syntax-directed (i.e. the program component on the left-hand-side may be a syntactic sub-component of the formula on the right-hand-side or if just the sides are incomparable), resulting in an abstract specification.

An analysis which has a single, global analysis component Θ, such as shown above, is called *verbose*. Alternatively, it is called *succinct* in the case there is an additional component ψ, local to the expression $\mathbf{p}$ in question and written $\Theta \models \mathbf{p} : \psi$. Occurrences of components ψ on the right-hand side of rules are implicitly existentially quantified. As shown in [100], succinct specifications can always be reduced to verbose ones, but are often preferred due to their conciseness.

5.1.1 Correctness

The correctness of a Flow Logic is determined by establishing the following properties [93, 100]:

(1) Well-definedness

(2) Semantic correctness

(3) Moore-family

Which, in order, are explained below.

Well-definedness. In order to establish well-definedness, note that the rules define an analysis functional $\mathcal{F} : \mathcal{Q} \to \mathcal{Q}$ for a complete lattice $(\mathcal{Q}, \sqsubseteq)$, namely the set of functions $\mathsf{A} \times \mathsf{P} \to \{\mathsf{true}, \mathsf{false}\}$ with the ordering:

$$\mathsf{Q}_1 \sqsubseteq \mathsf{Q}_2 \quad \textit{iff} \quad \forall \langle \Theta, \mathbf{p} \rangle \in \mathsf{A} \times \mathsf{P} : \ \mathsf{Q}_1(\Theta, \mathbf{p}) \Rightarrow \mathsf{Q}_2(\Theta, \mathbf{p})$$

In the case that the specification is compositional, then the least and the greatest fix-point of the functional coincides, resulting in a unique definition of $\models$, adequately established by ordinary induction. But in the general case the specification might be abstract and should be co-inductively interpreted, as we are always interested in the largest relation satisfying the specification (the least restrictive).

Thus, the analysis is called well-defined if the functional $\mathcal{F}$ has a greatest fixed point. This can be guaranteed by showing that $\mathcal{F}$ is monotonic, so that it follows from Tarski's fixed point theorem that it possess a complete lattice of fixed points in $\mathcal{Q}$.

Semantic correctness. Intuitively, an analysis is *semantically correct* if the analysis estimate contains information about the entire evolution of the program, as described by its formal semantics. This is usually expressed through a subject reduction theorem:

$$\text{if } \Theta \models \mathbf{p} \text{ and } \mathbf{p} \rightarrow \mathbf{p}' \text{ then } \Theta \models \mathbf{p}'$$

So if Θ is an acceptable analysis estimate for $\mathbf{p}$, and $\mathbf{p}$ evolves to $\mathbf{p}'$ according to the formal semantics described by relation $\rightarrow$, then Θ is an acceptable analysis estimate for $\mathbf{p}'$ as well.

Moore-family property. It is also reasonable to ask whether each program $\mathbf{p}$ (a) admits an acceptable analysis estimate and (b) has a best analysis estimate. Both questions can be answered by proving the *Moore-family property* (see Appendix A). A subset S of a complete lattice L is called a Moore-family if and only if $\bigsqcap S' \in S$ for all $S' \subseteq S$. The Moore-family property is then stated as follows:

$$\{\Theta \mid \Theta \models \mathbf{p}\} \text{ is a Moore-family for all } \mathbf{p} \in \mathsf{P}$$

By definition, a Moore-family is never empty, which shows (a), and $\bigsqcap\{\Theta \mid \Theta \models \mathbf{p}\}$ is a least solution, showing (b).

5.2 Control Flow Analysis

Using Flow Logic, we shall now define an analysis of the language to assist us in verifying protocols. Recall the verification methodology for Horn clauses we outlined in Chapter 2. Using Horn clauses, we described the protocol in a step-by-step fashion. The predicate, net, represented the network containing all values that *may* be transmitted during or after execution of the protocol. Similarly, by introducing auxiliary predicates, we could extract information about all bindings to specific variables that *may* occur during execution, in order to establish the integrity of the protocol.

The keyword *may* indicates that we are interested in an over-approximation of the possible protocol executions; we want the analysis to tell us everything that could happen. This type of analysis allows us to establish guarantees such as *"x*

can only be bound to the name n*"* or *"the value* $\{m\}_K$ *will never be sent on the ether"*, and, in turn, establish security properties of the protocol.

We also notice that the Horn clause verification of protocols is based on the possible variable bindings and which values that may be sent on the network at any given point. Therefore we can now state the purpose of our analysis. We want to specify an acceptable analysis estimate, ρ, of a program, $\mathsf{Prg}(\mathsf{L}, \Delta, N)$, such that: (1) if the variable x could become bound to a value v during any execution of N then $v \in \rho(x)$; and (2) if the value v could ever exist in a locality l during execution of N then $v \in \rho(l)$. It follows that the analysis estimate, ρ, belongs to:

$$\rho \quad : \quad \mathcal{L} \cup \mathcal{X} \to \wp(\mathsf{Val})$$

The idea is then to specify the analysis in a simple, syntax-directed manner, ensuring that the analysis over-approximates the semantics in all cases. We shall define the analysis in a bottom-up manner, first describing the analysis of pattern matching, then processes, and finally of nets.

Pattern matching. First we must be able to decide whether the matching of v against p may return the substitution θ assuming the analysis estimate ρ. This is captured by the judgement $\rho \models v \ \mathsf{as}\ p : \theta$ as defined in Table 5.1. Most of the rules are simple, we simply require v to structurally matching the pattern and that the subcomponents of v matches the corresponding subcomponents of p. Here the right-hand-sides of the rules are implicitly quantified over the subcomponents of v.[1]

Two rules stand out as interesting though, namely (AVar) and (APDec). In the former, we distinguish between two cases: (1) either the variable x is bound outside the matching (i.e. $x \notin dom(\Gamma)$) and we shall simply require it to belong to the estimate of possible bindings in $\rho(x)$; or (2) it is defined in the matching $x \in dom(\Gamma)$, whereafter we shall require θ to contain this map as well as v to match the decoding of x.

In the case (APDec), we say that at value $v = [\![v_1]\!]_{v_0}$ matches a pattern $[\![d]\!]_e \triangleleft \Gamma$ if there exists a value $\bar{v_0}$ such that $\langle v_0, \bar{v_0} \rangle$ is a key pair and $\bar{v_0}$ and v_1 matches e and d, respectively. The condition that $\langle v_0, \bar{v_0} \rangle$ is a key pair is written as $\langle v_0, \bar{v_0} \rangle \in \mathsf{kp}$. We, thus assume that kp denotes the infinitely enumerable set of key pairs; i.e. if k^+ and k^- is a key pair then $\langle k^+, k^- \rangle, \langle k^-, k^+ \rangle \in \mathsf{kp}$. This is tractable as, although kp is theoretically infinite, it is, for any particular program, always sufficient to only consider the smallest set of pairs containing the names occurring in that program. Hence, as this set is always finite, we may use kp for analysis purposes.

[1] An equivalent, but less intuitive, way of formulating the rules would be to put the structure of v explicitly on the left-hand-side.

(ANAME)	$\rho \models v \text{ as } n^\tau \lhd \Gamma : \theta$	*iff*	$v = n^\tau$
(AVAR)	$\rho \models v \text{ as } x \lhd \Gamma : \theta$	*iff*	$(x \notin dom(\Gamma) \Rightarrow\ v \in \rho(x)) \ \wedge$ $(x \in dom(\Gamma) \Rightarrow$ $\theta(x) = v \ \wedge\ \rho \models v \text{ as } \Gamma(x) \lhd \Gamma : \theta)$
(AETUP)	$\rho \models v \text{ as } \langle e_1, \ldots, e_k \rangle \lhd \Gamma : \theta$	*iff*	$v = \langle v_1, \ldots, v_k \rangle \wedge$ $\bigwedge_{i=1}^{k}\ \rho \models v_i \text{ as } e_i \lhd \Gamma : \theta$
(ASENC)	$\rho \models v \text{ as } \{e_1\}_{e_0} \lhd \Gamma : \theta$	*iff*	$v = \{v_1\}_{v_0} \ \wedge\ \bigwedge_{i=0}^{1}\ \rho \models v_i \text{ as } e_i \lhd \Gamma : \theta$
(APENC)	$\rho \models v \text{ as } [e_1]_{e_0} \lhd \Gamma : \theta$	*iff*	$v = [v_1]_{v_0} \ \wedge\ \bigwedge_{i=0}^{1}\ \rho \models v_i \text{ as } e_i \lhd \Gamma : \theta$
(AWILD)	$\rho \models v \text{ as } * \lhd \Gamma : \theta$	*iff*	true
(ADTUP)	$\rho \models v \text{ as } \langle d_1, \ldots, d_k \rangle \lhd \Gamma : \theta$	*iff*	$v = \langle v_1, \ldots, v_k \rangle \wedge$ $\bigwedge_{i=1}^{k}\ \rho \models v_i \text{ as } d_i \lhd \Gamma : \theta$
(ASDEC)	$\rho \models v \text{ as } \{\!|d|\!\}_e \lhd \Gamma : \theta$	*iff*	$v = \{v_1\}_{v_0} \wedge$ $\rho \models v_0 \text{ as } e \lhd \Gamma : \theta \ \wedge\ \rho \models v_1 \text{ as } d \lhd \Gamma : \theta$
(APDEC)	$\rho \models v \text{ as } [\![d]\!]_e \lhd \Gamma : \theta$	*iff*	$v = [v_1]_{v_0} \ \wedge\ \langle v_0, \bar{v_0} \rangle \in \mathsf{kp} \wedge$ $\rho \models \bar{v_0} \text{ as } e \lhd \Gamma : \theta \ \wedge\ \rho \models v_1 \text{ as } d \lhd \Gamma : \theta$

Table 5.1: Analysis of patterns; $\rho \models v \text{ as } p : \theta$.

Processes. The analysis for processes is specified in Table 5.2. A process must be analysed in the context of a process environment, Δ, and its locality, l, resulting in the judgement relation $\rho \models^l_\Delta P$. Each of the rules defining the analysis is described below.

Rule (ANIL) is trivial, as any analysis estimate is acceptable for the terminated process.

Rule (APPAR) says the analysis of parallel composition holds if it holds for the constituent processes.

Rule (APNEW) shows scopes are simply ignored by the analysis.

Rules (ALOUT) and (AOUT) state the analysis estimate is acceptable for transmission of an encoding e to a locality l (respectively l') if each value, v, produced as a result of replacing all free variables in e by a value from their respective analysis estimate in ρ, is guaranteed to be in $\rho(l)$. Essentially, this is equivalent to requiring:

$$\{v \mid v = e\theta \ \wedge\ \forall x \in dom(\theta) :\ \theta(x) \in \rho(x)\} \subseteq \rho(l)$$

Rules (ALIN) and (AIN) says that the analysis estimate is acceptable for input with a pattern p from a locality l (respectively l') if for all values, v, and

substitutions, θ, where $\rho \models v \text{ as } p : \theta$ then the result of the matching is contained in the analysis estimate; i.e. $\theta(x) \in \rho(\theta)$ whenever x is bound within the matching.

Rule (AMATCH) says that the analysis holds for a matching of an encoding e against a patterns p if there exists some locality, l', for which the analysis correctly models the transmission of e and the reception of p. Practically this can be satisfied by assigning a unique locality, l', to each match construct and simply require the analysis to hold for that particular locality.[2]

Rule (AINV) states the analysis requirements of a process invocation in a straightforward manner; analogously to output, all possible value arguments passed to the process are ensured to be contained in the analysis estimate of the corresponding variables in the process definition, whereafter the analysis is required to also hold for the invoked process counterpart.

Nets. Finally, the analysis specification of nets is straightforward and defined in Table 5.3. It contains no surprises or non-trivial rules.

To see how the analysis works, consider the following example:

Example 5.1 (Analysis of input) *Assume that we wish to analyse the process,*

$$\mathsf{in}\ [\![x]\!]_{k^-} \triangleleft [x].P$$

attempting to input a matching, asymmetrically encrypted value from the local tuple space and bind the variable x.

First, we shall establish the requirements to ρ, a substitution θ and a value v to successfully match against the pattern in the input. These are, according to Table 5.1, as follows:

$$\begin{array}{lll}
\rho \models v \text{ as } [\![x]\!]_{k^-} \triangleleft [x] : \theta & & \\
\Leftrightarrow \quad v = [\![v_1]\!]_{v_0} \ \wedge\ \langle v_0, \bar{v_0}\rangle \in \mathsf{kp} \wedge & & \text{(by APDEC)} \\
\quad \rho \models \bar{v_0} \text{ as } k^- \triangleleft [x] : \theta \ \wedge\ \rho \models v_1 \text{ as } x \triangleleft [x] : \theta & & \\
\Leftrightarrow \quad v = [\![v_1]\!]_{v_0} \ \wedge\ \langle v_0, \bar{v_0}\rangle \in \mathsf{kp} \wedge & & \text{(by ANAME,AVAR,AWILD)} \\
\quad \bar{v_0} = k^- \ \wedge\ \theta(x) = v_1 & & \\
\Leftrightarrow \quad v = [\![v_1]\!]_{v_0} \ \wedge\ \langle v_0, k^-\rangle \in \mathsf{kp} \ \wedge\ \theta(x) = v_1 & & (\star)
\end{array}$$

[2]Note that we do note require l' to be unique but only that it exists. In general, however, unique localities for each matching is preferred, as it achieves the best analysis estimate.

(Anil)	$\rho \models^l_\Delta 0$	*iff*	true
(Appar)	$\rho \models^l_\Delta P_1 \mid P_2$	*iff*	$\rho \models^l_\Delta P_1 \ \wedge\ \rho \models^l_\Delta P_2$
(Apnew)	$\rho \models^l_\Delta (\nu^T\ n)\ P$	*iff*	$\rho \models^l_\Delta P$
(Alout)	$\rho \models^l_\Delta \mathsf{out}\, e.P$		
	iff	$\forall\theta : \bigwedge_{x\in\mathsf{fv}(e)} \theta(x) \in \rho(x) \ \Rightarrow\ e\theta \in \rho(l)$	
		$\wedge\ \rho \models^l_\Delta P$	
(Aout)	$\rho \models^l_\Delta \mathsf{out}\, e@l'.P$		
	iff	$\forall\theta : \bigwedge_{x\in\mathsf{fv}(e)} \theta(x) \in \rho(x) \ \Rightarrow\ e\theta \in \rho(l')$	
		$\wedge\ \rho \models^l_\Delta P$	
(Alin)	$\rho \models^l_\Delta \mathsf{in}\, p.P$		
	iff	$v \in \rho(l) \ \wedge\ \rho \models v\ \mathsf{as}\ p : \theta \ \Rightarrow\ \bigwedge_{x\in\mathsf{bv}(p)} \theta(x) \in \rho(x)$	
		$\wedge\ \rho \models^l_\Delta P$	
(Ain)	$\rho \models^l_\Delta \mathsf{in}\, p@l'.P$		
	iff	$v \in \rho(l') \ \wedge\ \rho \models v\ \mathsf{as}\ p : \theta \ \Rightarrow\ \bigwedge_{x\in\mathsf{bv}(p)} \theta(x) \in \rho(x)$	
		$\wedge\ \rho \models^l_\Delta P$	
(Amatch)	$\rho \models^l_\Delta e\ \mathsf{as}\, p.P$		
	iff	$\exists l' : \rho \models^{l'}_\Delta \mathsf{out}\, e \ \wedge\ \rho \models^{l'}_\Delta \mathsf{in}\, p$	
		$\wedge\ \rho \models^l_\Delta P$	
(Ainv)	$\rho \models^l_\Delta A(e_1, \ldots, e_k)$		
	iff	$A[x_1, \ldots, x_k] \triangleq P \ \Rightarrow$	
		$\bigwedge_{i=1}^k \left(\forall\theta : \bigwedge_{y\in\mathsf{fv}(e_i)} \theta(y) \in \rho(y) \Rightarrow e_i\theta \in \rho(x_i)\right)$	
		$\wedge\ \rho \models^l_\Delta P$	

Table 5.2: Analysis of processes; $\rho \models^l_\Delta P$.

(Anode)	$\rho \models_\Delta l :: P$	*iff*	$\rho \models^l_\Delta P$
(Aval)	$\rho \models_\Delta l :: \langle v \rangle$	*iff*	$v \in \rho(l)$
(Anpar)	$\rho \models_\Delta N_1 \parallel N_2$	*iff*	$\rho \models_\Delta N_1 \ \wedge\ \rho \models_\Delta N_2$
(Annew)	$\rho \models_\Delta (\nu^T\ n)\ N$	*iff*	$\rho \models_\Delta N$

Table 5.3: Analysis of nets; $\rho \models_\Delta N$.

We may then apply the analysis of input to find the analysis constraints:

$$
\begin{array}{lll}
\rho \models \mathsf{in}\ [\![x]\!]_{k^-} \lhd [x].P\ \mathsf{as}\ : & & \\
\Leftrightarrow & v \in \rho(l)\ \wedge\ \rho \models v\ \mathsf{as}\ [\![x]\!]_{k^-} \lhd [x] : \theta\ \Rightarrow & \text{(by ALIN)} \\
& \quad \theta(x) \in \rho(x) & \\
& \wedge\ \rho \models^{l}_{\Delta} P & \\
\Leftrightarrow & [v_1]_{v_0} \in \rho(l)\ \wedge\ \langle k^-, v_0 \rangle \in \mathsf{kp}\ \Rightarrow\ v_1 \in \rho(x) & \text{(by } \star\text{)} \\
& \wedge\ \rho \models^{l}_{\Delta} P &
\end{array}
$$

And we can continue this way on P. The produced constraints are then ready to be fed into a solver.

5.2.1 Handling α-renaming

There is a non-trivial problem in analysing programs with a substitution-based semantics allowing free α-renaming [91]. In short, the analysis can never remain acceptable under evaluation, i.e. the subject reduction result cannot be proven, if the analysis must keep track of possible substitutions. This is because α-renaming allows every single name in the programs to change arbitrarily under evaluation, and, thus, a static analysis estimate containing applied occurrences of names is not tractable.

Environment-based semantics does not encounter this problem, as names are linked to a unique handle in the environment that remains static under evaluation. This allows the analysis to refer to names by their handle and thereby remain acceptable under α-renaming. We shall deal with the problem in a similar manner, and assume a canonical operator $\lfloor \cdot \rfloor : \mathsf{N} \to \lfloor \mathcal{N} \rfloor$ that maps each name to a unique canonical representative, which is preserved under the semantics and in particular under α-renaming. Thus, if we extend $\lfloor \cdot \rfloor$ homomorphically to processes and nets, we have that if $N =_\alpha N'$ then $\lfloor N \rfloor = \lfloor N' \rfloor$.

Remark 5.2 *Tags are invariant under canonicalisation; i.e.* $\lfloor n^\tau \rfloor = \lfloor n \rfloor^\tau$.

This allows us to show that the analysis estimate, with respect to the canonical entities, remains acceptable under evaluation:

$$\textit{If } \rho \models_\Delta \lfloor N \rfloor \textit{ and } N \to N' \textit{ then } \rho \models_\Delta \lfloor N' \rfloor$$

The implications of this abstraction may be hard to comprehend at first, but an easy way to regard it is to assume that the canonical representative is the original name in the initial process, prior to any α-renaming, and that the canonical

operator is simply a method for recalling this name; i.e. if the initial net is denoted $N_\star$ then $\lfloor N_\star \rfloor = N_\star$. The analysis estimate is then only concerned with the initial names, ignoring dynamic renaming required to make the semantics work, and gives the result in terms of the original naming scheme.

Convention 5.3 (Initial programs) *As a matter of convenience we shall also assume that programs,* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$*, subjected to static analysis, are initial in the sense that they satisfy* $\lfloor N \rfloor = N$*.*

5.3 Properties of the Analysis

In this section we will show the correctness of the analysis; i.e., according to §5.1.1 that it has the following properties:

(1) Well-definedness

(2) Semantic correctness

(3) Moore-family property

These are shown below.

5.3.1 Well-Definedness

As described in §5.1, a Flow Logic is well-defined if it has a greatest fixed point.

Theorem 5.4 (Well-Definedness) *The judgement* $\rho \models_\Delta N$ *is well-defined.*

Proof Since the judgement is compositional, a simple structural induction shows that the analysis functional it defines is monotonic. Thus the functional possesses a lattice of fixed points by Tarski's fixed point theorem. Furthermore, the compositional specification entails that the functional has exactly one fixed point so that the least and the greatest fixed points coincide. □

5.3.2 Semantic Correctness

Informally, we may say that semantic correctness expresses that the analysis estimate remains acceptable when the program evolves. Formally this is formulated with respect to the operational semantics of §4.3 as follows:

The proof follows closely that of type safety of the well-formedness condition in §4.5. First we shall show some auxiliary results that ease the presentation of the main result.

Fact 5.5 *$\rho \models v$ as $v' \lhd \Gamma : \theta$ if and only if $v = v'$.*

Proof The result is found by induction in the shape of the proof tree establishing v as $v' \lhd \Gamma : \theta$ where each case holds by assumption and the induction hypothesis. □

Lemma 5.6 *If v as $p : \theta$ and $\langle \mathsf{N}, \emptyset \rangle \vdash p$ then $\rho \models v$ as $p : \theta$ for all ρ.*

Proof The result is found by induction in the shape of the proof tree establishing v as $p : \theta$ where each case holds by assumption and the induction hypothesis. The proof for case (RBIND) also relies on Corollary 3.22 stating that $\mathsf{bv}(p) = dom(\theta)$ and, thus, for each x in p it will be the case that $x \in dom(\Gamma)$. □

Lemma 5.7 *If $\rho \models v$ as $p[x \to v'] : \theta$ and $v' \in \rho(x)$ then $\rho \models v$ as $p : \theta$.*

Proof Recall that if $x \in \mathsf{bv}(p)$ then $p[x \to v'] = p$ in which case the result holds trivially. Thus, the interesting case is when $x \notin \mathsf{bv}(p)$ which is shown by structural induction over patterns by regarding each of the rules in the analysis. Whenever one has to do a proof that concerns substitution the only interesting cases are the ones where the substitution modifies something. In this proof the interesting case, thus, is (AVAR), as e.g.

