Dottorato di Ricerca
in Informatica
Ciclo XXIII
(A.A. 2010 - 2011)

Verified Security Protocol Modeling and Implementation with AnBx

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di dottorato di Paolo Modesti, matricola 955513

Coordinatore del Dottorato	**Tutore del dottorando**
Prof. Antonino Salibra	**Prof. Michele Bugliesi**

Author's e-mail: modesti@dsi.unive.it

Author's address:

Dipartimento di Scienze Ambientali, Informatica e Statistica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: `http://www.dsi.unive.it`

Abstract

AnBx is an extension of the *Alice & Bob* notation for protocol narrations to serve as a specification language for a purely declarative modelling of distributed protocols. *AnBx* is built around a set of communication and data abstractions which provide primitive support for the high-level security guarantees, and help shield from the details of the underlying cryptographic infrastructure.

Being implemented on top of the OFMC verification tool, *AnBx* serves not only for specification and design, but also for security analysis of distributed protocols. Moreover the framework, keeping apart the protocol logic from the application logic, allow for automatic generation of Java source code of protocols specified in *AnBx*.

We demonstrate the practical effectiveness of our approach with the specification and analysis of two real-life e-payment protocols, obtaining stronger and more scalable security guarantees than those offered by the original ones.

In the second part of the thesis we formally analyze the Secure Vehicle Communication system (SeVeCom), using the AIF framework which is based on a novel set-abstraction technique. We report on two new attacks found and verify that under some reasonable assumptions, the system is secure.

Acknowledgments

First of all, I would like to express my deep gratitude to my supervisor Prof. Michele Bugliesi for his precious guide and encouragement over these years. He gave me the opportunity to enter into the research world, advancing professionally and personally. He also taught me how to develop my ideas in an effective manner, providing very useful advice during my Ph.D. course.

I would like to thank my external reviewers Prof. Luca Viganò and Prof. Sebastian Mödersheim for their time spent carefully reading the thesis and for their useful comments and suggestions.

In particular, I would like to show my gratitude to Sebastian Mödersheim. Together with Prof. Hanne Riis Nielson and Prof. Flemming Nielson, he allowed me to carry out my research within the Language-Based Technology Group at the Technical University of Denmark (DTU) where I spent three intensive months of work. The outcome was the joint paper on SeVeCom with Prof. Mödersheim, who has been very helpful discussing research topics and giving me advice in many occasions even after the visit at DTU.

I would like also to thank my colleagues at Ca' Foscari University Venezia, especially Stefano Calzavara, who was always ready to discuss interesting research topics, and for his valuable support and comments.

Contents

List of Figures

List of Tables

Introduction

An increasing number of cyber attacks is targeting software at the application layer. According to the "*IBM X-Force Trend and Risk Report 2009*" [46] a large majority of attacks (at least 75 percent) affects the application layer where customer information, credit card numbers and other valuable data resides.

Most attacks are exploiting software vulnerabilities which are well known. Experts from more than 30 security organizations jointly issued a list of the top 25 most dangerous programming errors [35]. Such errors can lead to severe software vulnerabilities which are often relatively easy to find, and easy to exploit. The list can help programmers to identify and avoid common mistakes that may occur before software is even released. In fact the same report includes a list of "*Monster Mitigations*" suggesting practices and techniques considered effective in eliminating or reducing the severity of the impact of the programming faults.

This is aimed to show the way of making software more secure. However many software developers are not educated in secure programming techniques. Moreover some of the recommendations are more related with the process of software development than with the specific technical issues.

Therefore there is a strong need to find solutions to proactively identify security weaknesses in the early phases of the software development life cycle. Reviewing and fixing the source code once in production becomes increasingly complex and time consuming.

From the point of view of the software developer, designing distributed protocols is challenging, as it requires actions at very different levels: from the choice of network-level mechanisms to protect the exchange of sensitive data, to the definition of structured interaction patterns to convey application-specific guarantees. These complex requirements may lead to a strong intertwining between the intended logic of the protocol and the low-level cryptographic mechanisms which are necessary to enforce the desired security properties, thus cluttering the design and undermining the scalability and robustness of the resulting protocol.

To counter these problems, in the first part of the thesis (Chapters 1–4), we propose an extension of the *Alice & Bob* notation for protocol narrations (*AnBx*) to serve as a specification language for a purely declarative modelling of distributed protocols. *AnBx* is built around a set of communication and data abstractions which provide primitive support for the high-level security guarantees required in the design of distributed protocols, and help shield the specification from the details of the underlying cryptographic infrastructure. *AnBx* is implemented on top of the OFMC [11] verification tool, by means of a translation to the *AnB* language [59] supported by OFMC. As a result, *AnBx* serves not only for specification and design, but also as a powerful tool for the security analysis of distributed protocols.