Case (ANAME). Assume that $\rho \models v$ as $(n^\tau \lhd \Gamma)[x \mapsto v'] : \theta$. By definition of substitution, for arbitrary choices of x and v' it holds that $n^\tau[x \mapsto v'] = n^\tau$ so it is immediate that also $\rho \models v$ as $n^\tau \lhd \Gamma : \theta$.

Case (AVAR). Assume that $\rho \models v$ as $(y \lhd \Gamma)[x \mapsto v'] : \theta$ then there are two

cases: either (1) $x \neq y$ in which case

$$
\begin{array}{lll}
\rho \models v \text{ as } (y \triangleleft \Gamma)[x \mapsto v'] : \theta \ \wedge \ v' \in \rho(x) & & \\
\Leftrightarrow \quad \rho \models v \text{ as } y \triangleleft \Gamma[x \mapsto v'] : \theta \ \wedge \ v' \in \rho(x) & & \text{(by Tab. 3.12)} \\
\Leftrightarrow \quad \theta(y) = v \ \wedge \ v' \in \rho(x) & & \text{(by AVAR)} \\
\qquad \wedge \ y \notin dom(\Gamma) \Rightarrow \ v \in \rho(y) & & \\
\qquad \wedge \ y \in dom(\Gamma) \Rightarrow \ \rho \models v \text{ as } \Gamma(y)[x \mapsto v'] \triangleleft \Gamma[x \mapsto v'] : \theta & & \\
\Rightarrow \quad \theta(y) = v & & \text{(by Ind. Hyp.)} \\
\qquad \wedge \ y \notin dom(\Gamma) \Rightarrow \ v \in \rho(y) & & \\
\qquad \wedge \ y \in dom(\Gamma) \Rightarrow \ \rho \models v \text{ as } \Gamma(y) \triangleleft \Gamma : \theta & & \\
\Leftrightarrow \quad \rho \models v \text{ as } y \triangleleft \Gamma : \theta & & \text{(by AVAR)}
\end{array}
$$

alternatively $x = y$ in which case we have

$$
\begin{array}{lll}
\rho \models v \text{ as } (y \triangleleft \Gamma)[x \mapsto v'] : \theta \ \wedge \ v' \in \rho(x) & & \\
\Leftrightarrow \quad \rho \models v \text{ as } v' \triangleleft \Gamma[x \mapsto v'] : \theta \ \wedge \ v' \in \rho(x) & & \text{(by Tab. 3.12)} \\
\Leftrightarrow \quad v = v' \ \wedge \ v \in \rho(x) & & \text{(by Fact 5.5)} \\
\Leftrightarrow \quad \rho \models v \text{ as } v' \triangleleft \Gamma : \theta \ \wedge \ v \in \rho(x) & & \text{(by Fact 5.5)} \\
\Leftrightarrow \quad \rho \models v \text{ as } x \triangleleft \Gamma : \theta & & \text{(by AVAR)} \\
= \quad \rho \models v \text{ as } y \triangleleft \Gamma : \theta & &
\end{array}
$$

The second-to-last step also uses that $x \notin dom(\Gamma)$. All remaining cases follow directly by the induction hypothesis. □

Lemma 5.8 *If* $\rho \models^l_\Delta P$ *and* $v \in \rho(x)$ *then* $\rho \models^l_\Delta P[x \to v]$.

Proof The proof proceeds by induction in the shape of the proof tree establishing $\rho \models^l_\Delta P$.

Cases (ANIL), (APPAR), (ANEW) hold by assumption and the induction hypothesis.

Case (ALOUT). Assume that $\rho \models^l_\Delta \mathsf{out}\, e.P$ by (ALOUT). The result then holds trivially if $x \notin \mathsf{fv}(e)$ and assuming the opposite, namely that $x \in \mathsf{fv}(e)$, we find

$$
\begin{array}{lll}
\rho \models^l_\Delta \mathsf{out}\, e.P \ \wedge \ v \in \rho(x) & & \\
\Leftrightarrow \quad \forall \theta : \bigwedge_{y \in \mathsf{fv}(e)} \theta(y) \in \rho(y) \ \Rightarrow \ e\theta \in \rho(l) & & \text{(by ALOUT)} \\
\qquad \wedge \ \rho \models^l_\Delta P \ \wedge \ v \in \rho(x) & & \\
\Rightarrow \quad \forall \theta : \bigwedge_{y \in \mathsf{fv}(e) \setminus \{x\}} \theta(y) \in \rho(y) \ \Rightarrow \ e[x \mapsto v]\theta \in \rho(l) & & \\
\qquad \wedge \ \rho \models^l_\Delta P \ \wedge \ v \in \rho(x) & & \\
\Rightarrow \quad \forall \theta : \bigwedge_{y \in \mathsf{fv}(e) \setminus \{x\}} \theta(y) \in \rho(y) \ \Rightarrow \ e[x \mapsto v]\theta \in \rho(l) & & \text{(by Ind. Hyp.)} \\
\qquad \wedge \ \rho \models^l_\Delta P[x \mapsto v] & & \\
\Leftrightarrow \quad \rho \models^l_\Delta (\mathsf{out}\, e.P)[x \mapsto v] & & \text{(by ALOUT)}
\end{array}
$$

Case (AOUT) is analogous.

Case (ALIN). Assume that $\rho \models^{l}_{\Delta} \mathsf{in}\, p.P$ by (ALOUT), we then find:

$$
\begin{array}{lll}
\rho \models^{l}_{\Delta} \mathsf{in}\, p.P \ \wedge\ v \in \rho(x) & & \\
\Leftrightarrow & \forall \theta :\ v' \in \rho(l) \ \wedge\ \rho \models v'\ \mathsf{as}\ p : \theta \ \Rightarrow & \text{(by ALIN)} \\
& \quad \bigwedge_{x \in \mathsf{bv}(p)} \theta(x) \in \rho(x) & \\
& \wedge\ \rho \models^{l}_{\Delta} P \ \wedge\ v \in \rho(x) & \\
\Rightarrow & \forall \theta :\ v' \in \rho(l) \ \wedge\ \rho \models v'\ \mathsf{as}\ p[x \mapsto v] : \theta \ \Rightarrow & \text{(by Lem. 5.7)} \\
& \quad \bigwedge_{x \in \mathsf{bv}(p)} \theta(x) \in \rho(x) & \\
& \wedge\ \rho \models^{l}_{\Delta} P \ \wedge\ v \in \rho(x) & \\
\Rightarrow & \forall \theta :\ v' \in \rho(l) \ \wedge\ \rho \models v'\ \mathsf{as}\ p[x \mapsto v] : \theta \ \Rightarrow & \text{(by Ind. Hyp.)} \\
& \quad \bigwedge_{x \in \mathsf{bv}(p)} \theta(x) \in \rho(x) & \\
& \wedge\ \rho \models^{l}_{\Delta} P[x \mapsto v] & \\
\Leftrightarrow & \rho \models^{l}_{\Delta} (\mathsf{in}\, p.P)[x \mapsto v] & \text{(by ALIN)}
\end{array}
$$

Case (AIN) is analogous.

Case (AMATCH) follows by the induction hypothesis.

Case (AINV) is analogous to (ALOUT). □

Lemma 5.9 *If $N \equiv N'$ then $\rho \models_{\Delta} \lfloor N \rfloor$ if and only if $\rho \models_{\Delta} \lfloor N' \rfloor$.*

Proof The proof proceeds by induction in the shape of the proof tree establishing $\equiv$. The rules for the congruence requirements and parallel distribution follow by the induction hypothesis and the fact that that logical conjunction is associative, commutative, and distributive. For the scoping rules for name bindings the lemma holds vacuously as name restrictions are ignored by the analysis. Finally, the case of α-equivalence relies on the fact that if $N =_{\alpha} N'$ then $\lfloor N \rfloor = \lfloor N' \rfloor$ and, thus, $\rho \models_{\Delta} \lfloor N \rfloor \Leftrightarrow \rho \models_{\Delta} \lfloor N' \rfloor$ follows by referential transparency. □

Theorem 5.10 (Subject Reduction) *If $\mathsf{N} \vdash^{\mathsf{L}}_{\Delta} \lfloor N \rfloor$, $\rho \models_{\Delta} \lfloor N \rfloor$, and $N \to N'$ then $\rho \models_{\Delta} \lfloor N' \rfloor$*

Proof The proof proceeds by induction on the structure of the inference tree establishing $N \to N'$. To retain readability of the proof, the $\lfloor \cdot \rfloor$ operator on all program parts is omitted, as it is only used for case (STRUCT) and otherwise merely passed around.

Case (L-Out). Assume $l :: \mathsf{out}\, v.P \to l :: P \parallel l :: \langle v \rangle$ by (L-Out). We then have:

$$
\begin{array}{lll}
\rho \models_\Delta l :: \mathsf{out}\, v.P & & \\
\Leftrightarrow & \rho \models^l_\Delta \mathsf{out}\, v.P & \text{(by Anode)} \\
\Leftrightarrow & v \in \rho(l) \ \wedge\ \rho \models^l_\Delta P & \text{(by Alout)} \\
\Leftrightarrow & \rho \models_\Delta l :: P \parallel l :: \langle v \rangle & \text{(by Anode,Aval,Wnpar)}
\end{array}
$$

Case (Out) is analogous.

Case (L-In). Assume $l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle \to l :: P\theta$ by (L-In) because $v\ \mathsf{as}\ p : \theta$.

$$
\begin{array}{lll}
\rho \models_\Delta l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle \ \wedge\ v\ \mathsf{as}\ p : \theta & & \\
\Leftrightarrow & \rho \models_\Delta l :: \mathsf{in}\, p.P \ \wedge\ \rho \models_\Delta l :: \langle v \rangle \ \wedge\ v\ \mathsf{as}\ p : \theta & \text{(by Anpar)} \\
\Leftrightarrow & \rho \models_\Delta l :: \mathsf{in}\, p.P \ \wedge\ v \in \rho(l) \ \wedge\ v\ \mathsf{as}\ p : \theta & \text{(by Aval)} \\
\Leftrightarrow & \rho \models^l_\Delta \mathsf{in}\, p.P \ \wedge\ v \in \rho(l) \ \wedge\ v\ \mathsf{as}\ p : \theta & \text{(by Anode)}
\end{array}
$$

By assumption also $\mathsf{N} \vdash^{\mathsf{L}}_\Delta l :: \mathsf{in}\, p.P \parallel l :: \langle v \rangle$ which by (Wnpar), (Wnode), and (Wlin) implies that $\langle \mathsf{N}, \emptyset \rangle \vdash p$, we then have

$$
\begin{array}{lll}
\rho \models^l_\Delta \mathsf{in}\, p.P \ \wedge\ v \in \rho(l) \ \wedge\ v\ \mathsf{as}\ p : \theta \ \wedge\ \langle \mathsf{N}, \emptyset \rangle \vdash p & & \\
\Leftrightarrow & \rho \models^l_\Delta \mathsf{in}\, p.P \ \wedge\ v \in \rho(l) \ \wedge\ \rho \models v\ \mathsf{as}\ p : \theta & \text{(by Lem. 5.6)} \\
\Leftrightarrow & \rho \models^l_\Delta P \ \wedge\ \bigwedge_{x \in \mathsf{bv}(p)} \theta(x) \in \rho(x) & \text{(by Alin)} \\
\Rightarrow & \rho \models^l_\Delta P\theta & \text{(by Lem. 5.8)} \\
\Leftrightarrow & \rho \models_\Delta l :: P\theta & \text{(by Anode)}
\end{array}
$$

The second-to-last step also uses Corollary 3.22; i.e. that $\mathsf{bv}(p) = dom(\theta)$.

Case (In) and (Match) are analogous.

Case (Invoc). Assume $l :: A(v_1, \ldots, v_k) \to l :: P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ by (Invoc) because $A[x_1, \ldots, x_k] \triangleq P$. We then have:

$$
\begin{array}{lll}
\rho \models_\Delta l :: A(v_1, \ldots, v_k) \ \wedge\ A[x_1, \ldots, x_k] \triangleq P & & \\
\Leftrightarrow & \rho \models^l_\Delta A(v_1, \ldots, v_k) \ \wedge\ A[x_1, \ldots, x_k] \triangleq P & \text{(by Anode)} \\
\Leftrightarrow & \bigwedge_{i=1}^k v_i \in \rho(x_i) \ \wedge\ \rho \models^l_\Delta P & \text{(by Ainv)} \\
\Leftrightarrow & \rho \models^l_\Delta P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] & \text{(by Lem. 5.8)} \\
\Leftrightarrow & \rho \models_\Delta l :: P[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] & \text{(by Anode)}
\end{array}
$$

Case (New) and (Par) follow by the induction hypothesis.

Case (Struct) is a direct consequence of Lemma 4.11, Lemma 5.9 and the induction hypothesis. □

Corollary 5.11 (Semantic Correctness) *If* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$, $\rho \models_\Delta \lfloor N \rfloor$, *and* $N \to^* N'$ *then* $\rho \models_\Delta \lfloor N' \rfloor$

Proof The result follows by induction of the length of the derivation sequence $N \to^* N'$, where the base case holds by assumption and the inductive step is established by Theorem 4.12 and Theorem 5.10. □

The semantic correctness result entails that the analysis estimate is invariant with respect to evaluation, but it does not show that the analysis actually achieves its purpose; namely that it may be used for verifying protocols. The usefulness of the analysis is stated by the following two corollaries of the semantic correctness result.

Corollary 5.12 (Values in ρ) *If* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$ *and* $\rho \models_\Delta \lfloor N \rfloor$ *and* $N \to^* N' \to N''$ *where the reduction* $N' \to N''$ *outputs the value* v *to the locality* l *then* $\lfloor v \rfloor \in \rho(l)$.

Proof The result is a direct consequence of Corollary 5.11, (Aval), and the fact that output of a value v to a locality l is reflected syntactically by $l :: \langle v \rangle$, which, thus, must be a structural sub-term of N''. □

Corollary 5.13 (Bindings in ρ) *If* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$ *and* $\rho \models_\Delta \lfloor N \rfloor$ *and* $N \to^* N' \to N''$ *where the reduction* $N' \to N''$ *involves the binding of a variable* x *to a value* v *then* $\lfloor v \rfloor \in \rho(x)$.

Proof That $\rho \models_\Delta \lfloor N' \rfloor$ follows from Corollary 5.11 and then the result is shown analogously to Theorem 5.10 as the binding of x can only be due to either rule (L-In), (In), or (Match). □

5.3.3 Moore-family property

Recall that a subset, S, of a complete lattice, $(L, \sqsubseteq)$, is called a Moore-family if and only if $\forall S' \subseteq S : \bigsqcap S' \in S$. Proving that the set of all acceptable analysis estimates is a Moore-family for all programs enables us to state that there is a *least* analysis estimate for any program, where *least* is to be understood with respect to the partial order $\subseteq$.

As we shall see in §5.4, the right-hand-side of the analysis is, in fact, equivalent to Horn clauses, thus the Moore-family property is a direct consequence of Proposition 2.1. Nevertheless, the result may also be shown by a structural induction on N, by assuming for some index set I that if $\forall i \in I : \rho_i \models_\Delta N$ then also $\bigcap_{i \in I} \rho_i \models_\Delta N$. We shall prove two auxiliary results before proceeding to the main result.

Fact 5.14 *If* $\forall i \in I : \rho_i \models v \text{ as } p : \theta$ *then* $(\bigcap_{i \in I} \rho_i) \models v \text{ as } p : \theta$.

Proof The result follows by straightforward structural induction on v. □

Lemma 5.15 *If* $\forall i \in I : \rho_i \models^l_\Delta P$ *then* $(\bigcap_{i\in I} \rho_i) \models^l_\Delta P$.

Proof This also is shown straightforwardly by structural induction on P, and we will thus only show two exemplary cases of the proof of the auxiliary result.

Case $\mathsf{out}\, e.P$. Assume $\forall i \in I : \rho_i \models^l_\Delta \mathsf{out}\, e.P$. Thus by (ALOUT) we have

$$
\begin{array}{lll}
\forall i \in I : \rho_i \models_\Delta l :: \mathsf{out}\, e.P & & \\
\Leftrightarrow & \forall i \in I : \forall \theta : \bigwedge_{x\in \mathsf{fv}(e)} \theta(x) \in \rho_i(x) \Rightarrow & \text{(by ALOUT)} \\
& \quad e\theta \in \rho_i(l') & \\
& \wedge \forall i \in I : \rho_i \models^l_\Delta P & \\
\Rightarrow & \forall \theta : \bigwedge_{x\in \mathsf{fv}(e)} \theta(x) \in (\bigcap_{i\in I} \rho_i)(x) \Rightarrow & \text{(by Ind. Hyp.)} \\
& \quad e\theta \in (\bigcap_{i\in I} \rho_i)(l') & \\
& \wedge (\bigcap_{i\in I} \rho_i) \models^l_\Delta P & \\
\Leftrightarrow & (\bigcap_{i\in I} \rho_i) \models^l_\Delta \mathsf{out}\, e.P & \text{(by ALOUT)}
\end{array}
$$

Case $\mathsf{in}\, p.P$. Assume $\forall i \in I : \rho_i \models^l_\Delta \mathsf{in}\, p.P$. Thus we have

$$
\begin{array}{lll}
\forall i \in I : \rho_i \models^l_\Delta \mathsf{in}\, p.P & & \\
\Leftrightarrow & \forall i \in I : \forall \theta : v \in \rho_i(l) \wedge \rho_i \models v \;\mathsf{as}\; p : \theta \Rightarrow & \text{(by ALIN)} \\
& \quad \bigwedge_{x\in \mathsf{bv}(p)} \theta(x) \in \rho_i(x) & \\
& \wedge \rho \models^l_\Delta P & \\
\Rightarrow & \forall \theta : v \in (\bigcap_{i\in I} \rho_i)(l) \wedge (\bigcap_{i\in I} \rho_i) \models v \;\mathsf{as}\; p : \theta \Rightarrow & \text{(by Fact 5.14)} \\
& \quad \bigwedge_{x\in \mathsf{bv}(p)} \theta(x) \in (\bigcap_{i\in I} \rho_i)(x) & \\
& \wedge \rho \models^l_\Delta P & \\
\Rightarrow & \forall \theta : v \in (\bigcap_{i\in I} \rho_i)(l) \wedge (\bigcap_{i\in I} \rho_i) \models v \;\mathsf{as}\; p : \theta \Rightarrow & \text{(by Ind. Hyp.)} \\
& \quad \bigwedge_{x\in \mathsf{bv}(p)} \theta(x) \in (\bigcap_{i\in I} \rho_i)(x) & \\
& \wedge (\bigcap_{i\in I} \rho_i) \models^l_\Delta P & \\
\Leftrightarrow & (\bigcap_{i\in I} \rho_i) \models^l_\Delta \mathsf{in}\, e.P & \text{(by ALIN)}
\end{array}
$$

The remaining cases follow either by the induction hypothesis or in an analogous fashion to one of the cases above. □

This allows us to show the main result.

Theorem 5.16 (Moore-family) *The set* $\{\rho \mid \rho \models_\Delta N\}$ *is a Moore-family for all* $\mathsf{Prg}(\mathsf{L}, \Delta, N)$.

Proof The result follows trivially by structural induction relying on Lemma 5.15. □

5.4 Implementation

The control flow analysis is formulated such as to make the transition from semantics to analysis as gentle as possible. However, as the language design was analysis-directed, we should expect that implementation of the analysis is straightforward. Fortunately this is the case, in fact, the right-hand-side of the analysis is basically already in Horn clauses, although it may require some rewriting to realise this.

First, analogous to the approach of §2.2, we see that an encoding, e, always have a corresponding term, t_e; i.e. each constructor in the cryptographic components is simply modelled by a unique constructor of the same arity, names are nullary constructors, and variables are variables.

Next, each implication produced by the analysis corresponds to one or more Horn clauses. This is seen as follows:

- Horn clauses are, similarly to the right-hand-side of the analysis, implicitly quantified over substitutions.
- In Horn clauses the domain of the model, ρ, is predicate names. Hence, we shall introduce a unique predicate name for each variable for each locality. This obtains the one-to-one correspondence between the ρ's; e.g. $e\theta \in \rho(x) \Leftrightarrow \rho, \theta \models x(t_e)$. Note that this is tractable as any program will only use a finite set of variables and localities.
- Introduce a predicate kp such that $\rho(\mathsf{kp})$ is the set of all key pairs, again this is tractable as only a finite number of key pairs will occur in any program.
- For each encoding, which is (implicitly) quantified over values, we replace the unspecified values by corresponding fresh variables and leave the quantification to the substitution; e.g. $\lfloor v_1 \rfloor_{v_0} \in \rho(l) \Leftrightarrow \forall \theta : (\lfloor x_1 \rfloor_{x_0})\,\theta \in \rho(l) \Leftrightarrow \rho, \theta \models l(\lfloor x_1 \rfloor_{x_0})$.
- Clauses with more than one literal in the conclusion are split up; i.e. $(\omega \Rightarrow g_1 \wedge \cdots \wedge g_k) \Leftrightarrow \bigwedge_{i=1}^{k} (\omega \Rightarrow g_i)$

It follows that each right-hand-side rule for nets and processes may be translated into equivalent Horn clauses.[3]

[3] A very compact Horn formulation can be obtained by encoding families of names through arguments of the constructors and leave the instantiation to the iterative framework; e.g for $|_{i \in I} P_i$ one may use a unary constructor $n(I)$ for each n_i occurring in P_i, add the literal $\mathsf{I}(I)$ to each precondition, and a fact $\mathsf{I}(i)$ for each $i \in I$ to the formula.

Example 5.17 (Analysis of Input [Example 5.1 cont.]) *Recall that the analysis of the process,*

$$\mathsf{in}\ [\![x]\!]_{k^-} \triangleleft [x].P$$

required solving the constraint:

$$\begin{array}{ll} \forall\theta:\ [v_1]_{v_0} \in \rho(l)\ \wedge\ \langle k^-, v_0\rangle \in \mathsf{kp}\ \Rightarrow\ v_1 \in \rho(x) & \\ \Leftrightarrow & \forall\theta:\ ([x_1]_{x_0})\,\theta \in \rho(l)\ \wedge\ \langle k^-, x_0\rangle\theta \in \mathsf{kp}\ \Rightarrow\ x_1 \in \rho(x) \\ \Leftrightarrow & \forall\theta:\ \rho,\theta \models l([X_1]_{X_0})\ \wedge\ \rho,\theta \models \mathsf{kp}(k^-, X_0)\ \Rightarrow\ \rho,\theta \models x(X_1) \\ \Leftrightarrow & \rho \models \big(x(X_1)\ \Leftarrow\ l([X_1]_{X_0}) \wedge \mathsf{kp}(k^-, X_0)\big) \end{array}$$

In other words, equivalent to the Horn clause,

$$x(X_1)\ \Leftarrow\ l([X_1]_{X_0}) \wedge \mathsf{kp}(k^-, X_0)$$

which, in conjunction with the analysis of P, may be solved using the techniques presented in Chapter 2.

5.5 Examples

We now ready to establish the constraints specifying an acceptable analysis estimate for each of the running examples.

Example 5.18 (Wide Mouthed Frog [Example 4.15 cont.]) *The analysis constraints for the protocol, $\rho \models_\Delta N_{WMF}$, from Figure 4.1, amounts to the conjunction of the analysis constraints for the three process invocations:*

$$\rho \models^{l_A}_\Delta A(A, B, Ka)\ \wedge\ \rho \models^{l_S}_\Delta S(A, B, Ka, Kb)\ \wedge\ \rho \models^{l_B}_\Delta B(A, Kb)$$

Each of which is shown in Figure 5.1 alongside their corresponding, fully expanded logical constraints. Finding the least analysis estimate satisfying the conjunction of these constraints, thus, corresponds finding the least model ρ satisfying Horn clauses listed in Figure 5.2.[4]

Example 5.19 (Needham-Schroeder Public Key[Example 4.16 cont.]) *Similarly, the analysis constraints for the protocol, $\rho \models_\Delta N_{NS}$, from Figure 4.2, amounts to the conjunction of the analysis constraints for the two process invocations:*

$$\rho \models^{l_A}_\Delta A(A, B, Kb^+, Ka^-)\ \wedge\ \rho \models^{l_B}_\Delta B(A, B, Kb^-, Ka^+)$$

The fully expanded right-hand-side counterpart of the analysis is shown in Figure 5.3, and the corresponding Horn clauses in Figure 5.4. The Horn clause formulation also includes the key pair sets occurring in the protocol. Notice how the analysis ignores the last integrity check at B in the protocol, as part of the over-approximation, as this check does not introduce new variable bindings.

[4]Note that we use a variable naming scheme similar to that of Chapter 2 to make the analysis constraints more intelligible.

$$
\begin{array}{l}
\rho \models^{\updownarrow}_{\Delta} A(A, B, Ka) \\
\quad \textit{iff} \quad A \in \rho(x^{A}) \ \wedge \ B \in \rho(x^{B}) \ \wedge \ Ka \in \rho(x^{Ka}) \\
\qquad \wedge \\
\qquad \forall \theta : \ \theta(x^{A}) \in \rho(x^{A}) \ \wedge \ \theta(x^{B}) \in \rho(x^{B}) \ \wedge \ \theta(x^{Ka}) \in \rho(x^{Ka}) \Rightarrow \\
\qquad\quad \langle x^{A}, \{x^{B}, Kab\}_{x^{Ka}} \rangle \theta \in \rho(\updownarrow) \\
\qquad \wedge \\
\qquad \{M\}_{Kab} \in \rho(\updownarrow) \\
\rho \models^{\updownarrow}_{\Delta} S(A, B, Ka, Kb) \\
\quad \textit{iff} \quad A \in \rho(z^{A}) \ \wedge \ B \in \rho(z^{B}) \ \wedge \ Ka \in \rho(z^{Ka}) \ \wedge \ Kb \in \rho(z^{Kb}) \\
\qquad \wedge \\
\qquad \langle v^{A}, \{\langle v^{B}, v^{Kab} \rangle\}_{v^{Kb}} \rangle \in \rho(\updownarrow) \ \wedge \ v^{A} \in \rho(z^{A}) \ \wedge \ v^{B} \in \rho(z^{B}) \ \wedge \ v^{Kb} \in \rho(z^{Kb}) \Rightarrow \\
\qquad\quad v^{Kab} \in \rho(z^{Kab}) \\
\qquad \wedge \\
\qquad \forall \theta : \ \theta(z^{A}) \in \rho(z^{A}) \ \wedge \ \theta(z^{Kab}) \in \rho(z^{Kab}) \ \wedge \ \theta(z^{Kb}) \in \rho(z^{Kb}) \Rightarrow \\
\qquad\quad \left(\{\langle z^{A}, z^{Kab} \rangle\}_{x^{Kb}}\right) \theta \in \rho(\updownarrow) \\
\rho \models^{\updownarrow}_{\Delta} B(A, Kb) \\
\quad \textit{iff} \quad A \in \rho(y^{A}) \ \wedge \ Kb \in \rho(y^{Kb}) \\
\qquad \wedge \\
\qquad \{\langle v^{A}, v^{Kab} \rangle\}_{v^{Kb}} \in \rho(\updownarrow) \ \wedge \ v^{A} \in \rho(y^{A}) \ \wedge \ v^{Kb} \in \rho(y^{Kb}) \Rightarrow \\
\qquad\quad v^{Kab} \in \rho(y^{Kab}) \\
\qquad \wedge \\
\qquad \{v^{M}\}_{v^{Kab}} \in \rho(\updownarrow) \ \wedge \ v^{Kab} \in \rho(y^{Kab}) \Rightarrow \\
\qquad\quad v^{M} \in \rho(y^{M})
\end{array}
$$

Figure 5.1: Analysis of Wide Mouthed Frog; $\rho \models_{\Delta} N_{WMF}$

$$
\begin{array}{l}
x^{A}(A) \ \wedge \ x^{B}(B) \ \wedge \ x^{Ka}(Ka) \\
\updownarrow(\langle X^{A}, \{X^{B}, Kab\}_{X^{Ka}} \rangle) \Leftarrow x^{A}(X^{A}) \wedge x^{B}(X^{B}) \wedge x^{Ka}(X^{Ka}) \\
\updownarrow(\{M\}_{Kab}) \\
z^{A}(A) \ \wedge \ z^{B}(B) \ \wedge \ z^{Ka}(Ka) \ \wedge \ z^{Kb}(Kb) \\
z^{Kab}(Z^{Kab}) \Leftarrow \updownarrow(\langle Z^{A}, \{\langle Z^{B}, Z^{Kab} \rangle\}_{Z^{Ka}} \rangle) \ \wedge \ z^{A}(Z^{A}) \ \wedge \ z^{B}(Z^{B}) \ \wedge \ z^{Ka}(Z^{Ka}) \\
\updownarrow(\{\langle Z^{A}, Z^{Kab} \rangle\}_{Z^{Kb}}) \Leftarrow z^{A}(Z^{A}) \ \wedge \ z^{Kab}(Z^{Kab}) \ \wedge \ z^{Kb}(Z^{Kb}) \\
y^{A}(A) \ \wedge \ y^{Kb}(Kb) \\
y^{Kab}(Y^{Kab}) \Leftarrow \updownarrow(\{\langle Y^{A}, Y^{Kab} \rangle\}_{Y^{Kb}}) \ \wedge \ y^{A}(Y^{A}) \ \wedge \ y^{Kb}(Y^{Kb}) \\
y^{M}(Y^{M}) \Leftarrow \updownarrow(\{Y^{M}\}_{Y^{Kab}}) \ \wedge \ y^{Kab}(Y^{Kab})
\end{array}
$$

Figure 5.2: Analysis of Wide Mouthed Frog in Horn clauses.

$$
\begin{array}{l}
\rho \models^{\updownarrow}_{\Delta} A(A, B, Kb^{+}, Ka^{-}) \\
\quad \mathit{iff} \quad A \in \rho(x^{A}) \ \wedge \ B \in \rho(x^{B}) \ \wedge \ Kb^{+} \in \rho(x^{Kb^{+}}) \ \wedge \ Ka^{-} \in \rho(x^{Ka^{-}}) \\
\qquad \wedge \\
\qquad \forall \theta : \ \theta(x^{Kb^{+}}) \in \rho(x^{Kb^{+}}) \ \wedge \ \theta(x^{A}) \in \rho(x^{A}) \Rightarrow \\
\qquad\qquad \left([\langle Na, x^{A} \rangle]_{x^{Kb^{+}}}\right) \theta \in \rho(\updownarrow) \\
\qquad \wedge \\
\qquad [\langle v^{Na}, v^{Nb} \rangle]_{v} \in \rho(\updownarrow) \ \wedge \ \langle v, v^{Ka^{-}} \rangle \in \mathsf{kp} \ \wedge \ v^{Ka^{-}} \in \rho(x^{Ka^{-}}) \ \wedge \ v^{Na} \in \rho(x^{Na}) \Rightarrow \\
\qquad\qquad v^{Nb} \in \rho(x^{Nb}) \\
\qquad \wedge \\
\qquad \forall \theta : \ \theta(x^{Kb^{+}}) \in \rho(x^{Kb^{+}}) \ \wedge \ \theta(x^{Nb}) \in \rho(x^{Nb}) \Rightarrow \\
\qquad\qquad \left([x^{Nb}]_{x^{Kb^{+}}}\right) \theta \in \rho(\updownarrow) \\
\rho \models^{\updownarrow}_{\Delta} B(A, B, Kb^{-}, Ka^{+}) \\
\quad \mathit{iff} \quad A \in \rho(y^{A}) \ \wedge \ B \in \rho(y^{B}) \ \wedge \ Kb^{-} \in \rho(y^{Kb^{-}}) \ \wedge \ Ka^{+} \in \rho(y^{Ka^{+}}) \\
\qquad \wedge \\
\qquad [\langle v^{Na}, v^{A} \rangle]_{v} \in \rho(\updownarrow) \ \wedge \ \langle v, v^{Kb^{-}} \rangle \in \mathsf{kp} \ \wedge \ v^{Kb^{-}} \in \rho(y^{Kb^{-}}) \ \wedge \ v^{A} \in \rho(y^{A}) \Rightarrow \\
\qquad\qquad v^{Na} \in \rho(y^{Na}) \\
\qquad \wedge \\
\qquad \forall \theta : \ \theta(y^{Ka^{+}}) \in \rho(y^{KA^{+}}) \ \wedge \ \theta(y^{Na}) \in \rho(y^{Na}) \Rightarrow \\
\qquad\qquad \left([\langle y^{Na}, NB \rangle]_{y^{Ka^{+}}}\right) \theta \in \rho(\updownarrow)
\end{array}
$$

Figure 5.3: Analysis of Needham-Schroeder Public Key; $\rho \models_{\Delta} N_{NS}$

$$
\begin{array}{l}
\mathsf{kp}(Ka^{+}, Ka^{-}) \ \wedge \ \mathsf{kp}(Ka^{-}, Ka^{+}) \ \wedge \ \mathsf{kp}(Kb^{+}, Kb^{-}) \ \wedge \ \mathsf{kp}(Kb^{-}, Kb^{+}) \\
\\
x^{A}(A) \ \wedge \ x^{B}(B) \ \wedge \ x^{Kb^{+}}(Kb^{+}) \ \wedge \ x^{Ka^{-}}(Ka^{-}) \\
\updownarrow([\langle Na, X^{A} \rangle]_{X^{Kb^{+}}}) \Leftarrow x^{Kb^{+}}(X^{Kb^{+}}) \wedge x^{A}(X^{A}) \\
x^{Nb}(X^{Nb}) \Leftarrow \updownarrow([\langle X^{Na}, X^{Nb} \rangle]_{X}) \ \wedge \ \mathsf{kp}(X, X^{Ka^{-}}) \ \wedge \ x^{Ka^{-}}(X^{Ka^{-}}) \ \wedge \ x^{Na}(X^{Na}) \\
\updownarrow([X^{Nb}]_{X^{Kb^{+}}}) \Leftarrow x^{Kb^{+}}(X^{Kb^{+}}) \wedge x^{Nb}(X^{Nb}) \\
\\
y^{A}(A) \ \wedge \ y^{B}(B) \ \wedge \ y^{Kb^{-}}(Kb^{-}) \ \wedge \ y^{Ka^{+}}(Ka^{+}) \\
y^{Na}(Y^{Na}) \Leftarrow \updownarrow([\langle Y^{Na}, Y^{A} \rangle]_{Y}) \ \wedge \ \mathsf{kp}(Y, Y^{Kb^{-}}) \ \wedge \ y^{Kb^{-}}(Y^{Kb^{-}}) \ \wedge \ y^{Na}(Y^{Na}) \\
\updownarrow([\langle Y^{Na}, Nb \rangle]_{Y^{Ka-}}) \Leftarrow y^{Ka^{-}}(Y^{Ka^{-}}) \wedge y^{Na}(Y^{Na})
\end{array}
$$

Figure 5.4: Analysis of Needham-Schroeder Public Key in Horn clauses.

5.6 Discussion

In this chapter we have presented a control flow analysis for CryptoKlaim. The analysis serves to show that the language can be easily verified. Indeed, as we shall see in the following chapter, we may apply the analysis successfully to several well-known protocols; specifically we shall pin-point the known attacks without getting any false positives and even verify an industrial case study.

The main topic of this dissertation is, however, the construction of the language and not the analysis. Thus, to avoid too much digression in the presentation, we have opted for a simple, yet powerful analysis, to establish proof-of-concept. This section addresses some of the topics that were left out during the presentation.

5.6.1 Complexity

There is an almost one-to-one correspondence between the representation in the language and the number of Horn clauses in the analysis. Each output action translates into exactly one Horn clause and each input action corresponds to a number of Horn clauses equal to the number of variables bound in the action.

The complexity of solving the Horn clauses depends on the solving technique. If this is the $\mathcal{H}_1$-iterative framework of Chapter 2, then the complexity of each solving operation is, as already mentioned, deterministic exponential in the number of clauses. Furthermore, each refinement step exponentially increases the number of clauses. This, however, is a crude approximation of the worst case complexity, and, as discussed in Appendix B.2, there are techniques for reducing the latter. In fact, the protocols are specified in a way where each variable corresponds to a single ground term, resulting in a linear instantiation.

The theoretical exponential complexity is a result of the complexity of protocol analysis in general, and the fact that we seek to retain as much information throughout our abstraction as possible. Since protocols are usually only a few lines of code, even industrial sized systems would be unlikely to exceed 100 lines of code, the theoretical complexity is of little importance for these cases.

In general, however, language-based techniques and analyses are used on programs consisting of several million lines of code. For such uses, the $\mathcal{H}_1$-iterative framework is much too expensive, and fortunately there are ways to reduce the complexity – of course at the cost of precision.

Consider that we wanted to optimise the analysis presented in this chapter. The

high cost of verification can, in fact, be narrowed down to two specific problems: (1) non-linear output actions and (2) non-linear input actions. Here non-linear refers to encodings or decodings where at least one variable occurs more than once. Non-linear outputs result in Horn clauses with non-linear conclusions, which are approximated by the relaxation, whereafter they are, possibly, instantiated by the iterative framework. A simple way to reduce complexity is therefore to simply apply the relaxation suggested in §2.4 and then use the normalisation only once, instead of the iterative framework. Non-linear inputs can be handled in much the same way, by treating the multiple occurrences of variables separately. With this approach, the resulting clauses would be guaranteed to have linear conclusions and each variable would have a non-cyclical dependance in the precondition. This class of Horn clauses is called $\mathcal{H}_3$ and is proven to be normalisable in cubic time [96].

5.6.2 History-based Analyses

Presently, concurrent processes and nets are analysed without information about context or execution history. This is evident, as if a value may eventually be transmitted in a locality then we assume that it is always present on that locality. Secondly, if a variable could be bound to a value in any execution of the program then we analyse all executions as if the variable had have been bound to that value. Below we discuss how the analysis could be improved in these respects.

Relational analysis. The first assumption means that the analysis cannot distinguish between values being sent only once and values sent repeatedly. This is an over-approximation and it follows that the analysis cannot be used for establishing *reachability* results (i.e. whether particular actions can always be executed successfully) or, specifically, establishing the security property availability. Later, in Chapter 6, we shall introduce a *strongest attacker* that is assumed present during all protocol executions. This is supposed to allow verification of the protocols in a worst case scenario, but in the context of such an attacker, who, among other things, can duplicate values on the localities it has access to, the approximation seems adequate. Furthermore, as protocols are usually designed to obtain something *good* all program parts are usually trivially reachable.

Nevertheless, there are still systems for which a more fine-grained analysis would be interesting.[5] One solution is to make the analysis relational and associate each

[5]Electronic payment or voting systems are good examples of systems where users have limited, finite execution rights.

value on each locality with an element of a finite lattice indicating its quantity (e.g., distinguishing between zero, one, or multiple occurrences, an element from the set $\{0, 1, \infty\}$). The term *relational* denotes that the analysis is obtained by relating two or more individual analyses. Informally, we say that a relational analysis records "sets of tuples" instead of "tuples of sets" and it should be noted that the complexity increases accordingly. The useability of relational analyses is well-known from abstract interpretation (e.g. in pointer analysis) and a good introduction to the subject is given in [93].

Flow-dependent analysis. As it is, the current analysis has a single, global analysis estimate which must be acceptable for the entire program execution. It follows that the analysis cannot differentiate between when in the program's life cycle something happens; in other words, it is flow-independent. Alternatively the analysis could be made flow-dependent, and take the order in which actions occurs into account.

The flow of a program could be found by decorating each action with a unique annotation (such as $a^i.P$ where $i \in \mathbb{N}$) and then use a simple analysis to establish the possible execution order of the actions; i.e. finding the set Flow such that $\langle i, j\rangle \in \mathsf{Flow}$ means that action j may be executed right after i). The analysis estimate could then be extended accordingly, creating an instance for each annotation, written $\rho[i]$. Using these ingredients an analysis for local output could then be formulated as:

$$\rho \models^{l}_{\Delta} \mathsf{out}^i\ e.\ P \qquad \textit{iff} \qquad \begin{array}{l} \forall j : \langle i, j\rangle \in \mathsf{Flow} \ \Rightarrow \\ \quad \forall \theta : \bigwedge_{x \in \mathsf{fv}(e)} \theta(x) \in \rho(x) \ \Rightarrow\ e\theta \in \rho[j](l) \\ \quad \wedge\ \rho[i] \subseteq \rho[j] \end{array}$$

All other rules should be written in a similar manner. It is important to stress that the set Flow may be relatively large, as it must take interleaving of concurrent processes into account. Nevertheless, it is still finite and can be calculated on advance. It follows that the analysis is directly translatable to Horn clauses, and, obviously that the modification increases the complexity in the size of the number of program points.

Note that the flow-dependent analysis of input must still retain the value in the locality estimate, as a value in the analysis estimate of a locality represents "one or more". Therefore, as protocol participants usually exhibit infinite behaviour, the flow becomes superfluous and is only interesting in conjunction with a relational analysis, such as the one suggested above. For examples on and techniques for flow-dependent analyses in Flow Logic, see [101, 108], and for combination with a relational analysis, see [99, 107, 109].

Context-dependent analysis. The last, but probably the most interesting, extension to the analysis is context-dependent. The analysis estimate collects all possible value-bindings to variables but does not remember the context that the binding occurred. If multiple variables are bound simultaneously, such as in the action $\mathsf{in}\,\langle x, y\rangle \triangleleft [x, y]$, then the analysis estimate does not relate these bindings. Consider, for example, that the pattern above could be matched against either $\langle a, b\rangle$ or $\langle b, a\rangle$ then the analysis estimate would yield $\rho(x) \times \rho(y) = \{a, b\} \times \{a, b\}$.

Alternatively, the analysis could be modified, such that variable bindings were recorded in tuples according to the patterns they occurred in; in particular, the analysis estimate of the above-mentioned pattern would include $\rho(x, y) = \{\langle a, b\rangle, \langle b, a\rangle\}$. This approach is closer to the intuitive one we constructed in Chapter 2. It is also a highly attractive improvement, as it would allow the analysis to deal with reuse of process definitions, such that the extensive encoding mentioned in Remark 4.18 would not be necessary. However, the improvement comes at a cost, and, in addition to the obvious increase in complexity in the size of the cartesian products, it results in that the expected subject reduction result (Theorem 5.10) no longer holds [92]. The correctness of the analysis can still be shown, but it is more demanding and does not belong in a proof-of-concept chapter.