In chapters 3 and 4 we present a tool for the automatic generation of the Java source code of security protocols specified in *AnBx*. Extending a previous work of Briais and Nestmann [25], we generate an optimized executable narration (Chapter 3), which includes the checks on reception derivable from the static information. Our optimization improves the protocol execution speed, for example avoiding repeating the same cryptographic operations on the same data.

The generation of the source code (Chapter 4) is designed keeping apart the protocol logic from the application logic. This makes the proposed strategy suitable not only for Java but also for other object-oriented or procedural languages. A new Java security library (Section 4.2), shielding the communication and cryptographic details, provides run time support to the generated application.

We demonstrate the practical effectiveness of our approach with the specification and analysis of two real-life e-payment protocols: *iKP* and *SET* (Chapter 2). The declarative nature of the *AnBx* abstractions pays off, and results in protocol specifications with stronger, and more scalable security guarantees than those offered by the original protocols.

In the second part of the thesis (Chapter 5) we focus only on protocol verification. We formally analyze the Secure Vehicle Communication system developed by the EU-project SeVeCom, using the AIF framework [60] which is based on a novel set-abstraction technique. The model involves the hardware security modules (HSMs) of a number of cars, a certification authority, and the protocols executed between them. Each participant stores a database of keys that can be added or deleted depending on the different operations. The AIF-framework allows us to model and automatically analyze such databases without bounding the number of steps that the system can make and, in contrast to previous approaches in protocol abstraction, can handle databases that do not monotonically grow (and thus allow for revocation of keys). We report on two new attacks found and verify that under some reasonable assumptions, the system is secure in a black-box cryptography model.

Some portions of the thesis include material previously published and they are here revised and extended. In particular, chapters 1 and 2 extend [29], a joint work with Michele Bugliesi presented at ARSPA-WITS 2010 with the additional contribution of Stefano Calzavara and Sebastian Mödersheim. A journal version will be soon submitted [27]. Chapter 5 extends a paper [61] with Sebastian Mödersheim presented at IWCMC 2011.

In details, my original contribution to Chapter 1 includes the definition of *AnBx* as an extension of the existing *AnB* language, the introduction of the notion of forwarding channels, the first translations from *AnBx* to *AnB* as they were outlined in [29]. Additionally, I fully developed the *AnBx* compiler and analyzed the case studies in Chapter 2. Chapters 3 and 4 include previously unpublished material describing the code generator tool I designed and implemented (excluding of course the credited ideas and components borrowed from the existing tools that were integrated in the *AnBx* compiler). Finally, in Chapter 5 my contribution consists in building the comprehensive models of the SeVeCom architecture, carrying out the related experiments, as well as proposing a new way to integrate the life-cycle of keys into an approach that actually abstracts from time. Lastly, I designed and implemented some new features of the AIF framework aimed to make easier modeling complex protocols such as SeVeCom.

I

AnBx

1

The AnBx language

1.1 Introduction

Providing security for distributed applications is challenging, as it requires careful design decisions and subtle implementation choices, at all level of the deployment stack: from the definition of structured, application-specific measures to enforce the high-level invariants of the application, to the choice of appropriate network-level primitives to support those invariants by protecting the exchange of sensitive data.

In the literature on security protocols, the *Alice & Bob* notation, also known as *protocol narrations*, has long been a popular device for the specification of distributed interactions (see e.g., [53]) and has enabled the development of a number of frameworks for formal analysis [37, 47, 52]. In such frameworks, the semantics of the specification languages are defined by translation into lower level formats amenable to model checking and automated verification. Besides making formal verification possible, these translation semantics provide for a clean separation between the abstract specification of the protocol structure, and the details of its implementation, which may be generated directly from the specification [30, 48, 58, 73]. Indeed, this separation has beneficial impact on both specification and implementation: on the one side, it helps focusing on the design of the application-level properties, staying away from unnecessary low-level details; on the other, it contributes to improve the implementation and to ensure the protocol end-to-end security, by delegating to the compiler the selection of the most adequate, core implementation components.