5.6.3 Canonical Names

The canonical name approach is a solution to a well-known problem pointed out in [91], where it is shown that analyses of languages with substitution-based semantics and free α-renaming cannot be defined straightforwardly in a sound, tractable manner. This result was surprising as other semantics did not cause the problem, despite the close resemblance in the definitions. As already mentioned, the solution is to introduce an operator that fixes all names to a static entity, and focus the analysis around the static entities. This solution is, in fact, completely analogous to the static entities that are given directly by the alternative semantics formulations, but it allows us to use the, arguably, much nicer substitution-based semantics.

Since the discovery of canonicalisation technique, various formulations for it have surfaced. In [26, 90] the canonical representatives are ensured preserved under α-renaming, by using a weaker α-renaming, called *disciplined α-renaming* that retains the canonical representatives. The formulation adopted in this dissertation, which is also used in [102], differs from that approach in that the subject reduction result is instead shown with respect to the original program and naming scheme. This allows the usual and expected definition of α-renaming and postpones the technicalities to the proof-burden. Nevertheless, the approaches

are essentially equivalent and differ only in the presentation.

It should be pointed out that enforcing a *Berendragt convention* (i.e. every bound name is distinct from the other bound and free names in the process, see [12]), prior to choosing the canonical names, results in the best analysis estimate, but is not a necessary condition. Without it, an analysis will only be coarser, as different names or variables with the same canonical name, will be coalesced in the analysis estimate. In fact, the precision obtained from the Berendragt convention is somewhat illusionary, as different names, introduced through process invocation and recursion, will be mapped to to the same canonical entity and, thus, introduce inevitable approximation.

5.6.4 Rebinding Variables

Let us finish off the discussion by showing a small encoding trick that greatly increases the practicality of the control flow analysis. The control flow analysis produces safe estimates of all the values each variable may be bound to. In this context, it is often interesting to differentiate between the value bindings that will result in a preemptive termination of the protocol and those that result in a successful termination. Consider, for example, the following naive protocol:

$$\begin{array}{lllll} 1. & A \rightarrow B & : & Na \\ 2. & A \rightarrow B & : & \{Na\}_{Kab} \end{array}$$

Here we assume that A and B share the key Kab prior to protocol execution. This is easily translated into the following extended protocol narration:

$$\begin{array}{lllll} 1. & A \rightarrow & : & Na \\ 1'. & \rightarrow B & : & x \triangleleft [x] \\ 2. & A \rightarrow & : & \{Na\}_{Kab} \\ 2'. & \rightarrow B & : & \{\!|x|\!\}_{Kab} \triangleleft [\,] \end{array}$$

This obviously simulates the intention of the protocol. It is, however, easy to see that this protocol allows anything to be bound to x in step $1'$ and, thus, the analysis will show that $\rho(x)$ includes everything that may float on the net. This result is of practically no use to us; we would rather know how many value bindings that are allowed to get through to step $2'$. Fortunately, the pattern matching mechanism provides us with the possibility to encode exactly this. Indeed, by writing,

$$\begin{array}{lllll} 1. & A \rightarrow & : & Na \\ 1'. & \rightarrow B & : & x \triangleleft [x] \\ 2. & A \rightarrow & : & \{Na\}_{Kab} \\ 2'. & \rightarrow B & : & \{\!|x'|\!\}_{Kab} \triangleleft [x' \mapsto x] \end{array}$$

we dictate the same behaviour to B, but enforce a new variable to be bound in the second matching. Using the control flow analysis on a program based on

the last narration will then show all the bindings to both x and x' and satisfy that $\rho(x') \subseteq \rho(x)$. This small trick allows us to investigate the protocol in a step-by-step manner when necessary.

5.7 Concluding Remarks

This chapter has presented a control flow analysis for CryptoKlaim that computes all values that may be transmitted or bound to variables during execution of a net. In the following chapter, we shall formally define an attacker process and, with it, analyse and verify several cryptographic protocols. This is possible because the analysis captures all events that may occur.

We also prove the correctness of the analysis and, as it produces Horn clauses, it is always possible to find an acceptable analysis estimates automatically with the techniques presented in Chapter 2.

Chapter 6

Verifying Protocols

The theoretical development of a language and a corresponding analysis is all good and well, but one big question remains: "can it be used for anything in practice?" In this chapter we shall try to affirm this.

We begin by formally defining an attacker process representing all malicious activity. This will, when used in conjunction with a protocol specification, reveal how the protocols behave under attack. Afterwards, in §6.2 and §6.3, we attempt to use our analysis to verify a number of well-known cryptographic protocols. Typically, protocol analyses are empirically justified by proving them capable of detecting known flaws to the classical protocols. We will follow this trend, but also show how different scenarios may be taken into account.

Finally, in §6.4, we shall show how to model and verify an industrial system design. We will present a real-life e-banking case study and in a step-wise manner model each part of the system in CryptoKlaim. The resulting specification is then verified using the analysis.

6.1 The Attacker

The environment in which a protocol is run, may not be secure. This means that in order to validate a protocol we generally must assume that it is executed in an environment where there is malicious activity. As for most process algebras, we shall consider the *characteristic context* [30] as described by:

$$N \parallel N_\bullet$$

Here N is the protocol and $N_\bullet$ the attacker. This attacker can see all messages sent on the tuple spaces that it has access to, but not those restricted from its view or any of the internal values in the net N.

Given any concrete attacker $N_\bullet$, the analysis presented in Chapter 5 can account for the behaviour of N under attack from $N_\bullet$. But if we want to analyse the protocol against an arbitrary attacker we need to define an attacker that captures the capabilities of all possible attackers. The standard approach to find this attacker is to define a net that captures the capabilities of an attacker as defined in [44] – the so-called Dolev-Yao attacker. This approach was in [94] also shown to be the hardest attacker, thus justifying this approach.

Dolev and Yao [44] defined the capabilities of the strongest possible network attacker. If we extend this definition to a calculus with distribution and where we assume that the attacker has access to the localities in $\mathsf{L}_\bullet$, then this attacker can:

(1) receive and intercept all messages sent on the localities $\mathsf{L}_\bullet$ and send new messages onto the localities $\mathsf{L}_\bullet$;

(2) decrypt encrypted messages, if it knows the encryption key, and construct new encryptions from known terms;

(3) decompose and compose tuples; and

(4) generate new names.

Due to absorbing input in the semantics of CryptoKlaim, we shall add a fifth rule: (5) duplicate messages.

We now want to define an attacker net, $N_\bullet$, and an attacker process environment, $\Delta_\bullet$, that captures these capabilities, such that for all N if $\rho \models_{\Delta \cup \Delta_\bullet} N \parallel N_\bullet$ then $\rho \models_{\Delta \cup \Delta'} N \parallel N'$ for an arbitrary attacker N' and environment Δ'.

$$
\begin{array}{lllll}
(1\text{A}) & DY_1 & \triangleq & |_{l \in \mathsf{L}_\bullet} \ \mathsf{in}\ x^\bullet \triangleleft [x^\bullet]@l.\ \mathsf{out}\ x^\bullet.\ DY_1 \\
(1\text{B}) & DY_2 & \triangleq & |_{l \in \mathsf{L}_\bullet} \ \mathsf{in}\ x^\bullet \triangleleft [x^\bullet].\ \mathsf{out}\ x^\bullet@l.\ DY_2 \\
(2\text{A}) & DY_3 & \triangleq & \mathsf{in}\ x_0^\bullet \triangleleft [x_0^\bullet].\ \mathsf{in}\ \{\!|x_1^\bullet|\!\}_{x_0^\bullet} \triangleleft [x_1^\bullet].\ \mathsf{out}\ x_1^\bullet.\ DY_3 \\
(2\text{B}) & DY_4 & \triangleq & \mathsf{in}\ x_0^\bullet \triangleleft [x_0^\bullet].\ \mathsf{in}\ [\![x_1^\bullet]\!]_{x_0^\bullet} \triangleleft [x_1^\bullet].\ \mathsf{out}\ x_1^\bullet.\ DY_4 \\
(2\text{C}) & DY_5 & \triangleq & \mathsf{in}\ x_0^\bullet \triangleleft [x_0^\bullet].\ \mathsf{in}\ x_1^\bullet \triangleleft [x_1^\bullet].\ \mathsf{out}\ \{x_1^\bullet\}_{x_0^\bullet}.\ DY_5 \\
(2\text{D}) & DY_6 & \triangleq & \mathsf{in}\ x_0^\bullet \triangleleft [x_0^\bullet].\ \mathsf{in}\ x_1^\bullet \triangleleft [x_1^\bullet].\ \mathsf{out}\ [x_1^\bullet]_{x_0^\bullet}.\ DY_6 \\
(3\text{A}) & DY_7 & \triangleq & |_{k \in \mathcal{K}_+} \ \mathsf{in}\ \langle x_1^\bullet, \ldots, x_k^\bullet \rangle \triangleleft [x_1^\bullet, \ldots, x_k^\bullet].\ \mathsf{out}\ x_1^\bullet \ldots \mathsf{out}\ x_k^\bullet.\ DY_7 \\
(3\text{B}) & DY_8 & \triangleq & |_{k \in \mathcal{K}_+} \ \mathsf{in}\ x_1^\bullet \triangleleft [x_1^\bullet] \ldots \mathsf{in}\ x_k^\bullet \triangleleft [x_k^\bullet].\ \mathsf{out}\ \langle x_1^\bullet, \ldots, x_k^\bullet \rangle.\ DY_8 \\
(4\text{A}) & DY_9 & \triangleq & (\nu\ n_\bullet)\ \mathsf{out}\ n_\bullet.\ DY_9 \\
(4\text{B}) & DY_{10} & \triangleq & (\nu^\pm\ n_\bullet)\ \mathsf{out}\ n_\bullet^+.\ \mathsf{out}\ m_\bullet^-.\ DY_{10} \\
(5) & DY_{11} & \triangleq & \mathsf{in}\ x^\bullet \triangleleft [x^\bullet].\ \mathsf{out}\ x^\bullet.\ \mathsf{out}\ x^\bullet.\ DY_{11}
\end{array}
$$

Figure 6.1: Dolev-Yao attacker processes.

As explained in §2.2.3, the attacker can, in principle, compose and decompose tuples of any arity. But, as already mentioned, it is in [24] shown that restricting the attacker to work on a limited set of arities, does not limit its capabilities, as long as the set includes the finite set of arities occurring in the protocol and at least one additional arity. Assuming that $\mathcal{K}$ denotes the finite set of occurring arities in the net to be analysed, we then create an extended set $\mathcal{K}_+ = \mathcal{K} \cup \{1\}$, as unary tuples never occurs, and a specific Dolev-Yao attacker for any given net can then be formulated with the process environment, $\Delta_\bullet$, defined in Figure 6.1, with the following corresponding net definition:

$$N_\bullet \quad = \quad \|_{i \in \{1,\ldots,11\}} \ l_\bullet :: DY_i$$

The attackers accumulated knowledge is described by the values in the locality $l_\bullet$. Each process definition is then straightforward and correspond to the Dolev-Yao definition.

It follows that $N_\bullet$ constitutes a program together with $\mathsf{L}_\bullet$ and $\Delta_\bullet$ (that is $\mathsf{Prg}(\mathsf{L}_\bullet, \Delta_\bullet, N_\bullet)$) and thus, by Proposition 4.9, it will also constitute a program in conjunction with another program. Furthermore, by Corollary 4.14, we have that the attacker does not violate perfect cryptography, thus establishing the completeness of the attacker definition. For the modelling approach described in §4.6, it is usually natural to assume that $\mathcal{L}_\bullet = \{\updownarrow\}$, but in general other scenarios may be interesting as well.

	$\mathcal{A} \neq \mathcal{B}$ $Ka \neq Kb$	$\mathcal{A} = \mathcal{B}$ $Ka \neq Kb$	$\mathcal{A} \neq \mathcal{B}$ $Ka = Kb$	$\mathcal{A} = \mathcal{B}$ $Ka = Kb$
WMF [6, 31]	√	√	√	(1)
WMF (fix) [72]	√	√	√	(2)
WMF (corr. fix) §C.1.1	√	√	√	√
NS [84]	√	√	√	√
NS (fix) [70]	√	√	√	√
Ya [31]	√	√	√	√
OR [103]	√	√	√	√
AS [117]	√	√	√	√

Table 6.1: Analysis results. √ means no problems found, whereas (1) and (2) means that a problem was found.

6.2 The Usual Suspects

There is a handful of protocols which is represented in virtually any piece of literature about analysis of security protocols written in the last decades; these are the *the usual suspects*. To justify the control flow analysis, it must be shown capable of handling these protocols; that is, verify those that are considered secure and find the known attacks on the flawed ones.

This section presents the accumulated results from worked examples. We have investigated five renowned protocols, the running examples *Needham-Schroeder Public Key (NS)* and *Wide Mouthed Frog (WMF)*, the protocol *Yahalom (Ya)* discussed in Chapter 1 and Chapter 2, and two additional protocols: *Otway-Rees (OR)* and *Andrew-Secure RPC (AS)*. The modelling of NS and WMF has been carried out throughout the chapters and for the remaining three protocols the specification can be found in Appendix C. Some of the protocols also exist in improved versions and we added two of these to the list, bringing the total to seven protocols.

Each protocol has been analysed in different settings with multiple initiators and responders. Specifically, we investigated the impact of when the same keys are used for both initiating and responding in the protocols, $Ka_i^\tau = Ka_i^\tau$, and when principals are allowed as both initiators and responders, $\mathcal{A} = \mathcal{B}$. The different scenarios are modelled simply by changing the definition of the net, following the guidelines described in §4.6.1.

The result of our labour is listed in Table 6.1 and each interesting finding is listed below:

(1) A well-known flaw in WMF, originally pointed out by Lowe in [72], is found. The analysis shows that $Kab_{ij} \in \rho(y_{ji}^{Kab})$ for all i and j. In other words; a key intended to be established between A and B can be incorrectly distributed in a protocol session in the reverse direction. The corresponding attack, explained in the Alice-Bob notation, is as follows:

$$\begin{array}{lllll} 1. & A \rightarrow S & : & A, \{B, Kab\}_{Ka} \\ 2. & S \rightarrow B & : & \{A, Kab\}_{Kb} \\ \mathbf{1}. & I(B) \rightarrow S & : & B, \{A, Kab\}_{Kb} \\ \mathbf{2}. & S \rightarrow A & : & \{B, Kab\}_{Ka} \end{array}$$

The attacker uses the second message to impersonate B and initiate a new protocol session in the opposite direction (the second session is indicated with bold font numbers).

(2) Lowe suggested in [72] an amended version of WMF to correct the flaw. However, our analysis shows that the protocol is still vulnerable, even in the amended version. The attack is an extension to the attack to the original protocol, using interleaving of the two protocol sessions, and a full narration for it is given in §C.1.1. We believe this attack must also be known, as it is fairly straightforward, but we have not been able to find a reference to it in the literature. The flaw is easily corrected and in §C.1.1 we also give the narration for a version that is correctly verified with our analysis.

The reader familiar with the above-mentioned protocols notices that the famous flaw on the Needham-Schroeder protocol is not found. This is not because the analysis is incorrect, but simply a consequence of the scenarios we have considered – it will be found properly in the following section.

6.3 The Insider Threat

Until now, all protocol models and analysis instances have only been concerned with the interaction between legitimate, well-behaved principals, and ensuring that a malicious third party cannot do evil. However, in practice the most effective attacks coming from the inside of a system, based on confidential information that should never have been in the hands of a malicious person in the first place. In this section we shall discuss how such a threat may be taken into account when analysing protocols.

	$\mathcal{A} \neq \mathcal{B}$ $Ka \neq Kb$	$\mathcal{A} = \mathcal{B}$ $Ka \neq Kb$	$\mathcal{A} \neq \mathcal{B}$ $Ka = Kb$	$\mathcal{A} = \mathcal{B}$ $Ka = Kb$
WMF [6, 31]	$\surd$	$\surd$	$\surd$	(1)
WMF (fix) [72]	$\surd$	$\surd$	$\surd$	(2)
WMF (corr. fix) §C.1.1	$\surd$	$\surd$	$\surd$	$\surd$
NS [84]	(3)	(3)	(3)	(3)
NS (fix) [70]	$\surd$	$\surd$	$\surd$	$\surd$
Ya [31]	$\surd$	$\surd$	$\surd$	$\surd$
OR [103]	$\surd$	$\surd$	$\surd$	$\surd$
AS [117]	$\surd$	$\surd$	$\surd$	$\surd$

Table 6.2: Analysis results for the insider threat; $\bullet \in \mathcal{A} \wedge \bullet \in \mathcal{B}$. $\surd$ means no problems found, (1), (2), and (3) means that a problem was found.

Let us first consider what an *insider* is. Obviously, the insider may have access to some information that an outsider does not or access to restricted communication channels. In the protocol setting, we have assumed that all communication is done on one insecure tuple space, so in this context there is no additional information to be gained for an insider. However, each of the cryptographic protocols that we have discussed up until now rely on some sort of confidential information, known prior to a protocol session; namely the shared long-time key or private keys of a trusted key pair. Obviously, an attacker can do harm to legitimate principal's protocol sessions, if it has guessed their secrets. A much more interesting question is the following:

If the attacker is a trusted party, can it interfere with other trusted parties' protocol execution?

In other words, we are interested in verifying protocol executions in a context where the attacker is allowed to be an initiator, $\bullet \in \mathcal{A}$, or a responder, $\bullet \in \mathcal{B}$. We must also ensure that the attacker knows all keys necessary for participating legally with the intended principals; specifically it should know the names passed as arguments to the process invocation describing the principal it simulates.[1] This small adjustment is enough to verify the same protocols for the insider threat.

[1] Practically this can be encoded by transmitting the compromised names without encryption in a locality the attacker has access to.

The results from the insider analysis are presented in Table 6.2. Apart from (1) and (2), which were explained in the previous section, we find the following attack:

(3) Lowe showed in [70] the Needham-Schroeder Public Key protocol to be flawed. This attack is also visible from the analysis result as we see that $Na_{i\bullet} \in \rho(y_{ij}^{Na})$ and $Nb_{ij} \in \rho(y_{i\bullet}^{Na})$ for all i and j. This reflects the following *man-in-the-middle* attack, where the attacker tricks B into believing that it is A:

$$
\begin{array}{llll}
1. & A \rightarrow I & : & [Na, A]_{Ki+} \\
\mathbf{1}. & I(A) \rightarrow B & : & [Na, A]_{Kb+} \\
\mathbf{2}. & B \rightarrow I(A) & : & [Na, Nb, B]_{Ka+} \\
2. & I \rightarrow A & : & [Na, Nb, B]_{Ka+} \\
3. & A \rightarrow I & : & [Nb]_{Kb+} \\
\mathbf{3}. & I(A) \rightarrow B & : & [Nb]_{Kb+}
\end{array}
$$

Notice that the attacker must be a legitimate responder as otherwise A would never initiate a session with it in the first place. Lowe shows how the protocol can be fixed in the same article and this version is correctly verified by our analysis.

It should be noted that while Yahalom is considered a secure protocol, the protocols Otway-Rees and Andrew Secure RPC have flaws with respect to the security property authentication. In both protocols, messages produced in one step of the protocol can be illegally reused at another step by the attacker. Such attacks are known as *Type flaw* attacks. Type flaw attacks will only violate confidentiality or integrity, if they actually result in either that the attacker gains confidential information or that the protocol session results in something unintended. Neither is the case for the attacks to the two protocols and, thus, they are not mentioned in the tables above. In section §7.2 we discuss a simple extension to the analysis that can detect these attacks as well.

6.4 Case Study : E-banking Credit Request

This section presents the case study and its various ingredients. The setting is a credit request from a customer to a bank, and as part of determining whether to grant the credit, the bank has to verify the customer's *Balance Total Assets* (the bank's internal term, denoted *Bta*).

In a service-oriented architecture, such a transaction involves several principals: the *Credit Request Client* (C), a process handling the credit request; a *Validation Service* (VS) who delegates the validation of the balance to the correct service

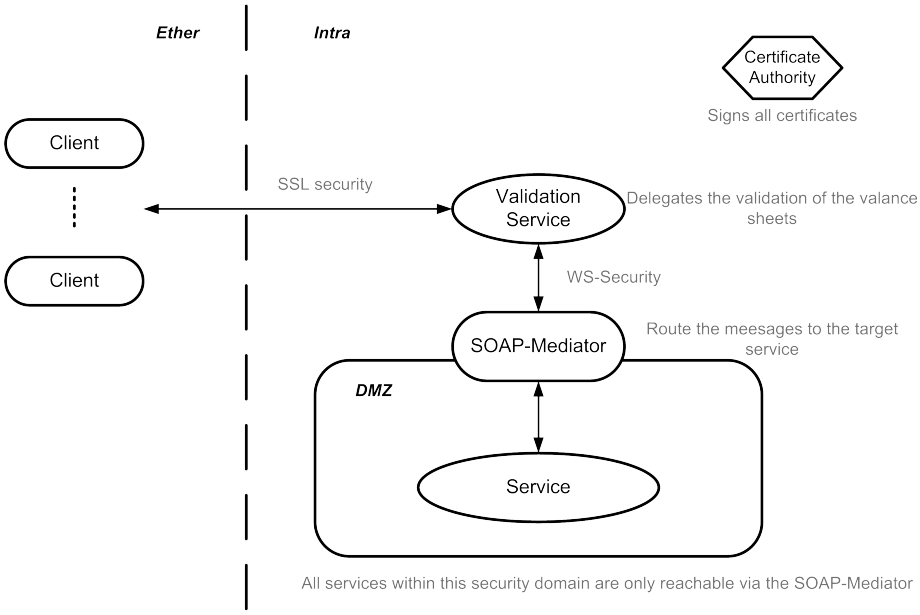

Figure 6.2: The Credit Request case study.

(via WS-Security); a *SOAP-Mediator* (*SM*), who works as an application level router using WS-Addressing and through whom all services are invoked; these services are located on machines in a so-called *demilitarised zone (DMZ)*.

The setup and overview is presented in Figure 6.2 and execution is performed as follows. The Client C establishes a TLS connection to the Validation Service VS, thereby establishing a shared key for encryption of all communication, and through this connection it submits the Bta to be validated. Upon reception of the Bta, the Validation Service will determine the which service that should perform the validation and invoke this service. This invocation is done using WS-Security, which involves encryption of the content of the message but leaves the header in plain text, allowing the SOAP Mediator (*SM*) to route the message correctly within the DMZ. Finally, the Bta is received by the invoked service S_i, analysed, and an answer is replied to the Validation Service, who forwards it to the Client.

The dashed line on the figure symbolises a firewall and it is assumed that the Client is placed outside the banks intra-net, and all other processes within.

6.4.1 X.509 Certificate

The protocols used in the case study both rely on the X.509 ITU-T standard [66] for public key infrastructure. More specifically, the X.509 standard format for public key certificates.

In the X.509 system, a Certificate Authority (CA) issues certificates, which bind public keys to corresponding principals, or rather, to corresponding names. The certificates also include additional information, such as version and serial number, encryption algorithm, issuer, etc., none of which, however, we shall be concerned with in our modelling.

The certificate is authenticated through the Certificate Authority's digital signature. Hence a certificate for a principal A can be formulated in the Alice-Bob notation as:

$$\lfloor A, Ka^{+} \rfloor_{Kca^{-}}$$

Describing the tuple, consisting of principal name and corresponding public key, digitally signed with the certificate authority's private key.

6.4.2 TLS One-Way Handshake

The *Transport Layer Security (TLS)* standard [43] presents a range of cryptographic protocols. In this case study we are interested in the *TLS one-way handshake* protocol, which is used for establishing a symmetric key between a client C and (here) a certified Validation Service VS. The protocol basically involves 3 steps, omitting details for agreeing on encryption algorithm and version number:

1. The Client sends a *ClientHello* message including a freshly generated nonce Nc.
2. The Server responds with a *ServerHello*, containing a new nonce Nvs as well as the Server's certificate $\lfloor VS, Kvs^{+} \rfloor_{Kca^{-}}$, issued by the Certificate Authority CA.
3. The Client responds with a *ClientKeyExchange* message, containing a nonce PMS, called the *Pre-Master Secret*, encrypted with the Server's public key. The Client and Server can now use the nonces Nc, Nvs and PMS to compute a common secret, called the *Master Secret*, by concatenating the three and hashing them, resulting in $\lfloor Nvs, Ns, PMS \rfloor_{H}$ (we here use the encoding for hashing explained in §3.5.2).

The Master Secret is the shared key, and the omitted remainder of the protocol merely serves to establish that the principals agree on it.

This description can be directly translated into an Alice-Bob notation as follows:

$$\begin{array}{lcl} 1. & C \rightarrow S & : \quad Nc \\ 2. & S \rightarrow C & : \quad Nvs, \lfloor VS, Kvs^{+} \rfloor_{Kca^{-}} \\ 3. & C \rightarrow S & : \quad \lfloor PMS \rfloor_{Kvs^{+}} \end{array}$$

Note that the one-way handshake version of TLS only guarantees one-way authentication; i.e. the Client knows that it is talking to the Server, through the Server's certificate, but the Server does not know the identity of the Client. After successful execution of the protocol, the Client and the Server can now communicate through a secure connection:

$$4. \quad C \rightarrow S \quad : \quad \{m\}_{[Nc,Nvs,PMS]_H}$$

Here the Client sends a message, m, to the Server through the newly established tunnel.

6.4.3 WS-Security

WS-Security (Web Services Security) is a communications protocol suite providing security to Web Services, while guaranteeing end-to-end integrity and authentication. WS-Security includes details on the use of various protocols and gives a standardised format for the respective SOAP messages involved in these protocols. However, in the Credit Request case study its use is restricted to signing and encryption of message content, while leaving the message header in plain, to allow SOAP-routing. This is done using the X.509 certificate format.

Assume that a certified principal A wishes to send an encrypted message m (left unspecified) to a certified principal B on a SOAP-routed network, and that A knows and trusts B's certificate $\lfloor B, Kb^{+} \rfloor_{Kca^{-}}$. The simplest abstraction of the message format is then:

$$\begin{array}{lcl} A \rightarrow SM & : & A, B, \lfloor \lfloor m \rfloor_{Kb^{+}} \rfloor_{Ka^{-}} \\ SM \rightarrow B & : & A, B, \lfloor \lfloor m \rfloor_{Kb^{+}} \rfloor_{Ka^{-}} \end{array}$$

The intuition is clear. A encrypts the secret content, using B's public key, signs it, and appends the result to A's and B's names in plain text, indicating the

sender and the receiver and allowing SOAP-routing of the message. The message is then sent on to the network, where it is received by the routing principal SM, who can read the intended recipient and forward the message. Optionally, A may include its certificate in m, in the case it is unknown to B, to allow B to reply.

6.4.4 The Grand Scheme

A necessary assumption for the Credit Request scenario is that all principals trust the Certificate Authority, and thus the certificates. Furthermore the scenario requires prior establishment of some trust relationships. For each principal it assumes that:

- All participants trusts the Certificate Authority CA.
- The Client knows the name of the Validation Service, VS (recall that the Validation Service's certificate is distributed as part of TLS).
- The Validation Service knows the name of the service it should invoke.
- The services requires no additional knowledge.

These requirements are all established in the Trust Establishment case study, which should be verified separately.

Given these assumptions a successful balance validation can be executed. In Table 6.3 a full Alice-Bob specification of such an execution is given.

Step 1-3 establishes an TLS tunnel between the Client and the Validation Service, and in step 4 the Client then submits the Balance Total Assets, Bta, to the Validation Service. The Validation Service then invokes the corresponding service in step 5-6, using WS-Security and SOAP-routing performed by the SOAP-Mediator. This invocation includes both the Bta, a serial number SN to ensure uniqueness of the request, and the Validation Service's certificate, allowing the invoked service to reply using encryption. Finally, in step 7-9, the service replies its evaluation result, Res, to the Validation Service, stating whether the credit has been approved or not. This is again performed through WS-Security and SOAP-routing. Once the Validation Service receives the evaluation result, this is forwarded to the client through the TLS-tunnel, successfully terminating the validation.

(TLS)
1. $C \rightarrow VS : Nc$
2. $VS \rightarrow C : Nvs, [VS, Kvs^{+}]_{Kca-}$
3. $C \rightarrow VS : [PMS]_{Kvs+}$

(BALANCE VALIDATION REQUEST)
4. $C \rightarrow VS : \{Bta\}_{[Nc,Nvs,PMS]_H}$

(SERVICE INVOCATION AND ROUTING)
5. $VS \rightarrow SM : VS, S, [[SN, Bta, [VS, Kvs^{+}]_{Kca-}]_{Ks+}]_{Kvs-}$
6. $SM \rightarrow S : VS, S, [[SN, Bta, [VS, Kvs^{+}]_{Kca-}]_{Ks+}]_{Kvs-}$

(EVALUATION AND REPLY)
7. $S \rightarrow SM : S, VS, [[SN, Res]_{Kvs+}]_{Ks-}$
8. $SM \rightarrow VS : S, VS, [[SN, Res]_{Kvs+}]_{Ks-}$
9. $VS \rightarrow C : \{Bta, Res\}_{[Nc,Ns,PMS]_H}$

Table 6.3: Alice-Bob specification of the Case Study.

6.4.5 Modelling in CryptoKlaim

Without further ado we present the specification of the three main processes (C, VS, and S) in CryptoKlaim in Figure 6.3. Having already explained each step, there is little surprise in the process definitions. The interesting parts are the use of certificates and the model of the communication structure.

The certificate authority sends out certificates whenever needed which, in our modelling, is reflected by placing them on the tuple space l_{Int}. This is used in the modelling of the Validation Service, VS, who must both use certificates to find the public key of the invoked service and to forward its own certificate. Hence the two first actions of VS are local inputs to bind these values to variables. Similarly, the invoked service, S, is assumed only to trust the Certificate Authority. Thus, in order to establish trust to the invoking process as well, a prerequisite of invocation is the existence of a certificate of that process. This is reflected by the first input action of S.

$$
\begin{array}{l}
C[x^{VS}, x^{Kca^+}, x^H] \\
\quad \triangleq \quad (\nu\ Nc)\ (\nu\ PMS)\ (\nu\ Bta) \\
\qquad \mathsf{out}\ Nc@\updownarrow. \\
\qquad\quad \mathsf{in}\ \langle x^{Nvs}, [\![\langle x^{VS}, x^{Kvs^+}\rangle]\!]_{x^{Kca+}}\rangle \triangleleft [x^{Nvs}, x^{Kvs^+}]@\updownarrow. \\
\qquad\qquad \mathsf{out}\ [PMS]_{x^{Kvs+}}@\updownarrow. \\
\qquad\qquad\quad \mathsf{out}\ \{Bta\}_{[\langle Nc, x^{Nvs}, PMS\rangle]_{x^H}}@\updownarrow. \\
\qquad\qquad\qquad C(x^{VS}, x^{Kca^+}, x^H) \\
\\
VS[y^{VS}, y^{Kvs^+}, y^{Kvs^-}, y^{Kca^+}, y^S, y^H] \\
\quad \triangleq \quad (\nu\ Nvs)\ (\nu\ Sn) \\
\qquad \mathsf{in}\ y^{Cert} \triangleleft [y^{Cert} \mapsto [\![\langle y^{VS}, y^{Kvs^+}\rangle]\!]_{y^{Kca+}}]. \\
\qquad\quad \mathsf{in}\ [\![\langle y^S, y^{Ks^+}\rangle]\!]_{y^{Kca+}} \triangleleft [y^{Ks^+}]. \\
\qquad\qquad \mathsf{in}\ y^{Nc} \triangleleft [y^{Nc}]@\updownarrow. \\
\qquad\qquad\quad \mathsf{out}\ \langle Nvs, y^{Cert}\rangle@\updownarrow. \\
\qquad\qquad\qquad \mathsf{in}\ [\![y^{PMS}]\!]_{y^{Kvs-}} \triangleleft [y^{PMS}]@\updownarrow. \\
\qquad\qquad\qquad\quad \mathsf{in}\ \{\!|y^{Bta}|\!\}_{[\langle y^{Nc}, Nvs, y^{PMS}\rangle]_{y^H}} \triangleleft [y^{Bta}]@\updownarrow. \\
\qquad\qquad\qquad\qquad \mathsf{out}\ \langle y^{VS}, y^S, [[\langle Sn, y^{Bta}, y^{Cert}\rangle]_{y^{Ks^+}}]_{y^{Kvs-}}\rangle. \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \langle y^S, y^{VS}, [\![[\![\langle Sn, y^{Res}\rangle]\!]_{y^{Kvs-}}]\!]_{y^{Ks+}}\rangle \triangleleft [y^{Res}]. \\
\qquad\qquad\qquad\qquad\qquad \mathsf{out}\ \{y^{Bta}, y^{Res}\}_{[\langle y^{Nc}, Nvs, y^{PMS}\rangle]_{y^H}}@\updownarrow. \\
\qquad\qquad\qquad\qquad\qquad\quad VS(y^{VS}, y^{Kvs^+}, y^{Kvs^-}, y^{Kca^+}, y^S, y^H) \\
S[z^S, z^{Ks^+}, z^{Ks^-}, z^{Kca^+}] \\
\quad \triangleq \quad (\nu\ Res) \\
\qquad \mathsf{in}\ [\![\langle z^{VS}, z^{Kvs^+}\rangle]\!]_{z^{Kca+}} \triangleleft [z^{VS}, z^{Kvs^+}]@l_{Int}. \\
\qquad\quad \mathsf{in}\ \langle z^{VS}, z^S, [\![[\![\langle z^{SN}, z^{Bta}, [\![\langle z^{VS}, z^{Kvs^+}\rangle]\!]_{z^{Kca+}}\rangle]\!]_{z^{Ks-}}]\!]_{z^{Kvs+}}\rangle \triangleleft [z^{SN}, z^{Bta}]. \\
\qquad\qquad \mathsf{out}\ \langle z^{VS}, z^S, [\![[\![\langle z^{SN}, Res\rangle]\!]_{z^{Kvs+}}]\!]_{z^{Ks-}}\rangle. \\
\qquad\qquad\quad S(z^S, z^{Ks^+}, z^{Ks^-}, z^{Kca^+})
\end{array}
$$

Figure 6.3: Credit Request processes.

$$
\begin{array}{rcl}
SM_1 & \triangleq & \mathsf{in}\,\langle v^A, v^B, v^{Enc}\rangle \triangleleft [v^A, v^B, v^{Enc}].\\
& & \quad \mathsf{out}\,\langle v^A, v^B, v^{Enc}\rangle @ l_{DMZ}.\\
& & \quad\quad SM_1\\
SM_2 & \triangleq & \mathsf{in}\,\langle v^A, v^B, v^{Enc}\rangle \triangleleft [v^A, v^B, v^{Enc}] @ l_{DMZ}.\\
& & \quad \mathsf{out}\,\langle v^A, v^B, v^{Enc}\rangle.\\
& & \quad\quad SM_2\\
CA[u^{Kca^-}, u^{VS}, u^{Kvs^+}, u^S, u^{Ks^+}] & \triangleq & \mathsf{out}\,[\langle u^{VS}, u^{Kvs^+}\rangle]_{u^{Kca^-}}.\\
& & \quad \mathsf{out}\,[\langle u^S, u^{Ks^+}\rangle]_{u^{Kca^-}}.\\
& & \quad\quad CA(u^{Kca^-}, u^{VS}, u^{Kvs^+}, u^S, u^{Ks^+})
\end{array}
$$

Figure 6.4: Credit Request aux. processes.

The net is then structured as follows:

$$
\begin{array}{l}
N_{CR} =\\
\quad (\nu\ VS)\ (\nu\ S)\ (\nu^{\pm}\ Kca)\ (\nu^{\pm}\ Kvs)\ (\nu^{\pm}\ Ks)\ (\nu\ H)\ (\\
\qquad \updownarrow \quad :: C(VS, Kca^+, H)\\
\qquad \| \quad l_{Int} :: VS(VS, Kvs^+, Kvs^-, Kca^+, S, H)\\
\qquad \| \quad l_{Int} :: (SM_1 \mid SM_2)\\
\qquad \| \quad l_{Int} :: CA\\
\qquad \| \quad l_{DMZ} :: S(S, Ks^+, Ks^-, Kca^+)\\
\quad)
\end{array}
$$

The client is placed on the ether, $\updownarrow$. The Validation Service, the Certificate Authority, and the SOAP Mediator are all placed on the intra net, l_{Int}, and the service is placed on the DMZ, l_{DMZ}. This structure simplifies the SOAP Mediator to be a conjunction of two processes; one which moves messages from the intra net to the DMZ and another for the converse direction. The Certificate Authority is also reduced to a simple process which simply outputs the certificates required to the local tuple space. These processes are specified in Figure 6.4.

6.4.6 Analysis Results

As one might expect, the analysis shows that an outside attacker (i.e. an attacker constrained to the locality $\updownarrow$) cannot interfere with the Credit Request regardless of whether there are single or multiple clients. More interestingly, the system remains safe when the attacker is an insider and has access to l_{Int} (or even l_{DMZ}, although such an attacker would probably be capable of doing quite some damage anyway).

It follows, that the Credit Request satisfies integrity and confidentiality; it is impossible for an attacker to violate these properties. It should be mentioned, however, that the security of the system relies entirely on the trust to the Certificate Authority, CA, and that the attacker may trivially attack the system if it itself is certified by CA (in that case S and possibly VS, depending on the implementation, would implicitly trust the attacker). Thus, it should be stressed that proper guideline for design is that the Certificate Authority should be local to the system and that certificates should not be easily obtained.

Chapter 7

Conclusion

7.1 Recapitulation

Modelling and analysis of distributed systems has been an active area of research for many years; indeed, the first attempts at formally analysing cryptographic protocols were produced three decades ago. So how can we claim that it is still in the embryonic stage in many respects?

The intrinsic problem with the theoretical approaches are that they usually do not reach their most important audience – the developers of communication systems. Modern development traditions seem to defy the application of formal techniques: technical standards are written in prose, focussing on implementation issues, and security is seldom a primary concern. Academic models, on the other hand, are usually proven mathematically correct and equipped with automated tools for verification, but formulated in languages that requires a wizard to decipher.

This dissertation has presented an attempt to bridge this gap. We formulated a formal language, CryptoKlaim, that is intended to be intuitive to the developer while maintaining the attractive property that it is verifiable. To reach this goal, we have presented contributions to the areas of constraint solving, process calculi and automated verification in networking system.

The dissertation took its offset in one of the oldest approaches to modelling cryptographic protocols: first-order logic. To deal with the problem of undecidability and non-termination, we introduced a general framework for finding approximative models of Horn clauses that guarantees soundness and termination. This serves as a perfect base of static analysis, where we allow some amount of approximation as long as it is on the safe side.

Based on the modelling in Horn clauses and the underlying framework, we built, in turn, the cryptographic components and CryptoKlaim itself. The language is equipped with a formal semantics, describing the entire execution space, and a well-formedness condition which, among other things, guarantee the assumption of perfect cryptography.

The design of the language was analysis-directed and, fortunately, this resulted in a straightforward development of a corresponding Control Flow Analysis. This analysis was naive in many respects, yet it proved precise enough to find the usual well-known flaws in many of the classical protocols. It even showed to be capable of verifying an industrial case study: an e-banking credit request. The case study was not only interesting because it was an example of a high-security, modern system design, but also because the entire modelling was done in collaboration with the original developers of the system. For confidentiality issues, this company wishes to remain anonymous, but it still serves to show that our original goal has been somewhat reached.

The remainder of this chapter, and the dissertation, will present a number of suggestions for future lines of work, extensions to the calculus and the analysis.

7.2 Directions for Future Work

Each chapter, which presented a theoretical development, has been concluded with a discussion of some of the alternatives or technicalities that were omitted during the presentation. The purpose of this section is not to reiterate those sections, but to extract the three most promising lines of future work from those discussions.

7.2.1 Dynamic Networks

Originally, KLAIM includes the actions $\mathsf{eval}(P)@l$ and $\mathsf{newloc}(l)$, for remote evaluation of processes and dynamic creation of tuple spaces. In CryptoKlaim

the tuple spaces are assumed static under evaluation and the language does not allow communication of localities, thus these actions would be superfluous and are omitted in the presentation. This means that CryptoKlaim cannot model dynamically changing networks properly, and while it is probably fair to assume that the structure of a wired network remains static, it is less likely for an ad-hoc network.

Technically, the language should be extended so that communication of localities is possible; either by coalescing the set of localities and names into one unified set, or, perhaps more elegantly, allow an additional syntactic category, localities, in terms. It should, however, be carefully considered if such a model of ad-hoc networks is realistic or whether it would be better to assume a topology in the style of [83].

7.2.2 Security Properties by Annotations

In this dissertation we have focussed on the security properties confidentiality and integrity. As mentioned in the introduction, there are numerous other properties, and in relation to cryptographic protocols one is particularly interesting: authentication.

Authentication is the prevention of forging communication of information. With respect to cryptographic protocols this is usually in the form of prevention of *replay attacks*. Syverson [122] presents the following taxonomy of replay attacks:

1. Replay of messages between sessions.
    a. Interleavings (requiring contemporaneous protocol runs)
        i. Deflections (message is directed to other than the intended recipient)
        ii. Straight replays (intended principal receives message, but message is delayed)
    b. Classic replays (runs need not be contemporaneous)
        i. Deflections (message is directed to other than the intended recipient)
        ii. Straight replays (intended principal receives message, but message is delayed)
2. Replay of messages within the same session.
    a. Deflections (message is directed to other than the intended recipient)

b. Straight replays (intended principal receives message, but message is delayed)

Any replay attack that would result in a variable being bound to an unintended value will be caught by the analysis presented in this dissertation. However, replay attacks, such as those on the Otway-Rees protocol (same value from a different step replayed) or the Andrew-Secure RPC (same value and step but different session), will not.

Such attacks require an authentication analysis such as the correspondence assertion analysis by Gordon and Jeffrey [58] or Blanchet [20]. However, Bodei et al. [22] show how a restricted type of authentication, called *destination/origin authentication*, suffices to detect all but one of the replay attack types (it cannot detect classic, straight replay attacks). This work is purely based on annotations and relies on a Control Flow Analysis similar to the one presented in this dissertation. In short, the technique amounts to decorating each encryption and decryption with a label and a destination/origin set. This could be incorporated in CryptoKlaim in the following way:

$$\{t_1\}^{\ell}_{t_0}[\mathsf{dest}\ \mathcal{L}] \qquad \{\!|t_1|\!\}^{\ell}_{t_0}[\mathsf{orig}\ \mathcal{L}]$$

The annotations would then be ignored by the semantics but guide the analysis. Specifically, if a value $\{v_1\}^{\ell}_{v_0}[\mathsf{dest}\ \mathcal{L}]$ is successfully matched against a decoding $\{\!|d_1|\!\}^{\ell'}_{d_0}[\mathsf{orig}\ \mathcal{L}']$ then the pair $\langle \ell, \ell' \rangle$ turns up in an error estimate whenever $\ell \notin \mathcal{L}'$ or $\ell' \notin \mathcal{L}$. This analysis is also translatable to Horn clauses.