Traditionally, protocol narrations have been employed for relatively small systems, such as key-exchange or authentication protocols, for which the abstractions provided by narrations allow the designer to focus on the flow and structure of the messages to be exchanged, leaving it to the implementer to make decisions on subtle details as key-length, nonce generation, choice of time-stamp windows, data redundancy for decryption, and so on. More powerful mechanisms, based on high-level *channel abstractions* have been advocated by many authors for a principled specification and design of complex protocols such as those required to support modern web services, payment systems and authorization platforms.

The idea behind the notion of channel abstractions is to provide a means to design and describe the application oblivious to the underlying cryptography by relying on the concept of channel as a communication medium protected against certain attacks (e.g., on confidentiality). *How* these properties are actually ensured is a different step of the design (and might not be a concern of the application designer at all). Several papers in the literature have taken this approach, and developed it along different directions. First, there are papers that discuss the definition and implementation of different channel types, based on cryptographic realizations

and interaction patterns [2, 5, 17, 33, 39]. Other papers opt for a more abstract characterization by defining the ideal behavior of a channel [7, 28, 50, 63], and devising frameworks for analysis directly in terms of that characterization. [76] considers a refinement approach where protocols are specified by means of abstract channels that are later replaced with concrete cryptographic mechanisms. [55] takes a very different perspective on the concept of channels, by means of a calculus to describe what new channels can be obtained from currently available channels, keys, and trust assumptions. This has inspired the work in [63] on compositional reasoning for channels, and its integration into the language *AnB* with the bullet-notation distinctive of [55]. The idea of protocol implementation stack has similarly been studied in [43] and a related compositionality result is found in [31].

Following this line of research, in the present chapter we develop the notion of channel one step further, and generalize it to capture *forwarding channels*: the corresponding, new communication primitive applies to any kind of forwarding of messages, where all or some of the properties of the original transmission are preserved. We develop the novel abstractions as part of *AnBx*, a specification language that we introduce in this chapter by extending the *AnB* language from [59].

We develop the semantics of *AnBx* by conservatively integrating the generalized notion of channel into the existing semantic characterization for *AnB*. In this characterization, we provide both an abstract interpretation of channels, in terms of their ideal behavior, and a concrete interpretation that, in turn, yields a simple, yet analysis-effective cryptographic implementation. Both interpretations are based on a translation to the AVISPA Intermediate Format, which makes *AnBx* amenable for verification within the AVISPA model-checking platform based on OFMC [11, 62]. In addition to the IF translation, we also define alternative cryptographic implementations, to be employed for practical deployment of *AnBx* as a protocol development tool (Chapters 3 and 4).

AnBx constitutes a powerful specification language for distributed applications, which generalizes its predecessors in the literature. We demonstrate the practical effectiveness of our approach by carrying out as case studies two real-life e-payment protocols: the *iKP* e-payment protocol family (internet keyed payment protocol [15, 16]), and the *SET* purchase protocol (Secure Electronic Transaction [12–14]). Although we can directly formulate them in all their complexity in *AnBx*, our concept of channels with forwarding allows for factoring out the cryptographic aspects almost entirely. The resulting protocols are more concise, easier to understand and, interestingly, more efficient to verify than the original versions. In addition, the *AnBx* formulation changes the protocols slightly, giving them a more systematic structure and actually improving them. Indeed, our *AnBx* versions of the protocols outperform the original protocols, in that they satisfy stronger security goals and properties. This is largely a consequence of the declarative nature of the specification style supported by *AnBx*: being defined as channel-level abstractions, the *AnBx* primitives convey protection on *all* message components, not just on some components as in the original specifications, yielding stronger encapsulation mechanisms, and, consequently, stronger and more scalable security guarantees. As a byproduct of our comparative analysis, we also found a (to the best of our knowledge) new flaw in the original specification of $\{2,3\}KP$, and propose an amended version that rectifies the problem.

Plan of the chapter Section 1.2 introduces *AnBx*, outlining its main features and its relationships with previous specification languages. Section 1.3 focuses on the semantics

Protocol : *Diffie-Hellman*
Types :
  Agent A, B;
  Number g, X, Y, Msg;
  Function exp;
Knowledge :
  A : $A, B, g,$ exp;
  B : $A, B, g,$ exp;
Actions :
$$A \rightarrow B : (A \mid B \mid -) : \mathsf{exp}(g, X)$$
$$B \rightarrow A : (B \mid A \mid -) : \mathsf{exp}(g, Y)$$
$$A \rightarrow B : (- \mid - \mid -) : \{\!|A, Msg|\!\}_{\mathsf{exp}(\mathsf{exp}(g,Y),X)}$$
Goals :
  B authenticates A on Msg
  Msg secret between A, B