Later work by Gao et. al [52] has shown that the last type of replay attacks can also be detected using similar annotation techniques and unfolding of the recursive process definitions. It follows that authentication is simply a question of correct annotation of the protocol and an orthogonal issue to the Control Flow Analysis. It would, however, be interesting to automatically extract these annotations from the protocol specification, thereby easing the amount of human interaction and work involved in the process.

7.2.3 Simpler Specifications

The language CryptoKlaim is created with the purpose of being intuitive and verifiable. The language itself allows a large amount of code-reusability and other clever encoding tricks, but unfortunately these are not supported by the current analyses.

First and foremost it would be convenient if it was not necessary to ensure singleton bindings of the variables in the process. This would allow reuse of

process definitions for various purposes; e.g. the protocol specifications in §4.6.1 could reuse the process definitions instead of having to create one for each combination of initiator and responder. To reach this goal, the analysis must be extended in several ways, and some of the required techniques for this are discussed in §5.6.2.

Interestingly, however, the challenge is mostly related to the formalisation and proof for soundness as it is already supported by Horn clauses. The two main ingredients would be context-dependent variable bindings (i.e. collecting variable bindings in the context of the previous variable bindings) and context-dependent name generation (creating different names for different instances of the process invocations). The former can be introduced by collecting tuples of variable bindings and the latter by modelling names with non-nullary constructors. This is analogous to the technique we suggested in Chapter 2 and shown to yield very precise models when normalised with the iterative scheme.

Appendix A

Theoretical Preliminaries and Notation

All definitions, notation and proofs in this dissertation rely heavily on knowledge of basic set and order theory, as well as the most common proof techniques. Although the reader is assumed to have a basic understanding of this mathematical branch, the simplicity and elegance of set theory allows us to present most relevant definitions and results for this dissertation in a succinct way in this appendix. However, as it is only supposed to be used for reference as it is stripped of all proofs, and the interested reader should refer to [42] or [124] for a more detailed explanation. The mathematical foundation of set theory and logic is well presented in [121] and also, very elegantly using the naive set theory approach, in [61]. Finally, for a briefer introduction to lattices and orders than the above mentioned, yet more elaborate than this appendix, see [93].

A.1 Order Theory

The foundation of program analysis is partial orders and the theory thereof.

Definition A.1 (Partial order) *A partial order $(A, \sqsubseteq)$ is a set A accompanied by a binary relation $\sqsubseteq\colon A \times A$ that is reflexive, transitive and antisymmetric.*

Notation A.2 *When necessary, we shall use subscripts to disambiguate the denotation of operators.*

If A has an element $a \in A$ such that $\forall a' \in A : a \sqsubseteq a'$, then this element is called the *least element* of A and is denoted $\bot$. Analogously, the *greatest element* of A is an element $a \in A$ such that $\forall a' \in A : a' \sqsubseteq a$ and is denoted $\top$. This may be generalised:

Definition A.3 (Upper bound) *For a partial order $(A, \sqsubseteq)$ and subset $S \subseteq A$ the element $a \in A$ is an upper bound of S if and only if*

$$\forall s \in S : s \sqsubseteq a$$

Definition A.4 (Least upper bound) *For a partial order $(A, \sqsubseteq)$ and subset $S \subseteq A$ the element $a \in A$ is a least upper bound of S if and only is*

(i) a is an upper bound of S

(ii) For all upper bounds s of S then $a \sqsubseteq s$

Whenever a is the least upper bound of S we denote it $\bigsqcup S$.

Specifically, we may denote the binary least upper bound of $a_1, a_2 \in A$ as $a_1 \sqcup a_2$ representing $\bigsqcup\{a_1, a_2\}$. The dual of the least upper bound is the *greatest lower bound*, which is defined analogously. We write $\bigsqcap S$ when referring to the greatest lower bound of S and $a_1 \sqcap a_2$ represent $\bigsqcap\{a_1, a_2\}$.

Definition A.5 (Lattice) *A non-empty partial order $(L, \sqsubseteq)$ that is closed under $\sqcup$ and $\sqcap$ (i.e. for all $l, l' \in L$ then $(l \sqcup l') \in L$ and $(l \sqcap l') \in L$) is a* lattice.

Definition A.6 (Complete lattice) *A non-empty partial order $(L, \sqsubseteq)$ that has $\bigsqcup S \in L$ and $\bigsqcap S \in L$ for all $S \subseteq L$ is a complete lattice.*

It follows that if $(L, \sqsubseteq)$ is a complete lattice then $\bot = \bigsqcup \emptyset = \bigsqcap L$ is the least element and $\top = \bigsqcap \emptyset = \bigsqcup L$ is the greatest element. Furthermore, Definition A.5 and Definition A.6 ensure that a complete lattice is also a lattice, and that any finite lattice is complete.

Proposition A.7 (Cartesian product of lattices) *If* $(L_1, \sqsubseteq_1)$ *and* $(L_2, \sqsubseteq_2)$ *are both complete lattices then so is the Cartesian product*

$$(L_1 \times L_2, \sqsubseteq)$$

where the componentwise order *is defined by*

$$\langle l_1, l_2 \rangle \sqsubseteq \langle l_1', l_2' \rangle \quad \Leftrightarrow \quad (l_1 \sqsubseteq_1 l_1') \wedge (l_2 \sqsubseteq_2 l_2')$$

Proposition A.8 (Function space of lattices) *Let* A *be a set and* $(L, \sqsubseteq_L)$ *be a complete lattice, then the* function space $(A \to L, \sqsubseteq)$ *where the* pointwise order *is defined by*

$$f \sqsubseteq g \quad \Leftrightarrow \quad \forall a \in A : f(a) \sqsubseteq_L g(a)$$

is also a complete lattice.

Definition A.9 (Moore-family) *A subset,* S*, of a complete lattice,* $(L, \sqsubseteq)$*, is a* Moore-family *if it is closed under greatest lower bounds, i.e.,*

$$\forall S' \subseteq S : \bigsqcap S' \in S$$

It follows that a Moore-family always contains a least element, $\bigsqcap S$, and a greatest element, $\bigsqcap \emptyset = \top_L$; thus it is never empty.

Definition A.10 (Monotone function) *A function,* $f : A \to B$*, between partially ordered sets,* $(A, \sqsubseteq_A)$ *and* $(B, \sqsubseteq_B)$*, is* monotone *if*

$$\forall a_1, a_2 : a_1 \sqsubseteq_A a_2 \Rightarrow f(a_1) \sqsubseteq_B f(a_2)$$

Definition A.11 (Chain) *A subset,* S*, of a partial order,* $(A, \sqsubseteq)$*, is a* chain *if it is totally ordered (i.e.* $\forall s_1, s_2 \in S : (s_1 \sqsubseteq s_2) \vee (s_2 \sqsubseteq s_1)$*). If* S *is a finite subset then it constitutes a* finite chain.

We say that a sequence $(a_n)_{n \in \mathbb{N}}$ of elements in $(A, \sqsubseteq)$ is an *ascending chain*, whenever

$$\forall n \leq m : a_n \sqsubseteq a_m$$

and, similarly, that it is a *descending chain* if

$$\forall n \leq m : a_n \sqsupseteq a_m$$

Clearly both ascending and descending chains are also chains.

Definition A.12 (Ascending chain condition) *A partial order, $(A, \sqsubseteq)$, satisfies the* ascending chain condition *if any ascending chain, $(a_n)_{n\in\mathbb{N}}$,* eventually stabilises*, i.e.*

$$\exists k : \forall n \geq k : a_k = a_n$$

And, analogously, we define the *descending chain condition* by substituting "descending" for "ascending" in Definition A.12. Note that any finite lattice trivially satisfies both the ascending and descending chain condition.

Definition A.13 (Fixed point) *Consider a monotone function, $f : L \to L$, on a complete lattice, $(L, \sqsubseteq)$. A fixed point of f is an element, $l \in L$, such that $f(l) = l$. We write*

$$\mathit{Fix}(f) = \{l \mid f(l) = l\}$$

for the set of such fixed points.

Proposition A.14 (Tarski's fixed point theorem) *Any monotone function, $f : L \to L$, on a complete lattice, $(L, \sqsubseteq)$, has a unique* least fixed point*, $\mathit{lfp}(f)$, defined by:*

$$\mathit{lfp}(f) = \bigsqcap \mathit{Fix}(f) = \bigsqcap \{l \mid f(l) \sqsubseteq l\} \in \mathit{Fix}(f)$$

and, similarly, a unique greatest fixed point*, $\mathit{gfp}(f)$, given by:*

$$\mathit{gfp}(f) = \bigsqcup \mathit{Fix}(f) = \bigsqcup \{l \mid f(l) \sqsupseteq l\} \in \mathit{Fix}(f)$$

We usually say that f is *reductive* upon the elements of $\{l \mid f(l) \sqsubseteq l\}$ and *extensive* upon the elements of $\{l \mid f(l) \sqsupseteq l\}$. Clearly, if L satisfies the ascending chain condition, then $\mathit{lfp}(f)$ may be computed in a finite number of iterations, from any element l that f is reductive at, as the sequence $(f^n(l))_{n\in\mathbb{N}}$ will eventually stabilise. It is customary to choose $\bot$ at the starting point of the sequence; i.e. $\bigsqcup_{n\in\mathbb{N}} f^n(\bot) = \mathit{lfp}(f)$. Analogously, we have that if the lattice satisfies the descending chain condition then $\bigsqcap_{n\in\mathbb{N}} f^n(\top) = \mathit{gfp}(f)$.

A.2 Proof Techniques

This appendix gives a brief introduction to the proof techniques used in this dissertation. The presentation relies heavily on the one given in [93], but presented in the notation chosen for this dissertation, for convenience of the reader.

A.2.1 Induction

Induction is most well-known in the form of *mathematical induction*, originally introduced by Euclid, where a logical property $\mathsf{Q}(n)$ is shown to hold for all natural numbers, $n \in \mathbb{N}$, by showing that it holds for the *base case*, $\mathsf{Q}(0)$, and for the *inductive step*, $\forall n \in \mathbb{N} : \ \mathsf{Q}(n) \Rightarrow \mathsf{Q}(n+1)$. In proving the inductive step, one assumes that $\mathsf{Q}(n)$ holds, referred to as the *induction hypothesis*, in order to show that $\mathsf{Q}(n+1)$ holds. However, the principle carries on over to other algebraic structures as well, and we here outline the most important ones.

Structural Induction

Structural induction is a generalisation of mathematical induction, and relies on structural recursion on any recursively defined structure as mathematical induction relies on ordinary recursion. Consider an algebraic data type, given by the abstract syntax:

$$d \in \mathsf{D}$$
$$d ::= \mathsf{base} \mid \ \diamond d \mid d \oplus d$$

where base is the base case, $\diamond$ is a unary constructor, and $\oplus$ is a binary constructor. We may then prove that a certain property

$$\mathsf{Q} : \mathsf{D} \to \{\mathsf{true}, \mathsf{false}\}$$

holds for all elements, $d \in \mathsf{D}$, by *structural induction*, if we can show:

$$(i) \quad \mathsf{Q}(\mathsf{base})$$
$$(ii) \quad \forall d \in \mathsf{D} : \ \mathsf{Q}(d) \Rightarrow \mathsf{Q}(\diamond d)$$
$$(iii) \quad \forall d_1, d_2 \in \mathsf{D} : \ \mathsf{Q}(d_1) \wedge \mathsf{Q}(d_2) \Rightarrow \mathsf{Q}(d_1 \oplus d_2)$$

Here, (i) is the base case, and (ii) and (iii) the inductive steps. Moreover, the preconditions $\mathsf{Q}(d)$ in (ii) and $\mathsf{Q}(d_1) \wedge \mathsf{Q}(d_2)$ in (iii) represent the induction hypothesis.

Induction on the Shape

Equivalently to induction in the structure of a data type, we may also use induction on the rules defining a relation. This is usually used upon natural

semantics. Consider the binary relation $\longrightarrow$ (typed $\mathsf{D} \times \mathbb{N}$) given by the rules:

$$(\textsc{base}) \quad \mathsf{base} \longrightarrow 0$$

$$(\textsc{con}1) \quad \frac{d \longrightarrow n}{\diamond d \longrightarrow n+1}$$

$$(\textsc{con}2) \quad \frac{d_1 \longrightarrow n_1 \quad d_2 \longrightarrow n_2}{d_1 \oplus d_2 \longrightarrow n_1 + n_2}$$

It follows that we have a notion of *inference trees* (or equivalently *evaluation trees* or *derivation trees*), $d_1 \xrightarrow{\triangledown} d_2$, expressing the entire tree used to derive the relation between two elements.[1] Now, we may prove that all such inference tree satisfies a certain logical property, Q, by showing that the property holds for each base case, and that if it holds for the precondition of a recursive rule, then it also holds for the inference tree of the conclusion. This is called *induction on the structure of the inference trees*, and for $\longrightarrow$ it amounts to showing:

$$(i) \quad \mathsf{Q}(\mathsf{base} \longrightarrow 0)$$

$$(ii) \quad \forall (d \xrightarrow{\triangledown} n) : \mathsf{Q}(d \xrightarrow{\triangledown} n) \Rightarrow \mathsf{Q}\left(\frac{d \xrightarrow{\triangledown} n}{\diamond d \xrightarrow{\triangledown} n+1}\right)$$

$$(iii) \quad \forall (d_1 \xrightarrow{\triangledown} n_1), (d_2 \xrightarrow{\triangledown} n_2) : \ \mathsf{Q}(d_1 \xrightarrow{\triangledown} n_1) \wedge \mathsf{Q}(d_2 \xrightarrow{\triangledown} n_2) \Rightarrow \mathsf{Q}\left(\frac{d_1 \xrightarrow{\triangledown} n_1 \quad d_2 \xrightarrow{\triangledown} n_2}{d_1 \oplus d_2 \xrightarrow{\triangledown} n_1 + n_2}\right)$$

From which we conclude

$$\forall (d \xrightarrow{\triangledown} n) : \ \mathsf{Q}(d \xrightarrow{\triangledown} n)$$

Again we say that (i) is the base case, and that the precondition in the inductive steps (ii) and (iii) is the induction hypothesis.

We say that a relation defined recursively upon a recursive structure, such as $\longrightarrow$ above, is *compositional* (sometimes *syntax-directed*); i.e. the validity of the relation between complex expressions may be determined based on the validity of the constituent expressions and the rules used to combine them. Compositional definitions are considered attractive, as they (usually) allow straightforward proofs by induction. Relations that are not defined compositionally are usually referred to as *abstract*.

[1] Usually $\xrightarrow{\triangledown}$ is written in infix style, but a more rigorous definition would represent it by a function. As the structure of the inference trees is usually superfluous, the operator in the example above could simply be typed $\xrightarrow{\triangledown} : (\mathsf{D} \times \mathbb{N}) \to \wp(\mathsf{D} \times \mathbb{N})$.

Well-founded induction

The last type of induction that we shall present in this appendix is *well-founded induction*. Assume a logical relation, $\mathsf{Q} : A \to \{\mathsf{true}, \mathsf{false}\}$, on a partially ordered set, $(A, \sqsubseteq)$, that satisfies the descending chain condition. If we prove

$$\forall a \in A : \ (\forall a' \sqsubseteq a : \mathsf{Q}(a')) \ \Rightarrow \ \mathsf{Q}(a)$$

then we may conclude

$$\forall a \in A : \mathsf{Q}(a)$$

Notice that the base cases are concealed within the definition, as each descending chain must have a least element, for which the precondition is universally true.

A.2.2 Co-induction

Assume that we wish to show a logical property

$$\mathsf{Q} : \mathsf{D} \to \{\mathsf{true}, \mathsf{false}\}$$

defined by the means of clauses

$$\begin{aligned} \mathsf{Q}(\mathsf{base}) \quad & \textit{iff} \quad \cdots \\ \mathsf{Q}(\diamond d) \quad & \textit{iff} \quad \cdots \mathsf{Q}(d') \cdots \\ \mathsf{Q}(d_1 \oplus d_2) \quad & \textit{iff} \quad \cdots \mathsf{Q}(d'_1) \cdots \mathsf{Q}(d'_1) \cdots \end{aligned}$$

holds for some or all elements of D. Notice that the clauses are given by biimplication, and thus deciding $\mathsf{Q}(d)$ for an element d may be deduced through application of either direction of a clause. In fact, there may even exist elements, d, for which an inductive approach to deciding $\mathsf{Q}(d)$ would never terminate. In these cases we may attempt an alternative approach.

Let us first use the clauses above as a basis for a definition of a functional (i.e. a function whose argument and result also contain functions):

$$\begin{aligned} \mathcal{A}[\mathsf{Q}'](\mathsf{base}) \quad & \textit{iff} \quad \cdots \\ \mathcal{A}[\mathsf{Q}'](\diamond d) \quad & \textit{iff} \quad \cdots \mathsf{Q}'(d') \cdots \\ \mathcal{A}[\mathsf{Q}'](d_1 \oplus d_2) \quad & \textit{iff} \quad \cdots \mathsf{Q}'(d'_1) \cdots \mathsf{Q}'(d'_1) \cdots \end{aligned}$$

We say that $\mathcal{A}$ is the *analysis functional* defined by Q. Clearly Q holds for all elements of D if and only if $\mathcal{A}[\mathsf{Q}]$ holds for all elements of D. Furthermore, we have that $(\mathsf{D} \to \{\mathsf{true}, \mathsf{false}\}, \sqsubseteq)$ is a complete lattice under the ordering

$$\mathsf{Q}_1 \sqsubseteq \mathsf{Q}_2 \quad \Leftrightarrow \quad \forall d : \mathsf{Q}_1(d) \Rightarrow \mathsf{Q}_2(d)$$

It follows that the least element, $\bot$, is given by $\forall d : \bot(d) = \mathsf{false}$, and the greatest element, $\top$, by $\forall d : \top(d) = \mathsf{true}$. Hence, we may apply Tarski's fixed point theorem whenever $\mathcal{A}$ is monotone (i.e. Q does not contain clauses such as "$\mathsf{Q}(\diamond d)$ *iff* $\neg\mathsf{Q}(d)$").

Specifically, if the ascending chain $(\mathcal{A}^n[\bot])_{n\in\mathbb{N}}$ eventually stabilise then we may find Q as the least fixed point of $\mathcal{A}$:

$$\mathsf{Q} = \mathit{lfp}(\mathcal{A}) = \bigsqcup_{n\in\mathbb{N}} \mathcal{A}^n[\bot]$$

This is equivalent to a proof by induction. Conversely, if the descending chain $(\mathcal{A}^n[\top])_{n\in\mathbb{N}}$ eventually stabilise then we may find Q as the greatest fixed point of $\mathcal{A}$:

$$\mathsf{Q} = \mathit{gfp}(\mathcal{A}) = \bigsqcap_{n\in\mathbb{N}} \mathcal{A}^n[\top]$$

This is known as *co-induction*. Note that the inductive and co-inductive approach may result in different Q's if the least and the greatest fixed point do not coincide, but that both approaches will find a Q which satisfies the clauses used for defining it.

Appendix B

A Succinct Solver for $\mathcal{H}_1$

B.1 Solver Implementation

In §2.4.3 we presented an inference system for, as well as an example of, the inductive steps needed for normalisation of an $\mathcal{H}_1$-clause. But, naturally, formulae consisting of more than a mere few clauses are generally not feasible to manually normalise; we need automated support for this. This appendix suggest a succinct, yet efficient, implementation of an $\mathcal{H}_1$-*solver*; i.e. an automated normalisation procedure for $\mathcal{H}_1$.

The solver is implemented in a demand-driven manner, and it relies on efficient implementation of some basic methods. In particular, we shall assume the existence the general methods defined in Table B.1.

The algorithm is then divided into four phases:

- **Phase 1.** First a universal predicate, i.e. a predicate including the entire Herbrand universe, is added in order to allow the solver to cope with unrestricted variables.
- **Phase 2.** The conclusion of each clause is flattened.

new_pred ()	Returns a fresh predicate name.
new_var ()	Returns a fresh variable name.
var e	Returns the set of variables occurring in e.
cons e	Returns the set of constructors, f_n, where f is the constructor name and n is the arity, occurring in e.
register fun	Stores the unary function fun, to be invoked whenever a clause is added to the least solution, $\widehat{\varphi}$, with this clause as argument.

Table B.1: Basic methods

let solve φ =
  forall (f, n) **in** (cons φ)
  do add $(\mathsf{p}_H(f(X_1, \ldots, X_n)) \Leftarrow \mathsf{p}_H(X_1) \wedge \cdots \wedge \mathsf{p}_H(X_n))$;
  forall $c \in \varphi$ **do** norm_con c

Figure B.1: Phase 1 - Introduce universal predicate; p_H.

- **Phase 3.** The precondition of each clause is transformed into a set of unary predicates.

- **Phase 4.** The core solving; determining satisfiability of all (intersections of) predicates in the precondition of each clause, and, if all the predicates are satisfiable, adds the conclusion to the least solution.

Each of the phases are described in detail below along with a pseudo-code implementation.

B.1.1 Phase 1

The first phase introduces a predicate p_H, for which the least solution is the entire Herbrand universe, H_φ, of the formula φ. Using the method cons that returns the set of constructors in an expression, this is done by adding a self-referential transition for every constructor. The resulting pseudo-code is given in Figure B.1.

```
let rec norm_con (g ⇐ ω) =
    match g with
    | p(t1, ..., tn)      → let (⟨X1, ..., Xn⟩, ω') = split ⟨t1, ..., tn⟩ ω
                            in norm_pre (p(X1, ..., Xm) ⇐ ω')
    | p(f(t1, ..., tn)) → let (⟨X1, ..., Xn⟩, ω') = split ⟨t1, ..., tn⟩ ω
                            in norm_pre (p(f(X1, ..., Xn)) ⇐ ω')

and split ⟨t1, ..., tn⟩ ω =
    if n = 0 then (ϵ, ω)
    else match t1 with
        | X1 → let (⟨X2, ..., Xn⟩, ω') = split ⟨t2, ..., tn⟩ ω
               in if X1 ∈ (var ω)
                  then (⟨X1, ..., Xn⟩, ω')
                  else (⟨X1, ..., Xn⟩, pH(X1) ∧ ω')
        | _  → let (⟨X2, ..., Xn⟩, ω') = split ⟨t2, ..., tn⟩ ω and
                   (X1, p1) = (new_var (), new_pred ())
               in norm_con (p1(t1) ⇐ ω);
                  (⟨X1, ..., Xn⟩, p1(X1) ∧ ω')
```

Figure B.2: Phase 2 - Normalise conclusion.

B.1.2 Phase 2

In the second phase, each clause is transformed into an equivalent set of clauses, in which each clause is of the form:

$$\begin{array}{lcl} \mathsf{p}(X_1, \ldots, X_m) & \Leftarrow & \omega \\ \mathsf{p}(f(X_1, \ldots, X_n)) & \Leftarrow & \omega \end{array}$$

Additionally we require all variables in the conclusion to also occur in the precondition. Notice that if the same variable occurs multiple times in the conclusion, then this is simply ignored by the solver, and the occurrences are treated individually.

The pseudo-code for Phase 2 is given in Figure B.2. The implementation consists of two mutually recursive functions, norm_con and split. norm_con simply uses split to flatten the conclusion of the current clause, and then forwards this to Phase 3. split introduces an auxiliary predicate and a new clause for each term in the conclusion that violates the form above. It then uses norm_con to flatten the new clause. If an unbound variable X is encountered, i.e. if it is not present in the precondition, then it is bound by the Herbrand universe by adding the literal $\mathsf{p}_H(X)$ to the precondition.

```
let  norm_pre c = norm_pf c ϵ
and rec norm_pf (g ⇐ ω) ω' =
    match ω with
    | ϵ                          → add (g ⇐ ω')
    | p(X) ∧ ω''                 → norm_pf (g ⇐ ω'') (p(X) ∧ ω')
    | p(t1, ..., tm) ∧ ω''       →
        let fun c = match c with
                    | (p(X1, ..., Xm) ⇐ ω''') → let θ = [X1 ↦ t1, ..., Xm ↦ tm]
                                                in norm_pf (g ⇐ ω'''θ ∧ ω'') ω'
                    | _                       → ()
        in register fun;
           forall c ∈ φ̂ do fun c
    | p(f(t1, ..., tn)) ∧ ω'' →
        let fun c = match c with
                    | (p(f(X1, ..., Xn)) ⇐ ω''') → let θ = [X1 ↦ t1, ..., Xn ↦ tn]
                                                   in norm_pf (g ⇐ ω'''θ ∧ ω'') ω'
                    | _                          → ()
        in register fun;
           forall c ∈ φ̂ do fun c
```

Figure B.3: Phase 3 - Normalise precondition.

B.1.3 Phase 3

The third phase converts the precondition into a set of unary literals, resulting in clauses of the form:

$$\begin{array}{lcl} \mathsf{p}(X_1, \ldots, X_m) & \Leftarrow & \mathsf{q}_1(Z_1) \wedge \cdots \wedge \mathsf{q}_k(Z_k) \\ \mathsf{p}(f(X_1, \ldots, X_n)) & \Leftarrow & \mathsf{q}_1(Z_1) \wedge \cdots \wedge \mathsf{q}_l(Z_l) \end{array}$$

Where the variables occurring in the conclusion also occurs in the precondition. This is done through resolution; iterative simplification of the precondition using clauses already in normal form (i.e. the transitions in the automaton describing the least model). Furthermore, if future normal clauses are added, these may also be used for resolving the clauses.

The pseudo-code for Phase 3 is given in Figure B.3. The function keeps track of the precondition already normalised, ω', and uses the method call register fun to store a function fun to be invoked whenever a new clause is added to the solution. When the precondition is empty, then the normalised clause is added to the solution, which is handled by Phase 4.

store c	Add the clause c to the set of normalised clauses $\widehat{\varphi}$
wait p fn	Adds the nullary function fn to the queue waiting for p to be non-empty.
requested $\{\mathsf{p}_1, \ldots, \mathsf{p}_k\}$	Returns a boolean indicating whether the intersection of the predicates $\mathsf{p}_1, \ldots, \mathsf{p}_k$ has been requested yet.

Table B.2: Methods for the Core Solver.

B.1.4 Phase 4

Phases 1,2, and 3 essentially described result in resolution, and we could simply stop after Phase 3, resulting in a polynomial normalisation, but an exponential time lookup in the least model. However, we wish to bring the clauses onto the normal form of Table 2.3 and that is performed by Phase 4. This phase is the Core Solver and it relies on the additional methods described in Table B.2.

The Core Solver adds only clauses to the solution if their precondition is satisfiable. The pseudo-code for a function, add, that does this is given in Figure B.4. The solver must compute all intersections occurring in the preconditions, i.e. unary predicates referring to the same variable, and for this we now assume that all predicates handles are sets of predicate names. This set indicates the intersection of original predicates they represent; i.e. the predicate $\{\mathsf{p}, \mathsf{q}\}$ represents the intersection of the predicates $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$, or analogously, as only intersection of unary predicates must be calculated, the solution to $\{\mathsf{p}, \mathsf{q}\}(X) \Leftarrow \{\mathsf{p}\}(X) \wedge \{\mathsf{q}\}(X)$.

The function add checks that the preconditions, i.e. the predicates in the precondition of the normalised clause, are satisfiable one by one. If a predicate is not yet satisfiable, the continuation function is delayed until this precondition is met. For this purpose, the function uses the method wait for awaiting that a particular predicate is satisfiable. Once all preconditions are met the clause is added to the least model, if it is not already present, and the predicate in question is declared non-empty[1]. This also declares the clause eligible for resolution of other clauses waiting in Phase 3. A special case applies if the transition is of the form $\mathsf{p} \to \mathsf{q}$, i.e. a *replication* rule. In this case all transitions of q are copied to p, instead of adding the replication itself, to avoid cyclic redundancy.

As already mentioned, predicate handles are sets describing an intersection of original predicates. Hence, when a predicate is checked for being satisfiable,

[1] If a clause is added for a predicate, then the predicate must be non-empty, as all intersections in the precondition are non-empty.

this intersection may not have been calculated yet. This task is performed by the functions request, inter, and unify. request ensures that each intersection is only calculated once. inter calculates the intersection, by computing the intersection of every combination of compatible transitions of the predicates. Lastly, the combination of two rules into one rule that describes their intersection is straightforwardly done by unify.

B.2 Engineering the Solver

From a theoretical point of view, the pseudo-code above is succinct and correct. But in an actual implementation, there are a few caveats that one must sidestep to ensure efficiency. We shall discuss the most important of these below.

Avoiding redundant auxiliary clauses

The function split introduces a new predicate and a new clause, for each splitting performed. This extensive introduction of auxiliary predicates and clauses can somewhat be avoided by reusing, if possible, existing clauses; e.g. it is redundant to introduce a new auxiliary predicate aux_1 and a clause $\mathsf{aux}_1(f(a, X)) \Leftarrow \mathsf{q}(X) \wedge \mathsf{r}(Y)$, if another auxiliary clause $\mathsf{aux}_2(f(a, Y)) \Leftarrow \mathsf{r}(X) \wedge \mathsf{q}(Y)$ already exists, as we may as well reuse aux_2. Notice, however, that this reuse is only safe with predicates introduced by the solver, as original predicates may exist in conclusions of more than one clause.[2]

For proper detection of equivalent clauses, the solver must be able to detect that permutated preconditions are equivalent. One option is to use sets to model preconditions and hash functions for comparison. Using hash is efficient, but risks collision. Furthermore, clauses with variables ω-renamed apart should also be detected equivalent.

Simplifying clauses upon splitting

The function split introduces copies the precondition from the original clause onto the introduced clause. However, some of the precondition may force the

[2]Implemented, but somewhat a hack. Would be nice to incorporate in the pseudo-code in some way.

```
let  add (g ⇐ ω) = add_f (g ⇐ ω) []
and rec add_f c θ =
    match c with
    | (p(X) ⇐ ϵ)                → let fun c = match c with
                                              | (θ(X)(t) ⇐ ω) → store (p(t) ⇐ ω)
                                              | _             → ()
                                  register fun;
                                  forall c ∈ φ̂ do fun c
    | (p(X1, ..., Xm) ⇐ ϵ)      →
        store (p(X1, ..., Xm) ⇐ θ(X1)(X1) ∧ ··· ∧ θ(Xm)(Xm))
    | (p(f(X1, ..., Xn)) ⇐ ϵ)   →
        store (p(f(X1, ..., Xn)) ⇐ θ(X1)(X1) ∧ ··· ∧ θ(Xn)(Xn))
    | (g ⇐ q(X) ∧ ω)            → let θ' = θ[X ↦ {q} ∪ θ(X)]
                                  in  request θ'(X);
                                      wait θ'(X) (add_f (g ⇐ ω) θ')

and request p =
    if ¬(requested p)
    then requested p := true;
         inter p
and inter p =
    match p with
    | {p1}                → ()
    | {p1, p2, ..., pk}   →
      let fun1 c = match c with
                   | ({p1}(t) ⇐ ω) → forall ({p2, ..., pk}(t') ⇐ ω') ∈ φ̂
                                     do unify c ({p2, ..., pk}(t') ⇐ ω')
                   | _             → ()
      let fun2 c = match c with
                   | ({p2, ..., pk}(t') ⇐ ω') → forall ({p1}(t) ⇐ ω) ∈ φ̂
                                                do unify c ({p1}(t) ⇐ ω)
                   | _                        → ()
      let fn () = register fun1;
                  register fun2;
                  forall c ∈ φ̂ do fun2 c
      in  request {p2, ..., pk};
          wait {p2, ..., pk} (wait {p1} fn)

let  unify c c' =
    match (c, c') with
    | ( p(f(X1, ..., Xn)) ⇐ q1(X1) ∧ ··· ∧ qn(Xn)  ,
        p'(f(Y1, ..., Yn)) ⇐ q'1(Y1) ∧ ··· ∧ q'n(Yn) )
        → add ((p ∪ p')(f(X1, ..., Xn)) ⇐ (q1 ∪ q'1)(X1) ∧ ··· ∧ (qn ∪ q'n)(Xn))
    | _ → ()
```

Figure B.4: Phase 4 - Core solver.

solver to perform redundant checks. Consider, for example, the following clause:

$$\mathsf{p}(X, f(Y)) \quad \Leftarrow \quad \mathsf{q}_1(X) \wedge \mathsf{q}_2(Y)$$

For splitting this clause, an auxiliary predicate would be introduced, and the resulting clauses would be produced:

$$\begin{aligned} \mathsf{p}(X, Z) &\Leftarrow \mathsf{q}_1(X) \wedge \mathsf{q}_2(Y), \mathsf{aux}(Z) \\ \mathsf{aux}(f(Y)) &\Leftarrow \mathsf{q}_1(X) \wedge \mathsf{q}_2(Y) \end{aligned}$$

Thus both q_1 and q_2 will be investigated twice upon solving these clauses. An optimisation to this is to only include the set of literals, that contain variables connected to variables in the conclusion of the auxiliary predicate, in the precondition of the auxiliary predicate. We may then remove the subset of these literals, that no longer contain variables connected to variables in the conclusion of the original clause, from the precondition of the original clause. Doing this, on the clauses above, yields the following optimal splitting:

$$\begin{aligned} \mathsf{p}(X, Z) &\Leftarrow \mathsf{q}_1(X) \wedge \mathsf{aux}(Z) \\ \mathsf{aux}(f(Y)) &\Leftarrow \mathsf{q}_2(Y) \end{aligned}$$

It may not be apparent why this is an optimisation to the original solver, as computing the set of variables connected in the precondition is usually much more requiring than redundant detection of satisfiability of predicates. But, in conjunction with the optimisation of §B.2, this optimisation will usually result in a drastic decrease of clauses introduced, and thus, a much more efficient resolution procedure.

Detecting Tautologies

If a precondition of a clause includes the conclusion or a literal that trivially implies the conclusion, we can safely stop further resolution.

That the conclusion is included in the precondition; i.e. if $g \in \omega$ for $g \Leftarrow \omega$, is easily detected. Detecting literals that trivially implies the conclusion in the precondition, is a more vague case. However, whenever we have detected a copy-clause $\mathsf{p}(X) \Leftarrow \mathsf{q}(X)$ and we are faced with a predicate $\mathsf{p}(t) \Leftarrow \mathsf{q}(t) \wedge \omega$, we may conclude that the latter is a tautology.

Appendix C

Protocol Specifications

This appendix holds the specification of five cryptographic protocols: the amended Wide Mouthed Frog, the amended Needham-Schroeder Public Key, Yahalom, Otway-Rees, and the Andrew Secure RPC. The specification of the original Wide Mouthed Frog and Needham-Schroeder Public Key protocols have been carried out in running examples and may be found there. For succinctness the protocols are translated directly to CryptoKlaim the extended narration step is omitted.

C.1 Wide Mouthed Frog (Amended)

Lowe suggests in [72] an amended version to the Wide Mouthed Frog protocol:

$$\begin{array}{lcll} 1. & A \rightarrow S & : & A, \{B, Kab\}_{Ka} \\ 2. & S \rightarrow B & : & \{A, Kab\}_{Kb} \\ 3. & B \rightarrow A & : & \{Nb\}_{Kab} \\ 4. & A \rightarrow B & : & \{Nb + 1\}_{Kab} \end{array}$$

An exchange of a nonce is added to avoid an attack pointed out in the same article (see §6.3 for a detailed explanation of the attack).

To encode this protocol we must use both the trick mentioned in §5.6.4, although

$$
\begin{array}{lcl}
A[x^A, x^B, x^{Ka}, x^{succ}] & \triangleq & (\nu\ Kab) \\
& & \mathsf{out}\ \langle x^A, \{\!|\langle x^B, Kab\rangle|\!\}_{x^{Ka}}\rangle@\updownarrow. \\
& & \quad \mathsf{in}\ \{\!|x^{Nb}|\!\}_{Kab} \triangleleft [x^{Nb}]@\updownarrow. \\
& & \quad\quad \mathsf{out}\ \{\!|\{\!|x^{Nb}|\!\}_{x^{succ}}|\!\}_{Kab}@\updownarrow. \\
& & \quad\quad\quad A(x^A, x^B, x^{Ka}, x^{succ}) \\
S[z^A, z^B, z^{Ka}, z^{Kb}] & \triangleq & \mathsf{in}\ \langle z^A, \{\!|\langle z^B, z^{Kab}\rangle|\!\}_{z^{Ka}}\rangle \triangleleft [z^{Kab}]@\updownarrow. \\
& & \quad \mathsf{out}\ \{\!|\langle z^A, z^{Kab}\rangle|\!\}_{z^{Kb}}@\updownarrow. \\
& & \quad\quad S(z^A, z^B, z^{Ka}, z^{Kb}) \\
B[y^A, y^{Kb}, y^{succ}] & \triangleq & (\nu\ Nb) \\
& & \mathsf{in}\ \{\!|\langle y^A, y^{Kab}\rangle|\!\}_{y^{Kb}} \triangleleft [y^{Kab}]@\updownarrow. \\
& & \quad \mathsf{out}\ \{\!|Nb|\!\}_{y^{Kab}}@\updownarrow. \\
& & \quad\quad \mathsf{in}\ y' \triangleleft [y' \mapsto \{\!|\{\!|Nb|\!\}_{y^{succ}}|\!\}_{y^{Kab}}]@\updownarrow. \\
& & \quad\quad\quad B(y^A, y^{Kb}, y^{succ})
\end{array}
$$

Figure C.1: Wide Mouthed Frog (Amended) processes.

in a slightly modified version, and a technique similar to that of §3.5.2 to model $Nb+1$. Lets discuss the latter first. Modelling an increment is easy; we introduce a name $succ$, ensure that the attacker knows it (by placing it on the net $\updownarrow$), and use this to model increment of a nonce Nb as $\{\!|Nb|\!\}_{succ}$.

The other technique required is due to a more complex issue. The exchange of the nonce is introduced to ensure that K is the intended key, thus the possible value bindings of y^{Kab} (the variable B believes to hold Kab) should be guarded by this exchange. As pointed out in §5.6.4, a simple solution to this is to introduce a new variable, $y^{Kab'}$, and restrict it to the bindings of y^{Kab}. In the above protocol, however, we cannot do this, as we are not allowed to bind variables exclusively based on keys, and instead we shall bind the entire last construct $\{\!|succ(Nb)|\!\}_K$ to a variable y', if it is structured as expected. This allows us to inspect the bindings to y' and thereby conclude the possible bindings to y^{Kab}. This results in the specification of processes in Figure C.1. We also define the corresponding net:

$$
\begin{array}{rl}
N_{WMF\text{-}a} = & (\nu\ A)\ (\nu\ B)\ (\nu\ Ka)\ (\nu\ Kb)\ (\nu\ succ)\ (\\
& \quad l_A :: A(A, B, Ka, succ) \\
& \|\ \ l_S :: S(A, B, Ka, Kb) \\
& \|\ \ l_B :: B(A, Kb, succ) \\
& \|\ \ \updownarrow :: \langle succ\rangle \\
&)
\end{array}
$$

C.1.1 An attack

The attack to the amended version of the Wide Mouthed Frog uses interleaving of sessions and is as follows:

$$
\begin{array}{lll}
1. & A \rightarrow S & : \; A, \{B, Kab\}_{Ka} \\
2. & S \rightarrow B & : \; \{A, Kab\}_{Kb} \\
\mathbf{1}. & I(B) \rightarrow S & : \; B, \{A, Kab\}_{Kb} \\
\mathbf{2}. & S \rightarrow A & : \; \{B, Kab\}_{Ka} \\
\mathbf{3}. & A \rightarrow I(B) & : \; \{Nb'\}_{Kab} \\
3. & I(B) \rightarrow A & : \; \{Nb'\}_{Kab} \\
4. & A \rightarrow I(B) & : \; \{Nb' + 1\}_{Kab} \\
\mathbf{4}. & I(B) \rightarrow A & : \; \{Nb' + 1\}_{Kab}
\end{array}
$$

The attacker uses the message from the second step of the first session to initiate a new session with A. The attack now requires interleaving as the increment of the nonce must be done by someone with knowledge of Kab; i.e. by A.

C.1.2 Correcting the flaw

From above we see that the exchange of the nonce does not fix the problem, because it does not include any information that links it to the current session. This is easily solved by including B's name in the exchange:

$$
\begin{array}{lll}
1. & A \rightarrow S & : \; A, \{B, Kab\}_{Ka} \\
2. & S \rightarrow B & : \; \{A, Kab\}_{Kb} \\
3. & B \rightarrow A & : \; \{B, Nb\}_{Kab} \\
4. & A \rightarrow B & : \; \{Nb + 1\}_{Kab}
\end{array}
$$

C.2 Needham-Schroeder Public Key (Amended)

Lowe suggests in [70] an amended version to the Needham-Schroeder Public Key protocol:

$$
\begin{array}{lll}
1. & A \rightarrow B & : \; [Na, A]_{Kb+} \\
2. & B \rightarrow A & : \; [Na, Nb, B]_{Ka+} \\
3. & A \rightarrow B & : \; [Nb]_{Kb+}
\end{array}
$$

The second transmitted message is simply extended with the name of the sender to avoid an attack pointed out in the same article (see §6.3 for a detailed explanation of the attack).

$$
\begin{array}{lcl}
A[x^{A}, x^{B}, x^{Kb^{+}}, x^{Ka^{-}}] & \triangleq & (\nu\ Na) \\
 & & \mathsf{out}\ [\langle Na, x^{A}\rangle]_{x^{Kb+}}\ @\updownarrow. \\
 & & \quad \mathsf{in}\ [\![\langle Na, x^{Nb}, x^{B}\rangle]\!]_{x^{Ka-}} \triangleleft [x^{Nb}]@\updownarrow. \\
 & & \quad\quad \mathsf{out}\ [x^{Nb}]_{x^{Kb+}}\ @\updownarrow. \\
 & & \quad\quad\quad A(x^{A}, x^{B}, x^{Kb^{+}}, x^{Ka^{-}}) \\
B[y^{A}, y^{B}, y^{Kb^{-}}, y^{Ka^{+}}] & \triangleq & (\nu\ Nb) \\
 & & \mathsf{in}\ [\![\langle y^{Na}, y^{A}\rangle]\!]_{y^{Kb-}} \triangleleft [y^{Na}]@\updownarrow. \\
 & & \quad \mathsf{out}\ [\langle y^{Na}, Nb, y^{B}\rangle]_{y^{Ka+}}\ @\updownarrow. \\
 & & \quad\quad \mathsf{in}\ [\![Nb]\!]_{y^{Kb-}} \triangleleft [\,]@\updownarrow. \\
 & & \quad\quad\quad B(y^{A}, y^{B}, y^{Kb^{-}}, y^{Ka^{+}})
\end{array}
$$

Figure C.2: Needham-Schroeder Public Key (amended) processes.

The modification is easily added to the specification we gave in Example 4.15, resulting process definitions of Figure C.2. The net remains unaltered:

$$
\begin{array}{l}
N_{NS\text{-}a} = \\
\quad (\nu\ A)\ (\nu\ B)\ (\nu^{\pm}\ Ka)\ (\nu^{\pm}\ Kb)\ (\\
\qquad\quad l_A :: A(A, B, Kb^{+}, Ka^{-}) \\
\qquad \| \quad l_B :: B(A, B, Kb^{-}, Ka^{+}) \\
\quad)
\end{array}
$$

C.3 Yahalom

The Yahalom protocol is, as already described in Example 1.1, given by the following Alice-Bob narration:

$$
\begin{array}{lllll}
1. & A \to B & : & A, Na \\
2. & B \to S & : & B, \{A, Na, Nb\}_{Kb} \\
3. & S \to A & : & \{B, Kab, Na, Nb\}_{Ka}, \{A, Kab\}_{Kb} \\
4. & A \to B & : & \{A, Kab\}_{Kb}, \{Nb\}_{Kab}
\end{array}
$$

This is directly translatable into CryptoKlaim, no encoding tricks requires, resulting in the processes of Figure C.3 and the net definition:

$$
\begin{array}{lll}
N_{Ya} = & (\nu\ A)\ (\nu\ B)\ (\nu\ Ka)\ (\nu\ Kb)\ (\\
 & \qquad l_A :: A(A, B, Ka) \\
 & \| \quad l_S :: S(A, B, Ka, Kb) \\
 & \| \quad l_B :: B(A, B, Kb) \\
 &)
\end{array}
$$

$$
\begin{array}{lcl}
A[x^A, x^B, x^{Ka}] & \triangleq & (\nu\ Na) \\
& & \mathsf{out}\ \langle x^A, Na \rangle @ \updownarrow. \\
& & \quad \mathsf{in}\ \langle \{\!| \langle x^B, x^{Kab}, Na, x^{Nb} \rangle |\!\}_{x^{Ka}}, x^{Enc} \rangle \triangleleft [x^{Kab}, x^{Nb}, x^{Enc}] @ \updownarrow. \\
& & \quad\quad \mathsf{out}\ \langle x^{Enc}, \{ x^{Nb} \}_{x^{Kab}} \rangle @ \updownarrow. \\
& & \quad\quad\quad A(x^A, x^B, x^{Ka}) \\
S[z^A, z^B, z^{Ka}, z^{Kb}] & \triangleq & (\nu\ Kab) \\
& & \mathsf{in}\ \langle z^B, \{\!| \langle z^A, z^{Na}, z^{Nb} \rangle |\!\}_{z^{Kb}} \rangle \triangleleft [z^{Na}, z^{Nb}] @ \updownarrow. \\
& & \quad \mathsf{out}\ \langle \{ \langle z^B, Kab, z^{Na}, z^{Nb} \rangle \}_{z^{Ka}}, \{ \langle z^A, Kab \rangle \}_{z^{Kb}} \rangle @ \updownarrow. \\
& & \quad\quad S(z^A, z^B, z^{Ka}, z^{Kb}) \\
B[y^A, y^B, y^{Kb}] & \triangleq & (\nu\ Nb) \\
& & \mathsf{in}\ \langle y^A, y^{Na} \rangle \triangleleft [y^{Na}] @ \updownarrow. \\
& & \quad \mathsf{out}\ \langle y^B, \{ \langle y^A, y^{Na}, Nb \rangle \}_{y^{Kb}} \rangle @ \updownarrow. \\
& & \quad\quad \mathsf{in}\ \langle \{\!| \langle y^A, y^{Kab} \rangle |\!\}_{y^{Kb}}, \{\!| Nb |\!\}_{y^{Kab}} \rangle \triangleleft [y^{Kab}] @ \updownarrow. \\
& & \quad\quad\quad B(y^A, y^B, y^{Kb})
\end{array}
$$

Figure C.3: Yahalom processes.

C.4 Otway-Rees

The Otway-Rees protocol is given by the following Alice-Bob narration:

$$
\begin{array}{lll}
1. & A \rightarrow B & : \ M, A, B, \{ Na, M, A, B \}_{Ka} \\
2. & B \rightarrow S & : \ M, A, B, \{ Na, M, A, B \}_{Ka}, \{ Nb, M, A, B \}_{Kb} \\
3. & S \rightarrow B & : \ M, \{ Na, Kab \}_{Ka}, \{ Nb, Kab \}_{Kb} \\
4. & B \rightarrow A & : \ M, \{ Na, Kab \}_{Ka}
\end{array}
$$

Here M is the session number, a unique nonce for each session. Notice that B cannot decrypt the encrypted component in step 1, but must forward it unverified. Similarly, in step 3, B cannot decrypt the first encrypted component and must also forward this. The protocol is then translated into CryptoKlaim in Figure C.3 given the following net definition:

$$
\begin{array}{rl}
N_{OR} = & (\nu\ A)\ (\nu\ B)\ (\nu\ Ka)\ (\nu\ Kb)\ (\nu\ M)\ (\\
& \quad l_A :: A(A, B, Ka, M) \\
& \| \quad l_S :: S(A, B, Ka, Kb, M) \\
& \| \quad l_B :: B(A, B, Kb, M) \\
&)
\end{array}
$$

$$
\begin{array}{lcl}
A[x^A, x^B, x^{Ka}, x^M] & \triangleq & (\nu\ Na) \\
& & \mathsf{out}\ \langle x^M, x^A, x^B, \{\!|\langle Na, x^M, x^A, x^B\rangle|\!\}_{x^{Ka}}\rangle@\updownarrow. \\
& & \quad \mathsf{in}\ \langle x^M, \{\!|\langle Na, x^{Kab}\rangle|\!\}_{x^{Ka}}\rangle \triangleleft [x^{Kab}]@\updownarrow. \\
& & \quad\quad A(x^A, x^B, x^{Ka}, x^M) \\
S[z^A, z^B, z^{Ka}, z^{Kb}, z^M] & \triangleq & (\nu\ Kab) \\
& & \mathsf{in}\ \langle z^M, \{\!|\langle z^{Na}, z^M, z^A, z^B\rangle|\!\}_{z^{Ka}}, \\
& & \qquad \{\!|\langle z^{Nb}, z^M, z^A, z^B\rangle|\!\}_{z^{Kb}}\rangle \triangleleft [z^{Na}, z^{Nb}]@\updownarrow \\
& & \quad \mathsf{out}\ \langle z^M, \{\langle z^{Na}, Kab\rangle\}_{z^{Ka}}, \{\langle z^{Nb}, Kab\rangle\}_{z^{Kb}}\rangle@\updownarrow. \\
& & \quad\quad S(z^A, z^B, z^{Ka}, z^{Kb}, z^M) \\
B[y^A, y^B, y^{Kb}, y^M] & \triangleq & (\nu\ Nb) \\
& & \mathsf{in}\ \langle y^M, y^A, y^B, y^{Enc}\rangle \triangleleft [y^{Enc}]@\updownarrow. \\
& & \quad \mathsf{out}\ \langle y^M, y^{Enc}, \{\langle Nb, y^M, y^A, y^B\rangle\}_{y^{Kb}}\rangle@\updownarrow. \\
& & \quad\quad \mathsf{in}\ \langle y^M, y^{Enc'}, \{\!|\langle Nb, y^{Kab}\rangle|\!\}_{y^{Kb}}\rangle \triangleleft [y^{Enc'}, y^{Kab}]@\updownarrow. \\
& & \quad\quad\quad \mathsf{out}\ \langle y^M, y^{Enc'}\rangle@\updownarrow. \\
& & \quad\quad\quad B(y^A, y^B, y^{Kb}, y^M)
\end{array}
$$

Figure C.4: Otway-Rees processes.

C.5 Andrew Secure RPC

The Andrew Secure RPC protocol was introduced in [117]:

$$
\begin{array}{lll}
1. & A \rightarrow B & : \quad A, \{Na\}_{Kab} \\
2. & B \rightarrow A & : \quad \{Na+1, Nb\}_{Kab} \\
3. & A \rightarrow B & : \quad \{Nb+1\}_{Kab} \\
4. & B \rightarrow A & : \quad \{Kab', Nb'\}_{Kab}
\end{array}
$$

The protocol assumes that A and B already shares Kab and wants to establish a new key Kab'. The nonce Nb' is supposed to be used for a future protocol execution. The encoding in CryptoKlaim is straightforward using the techniques described for the amended version of Wide Mouthed Frog in §C.1.

The encoding in CryptoKlaim is given in Figure C.5 and should be associated with the following net:

$$
\begin{array}{rl}
N_{AS} = & (\nu\ A)\ (\nu\ Kab)\ (\nu\ succ)\ (\\
& \qquad l_A :: A(A, Kab, succ) \\
& \quad \| \quad l_B :: B(A, Kab, succ) \\
& \quad \| \quad \updownarrow :: \langle succ\rangle \\
&)
\end{array}
$$

$$
\begin{array}{lcl}
A[x^{A}, x^{Kab}, x^{succ}] & \triangleq & (\nu\ Na) \\
 & & \mathsf{out}\ \langle x^{A}, \{Na\}_{x^{Ka}}\rangle@\updownarrow. \\
 & & \quad \mathsf{in}\ \{\!|\{\!|\langle Na, x^{Nb}\rangle|\!\}_{x^{succ}}|\!\}_{x^{Kab}} \triangleleft [x^{Nb}]@\updownarrow. \\
 & & \quad\quad \mathsf{out}\ \{\{x^{Nb}\}_{x^{succ}}\}_{x^{Kab}}@\updownarrow. \\
 & & \quad\quad\quad \mathsf{in}\ \{\!|\langle x^{Kab'}, x^{Nb'}\rangle|\!\}_{x^{Kab}} \triangleleft [x^{Kab'}, x^{Nb'}]@\updownarrow. \\
 & & \quad\quad\quad\quad A(x^{A}, x^{Kab}, x^{succ}) \\
B[y^{A}, y^{Kab}, y^{succ}] & \triangleq & (\nu\ Nb)\ (\nu\ Kab')\ (\nu\ Nb') \\
 & & \mathsf{in}\ \langle y^{A}, \{\!|y^{Na}|\!\}_{y^{Kab}}\rangle \triangleleft [y^{Na}]@\updownarrow. \\
 & & \quad \mathsf{out}\ \{\{\langle y^{Na}, Nb\rangle\}_{y^{succ}}\}_{y^{Kab}}@\updownarrow. \\
 & & \quad\quad \mathsf{in}\ \{\!|\{\!|Nb|\!\}_{x^{succ}}|\!\}_{y^{Kab}} \triangleleft [\,]@\updownarrow. \\
 & & \quad\quad\quad \mathsf{out}\ \{\langle Kab', Nb'\rangle\}_{y^{Kab}}@\updownarrow. \\
 & & \quad\quad\quad\quad B(y^{A}, y^{Kab}, y^{succ})
\end{array}
$$

Figure C.5: Andrew Secure RPC processes.

Bibliography

[1] F. 197. Advanced encryption standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce, National Institute for Standards and Technology, Information Technology Laboratory, 2001.

[2] F. 46. Data encryption standard. Federal Information Processing Standards Publication 46, U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, 1977. (revised versions: FIPS 46-1, 1988; FIPS 46-2, 1993; FIPS 46-3, 1999).

[3] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[4] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[5] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM.

[6] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[7] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols. *Inf. Comput.*, 148(1):1–70, 1999.

[8] M. Abadi and M. R. Tuttle. A semantics for a logic of authentication (extended abstract). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 201–216, New York, NY, USA, 1991. ACM.

[9] J. Alves-foss. Multi-protocol attacks and the public key infrastructure. In *In Proc. National Information System Security Conference*, pages 566–576, 1998.

[10] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. Mcmillan. An analysis of sat-based model checking techniques in an industrial environment. In *CHARME*, pages 254–268, 2005.

[11] A. Armando and L. Compagna. An optimized intruder model for sat-based model-checking of security protocols. In A. Armando and L. Viganò, editors, *Electronic Notes in Theoretical Computer Science*, volume 125, pages 91–108. Elsevier Science Publishers, March 2005.

[12] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.

[13] D. Basin, S. Mödersheim, and L. Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.

[14] J. D. C. Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, 1987.

[15] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[16] G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM Press.

[17] L. Bettini, V. Bono, R. D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing: Programming Environments, Languages, Security and Analysis of Systems, volume 2874 of LNCS*, pages 88–150. Springer-Verlag, 2003.

[18] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based web services security. *ACM Trans. Program. Lang. Syst.*, 30(6):1–59, 2008.

[19] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.

[20] B. Blanchet. From secrecy to authenticity in security protocols. In *In 9th International Static Analysis Symposium (SAS'02)*, pages 342–359. Springer, 2002.

[21] A. Bleeker and L. Meertens. A semantics for ban logic. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, New Brunswick, NJ, 1997.

[22] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Automatic validation of protocol narration. In *CSFW*, pages 126–140. IEEE Computer Society, 2003.

[23] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis can find new flaws too. In *Proceedings of Workshop on Issues in the Theory of Security (WITS 04)*, page 2004. Elsevier, 2004.

[24] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[25] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis for the pi-calculus. In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1998.

[26] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In V. E. Malyshkin, editor, *PaCT*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2001.

[27] S. Briais and U. Nestmann. A formal semantics for protocol narrations. In Nicola and Sangiorgi [85], pages 163–181.

[28] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[29] M. Buchholtz. Automated analysis of infinite scenarios. In Nicola and Sangiorgi [85], pages 334–352.

[30] M. Buchholtz, H. R. Nielson, and F. Nielson. A calculus for control flow analysis of security protocols. *Int. J. Inf. Sec.*, 2(3-4):145–167, 2004.

[31] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[32] C. Caleiro, L. Viganò, and D. A. Basin. Deconstructing alice and bob. *Electr. Notes Theor. Comput. Sci.*, 135(1):3–22, 2005.

[33] C. Caleiro, L. Viganò, and D. A. Basin. Relating strand spaces and distributed temporal logic for security protocol analysis. *Logic Journal of the IGPL*, 13(6):637–663, 2005.

[34] C. Caleiro, L. Viganò, and D. A. Basin. On the semantics of alice&bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.

[35] A. Church. An unsolvable problem of elementary number theory. In *American Journal of Mathematics*, volume 58, pages 345–363, 1936.

[36] E. M. Clarke, S. Jha, and W. R. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In D. Gries and W. P. de Roever, editors, *PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 87–106. Chapman & Hall, 1998.

[37] E. M. Clarke, S. Jha, and W. R. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.

[38] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2003.

[39] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. *Sci. Comput. Program.*, 50(1-3):51–71, 2004.

[40] L. F. Cranor and R. K. Cytron. Design and Implementation of a Practical Security-Conscious Electronic Polling System. Research Report WUCS-96-02, Department of Computer Science, Washington University, 1996.

[41] Z. Dang and R. A. Kemmerer. Using the astral model checker for cryptographic protocol analysis. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, pages 132–141, 1997.

[42] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, April 2002.

[43] T. Dierks and C. Allen. The tls protocol version 1.0. Technical report, RFC Editor, United States, 1999.

[44] D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[45] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. 1999.

[46] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces. Technical Report Research Report 67, The MITRE corporation, 1997.

[47] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1), 1999.

[48] L. Falk, A. Prakash, and K. Borders. Analyzing websites for user-visible security design flaws. In *SOUPS '08: Proceedings of the 4th symposium on Usable privacy and security*, pages 117–126, New York, NY, USA, 2008. ACM.

[49] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1791–1849. Elsevier and MIT Press, 2001.

[50] A. Fujioka, T. Okamoto, and K. Ohta. A Practical Secret Voting Scheme for Large Scale Elections. *Lecture Notes in Computer Science: Advances in Cryptology - AUSCRYPT '92*, 718:244–251, 1992.

[51] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[52] H. Gao, C. Bodei, P. Degano, and H. R. Nielson. A formal analysis for capturing replay attacks in communication protocols. In *12th Annual Asian Computing Science Conference: Focusing on Computer and Network Security*, Lecture Notes in Computer Science, dec 2007.

[53] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[54] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.

[55] D. Gollmann. *Computer security*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[56] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. pages 234–248. IEEE Computer Society Press, 1990.

[57] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, 2003.

[58] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003.

[59] J. Goubault-Larrecq. Deciding h_1 by resolution. *Inf. Process. Lett.*, 95(3):401–408, 2005.

[60] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real c code. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2005.

[61] P. Halmos. *Naive Set Theory*. Van Nostrand, 1960. Reprinted by Springer-Verlag, Undergraduate Texts in Mathematics, 1974.

[62] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[63] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, 1978.

[64] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.

[65] K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 180–199, London, UK, 2000. Springer-Verlag.

[66] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459, RFC Editor, United States, 1999.

[67] I. S. S. Institute. Implementing software inspections. Technical report, IBM, 1981.

[68] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In *In Proc. 1997 Security Protocols Workshop*, pages 91–104. Springer-Verlag, 1997.

[69] R. A. Kemmerer, C. Meadows, and J. K. Millen. Three system for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[70] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[71] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAS'96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[72] G. Lowe. A family of attacks upon authentication protocols. Technical Report 1997/5, Department of Mathematics and Computer Science, University of Leicester, 1997.

[73] G. Lowe and B. Roscoe. Using csp to detect errors in the tmn protocol. *Software Engineering, IEEE Transactions on*, 23(10):659–669, 1997.

[74] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[75] W. Mao and C. Boyd. Towards formal analysis of security protocols. In *In Computer Security Foundations Workshop VI*, pages 147–158. IEEE Computer Society Press, 1993.

[76] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[77] C. Meadows. A cost-based framework for analysis of denial of service networks. *Journal of Computer Security*, 9(1/2):143–164, 2001.

[78] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[79] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[80] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, fifth edition edition, 1999.

[81] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts i and II. Technical Report -86, 1989.

[82] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur/spl phi/. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1997. IEEE Computer Society.

[83] S. Nanz and C. Hankin. A framework for security analysis of mobile wireless networks. *Theor. Comput. Sci.*, 367(1):203–227, 2006.

[84] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[85] R. D. Nicola and D. Sangiorgi, editors. *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, volume 3705 of *Lecture Notes in Computer Science*. Springer, 2005.

[86] C. R. Nielsen, E. H. Andersen, and H. R. Nielson. Static validation of a voting protocol. *Electr. Notes Theor. Comput. Sci.*, 135(1):115–134, 2005.

[87] C. R. Nielsen, F. Nielson, and H. R. Nielson. Cryptographic pattern matching. *Electr. Notes Theor. Comput. Sci.*, 168:91–107, 2007.

[88] C. R. Nielsen, F. Nielson, and H. R. Nielson. Iterative specialisation of horn clauses. In S. Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2008.

[89] C. R. Nielsen and H. R. Nielson. Static analysis for blinding. *Nord. J. Comput.*, 13(1-2):98–116, 2006.

[90] F. Nielson, R. R. Hansen, and H. R. Nielson. Abstract interpretation of mobile ambients. *Sci. Comput. Program.*, 47(2-3):145–175, 2003.

[91] F. Nielson and H. R. Nielson. Flow logic and operational semantics. *Electr. Notes Theor. Comput. Sci.*, 10, 1997.

[92] F. Nielson, H. R. Nielson, J. Bauer, C. R. Nielsen, and H. Pilegaard. Relational analysis for delivery of services. In G. Barthe and C. Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2007.

[93] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[94] F. Nielson, H. R. Nielson, and R. R. Hansen. Validating firewalls using flow logics. *Theor. Comput. Sci.*, 283(2):381–418, 2002.

[95] F. Nielson, H. R. Nielson, and H. Pilegaard. What is a free name in a process algebra? *Inf. Process. Lett.*, 103(5):188–194, 2007.

[96] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In M. V. Hermenegildo and G. Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2002.

[97] F. Nielson, H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl. The succinct solver suite. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2004.

[98] H. R. Nielson and F. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL*, pages 332–345, 1997.

[99] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL*, pages 142–154, 2000.

[100] H. R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer, 2002.

[101] H. R. Nielson and F. Nielson. Data flow analysis for ccs. In Reps et al. [112], pages 311–327.

[102] H. R. Nielson, F. Nielson, and H. Pilegaard. Flow logic for process calculi: A tutorial. Manuscript, to appear.

[103] D. J. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[104] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[105] L. C. Paulson. Relations between secrets: Two formal analyses of the yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.

[106] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[107] H. Pilegaard. *Language Based Techniques for Systems Biology*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2007.

[108] H. Pilegaard, F. Nielson, and H. R. Nielson. Context dependent analysis of bioambients. In R. Giacobazzi and F. Ranzato, editors, *Emerging Aspects of Abstract Interpretation 2006*, mar 2006. Printed in informal proceedings only.

[109] H. Pilegaard, F. Nielson, and H. R. Nielson. Pathway analysis for bioambients. *Journal of Logic and Algebraic Programming*, 77(1-2):92 – 130, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[110] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[111] C. W. Probst, R. R. Hansen, and F. Nielson. Where can an insider attack? In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, and S. A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 4691 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2006.

[112] T. W. Reps, M. Sagiv, and J. Bauer, editors. *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*. Springer, 2007.

[113] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[114] A. W. Roscoe. Model-checking csp. pages 353–378, 1994.

[115] P. Ryan and S. A. Schneider. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley Professional, 2001.

[116] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[117] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7:247–280, 1989.

[118] H. Seidl and K. N. Verma. Flat and one-variable clauses: Complexity of verifying cryptographic protocols with single blind copying. *CoRR*, abs/cs/0511014, 2005.

[119] H. Seidl and K. N. Verma. Cryptographic protocol verification using tractable classes of horn clauses. In Reps et al. [112], pages 97–119.

[120] D. X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[121] R. R. Stoll. *Set Theory and Logic*. Dover Publications, New York, 1979.

[122] P. Syverson. A taxonomy of replay attacks. In *In Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 187–191. Society Press, 1994.

[123] S.-Å. Tärnlund. Horn clause computability. *BIT*, 17(2):215–226, 1977.

[124] P. Taylor. *Practical Foundations of Mathematics (Cambridge Studies in Advanced Mathematics)*. Cambridge University Press, May 1999.

[125] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.