Figure 1.1: Diffie-Hellman specification in *AnBx*.

characterization, which we give in terms of two translations to the AVISPA Intermediate Format, to serve as the basis for the implementation of the *AnBx* analytical tool. Section 1.4 outlines an alternative cryptographic translation based on a public-key infrastructure and aimed to protocol synthesis. Sections 2.1 to Section 2.3 report the results of our case studies on the *iKP* and *SET* protocols. A separate Appendix collects the *AnBx* scripts employed in the case studies. The *AnBx* implementation, together with the analytical tool employed in the case studies, is available at the following URL: `http://www.dsi.unive.it/~modesti/anbx/`.

1.2 *AnBx* Protocol Specifications

AnBx is a formalism for protocol narrations which extends the familiar *Alice & Bob* notation with new support for various mechanisms for securing remote communications, based on channel abstractions to be employed for a purely declarative modelling of distributed protocols. Below, we outline the main features of the formalism, which we have implemented as an extension of the *AnB* language [59] developed within the AVISPA project [6].

1.2.1 Protocol Types and Agent Knowledge

Protocol narrations in *AnBx* are built around an underlying signature of typed identifiers that include protocol variables, constants and function symbols. Variables are noted with upper-case initials and represent values that are dynamically determined at run time, for each protocol execution. Constants, in turn, are noted by lower-case identifiers and represent values and functions that are invariant across different protocol executions. Variables of type `Agent` are *roles*: in the protocol specification above, we have the roles A and B, which get instantiated to arbitrary concrete agents when executing the protocol. The numbers g, X

and Y, in turn, are the (constant) group generator and the (variable) random exponents of the Diffie-Hellman key exchange.

All symbols may either be public, in which case all agents may use them to construct terms, or private to some specific role. Each role is associated with an initial knowledge that defines the set of terms and function symbols known to that role. The initial knowledge, together with the terms that each role acquires along the protocol steps, determines the set of legal messages each agent may send and receive. Variables that do not occur in the initial knowledge of any role represent values that are freshly created by the agent who first uses them. In the example, X and *Msg* are created by A, while Y is created by B.

1.2.2 Protocol Actions

The core of an *AnBx* specification is the protocol narration, which describes the sequence of actions composing an ideal, unattacked run of the protocol. Every action has either of these two forms:

$$A \to B, \eta : M \qquad \text{or} \qquad A \overset{@}{\to} B, \eta : M$$

noting *standard* and *fresh* exchanges, respectively. In both cases, an agent playing role A communicates message M to the agent playing role B, along a communication channel that conveys the security guarantees specified by *mode* η. The mode η is defined as a triple of the following form:

$$\eta \ ::= \ (\langle \mathit{Auth} \rangle \,|\, \langle \mathit{Verifiers} \rangle \,|\, \langle \mathit{Conf} \rangle) \tag{1.1}$$

Each field may be set to an agent name (a list of names, for the *Verifiers* field), or unset, in which case it is filled with the distinguished symbol "−". The triple qualifies the security guarantees conveyed by the exchange involved in the action. When the *Auth* field is set, the action identifies an authentic exchange, which guarantees that the message exchanged originates from the agent named in the *Auth* field. When the *Conf* field is set, the action represents a confidential exchange, which guarantees that only the agent named in the *Conf* field is exposed to the message. As to the *Verifiers* field, it includes the list of agents entitled to verify an authentic exchange. Namely, the sender may express the intention to authenticate with many different principals, but note that this is not a list of mandatory recipients, i.e., it does not involve any form of cooperation among different agents. Authentic exchanges may further specify that the message being exchanged is *freshly* communicated by the agent referenced in the *Auth* field: the notation $A \overset{@}{\to} B, \eta : M$ serves that purpose.

The aim of the protocol modes is to abstract away from the details of how the standard security guarantees of authenticity and confidentiality may be enforced in a remote exchange. Notice, however, that *AnBx* is a conservative extension of the familiar *AnB* notation, in which abstract exchanges and cryptographic terms may freely be intermixed. Specifically, the first two actions in the Diffie-Hellman specifications employ the abstract modes to express the authentic exchanges of the two *half keys*, while the third describes the exchange of message *Msg* encrypted under the generated key.

The idea to structure protocol specifications around abstract mechanisms for secure communications and message exchanges is certainly not new. Among the various approaches in the literature, the closest to ours is the "bullet" notation supported by *AnB•* [63], which *AnBx*

conservatively extends, as detailed by the correspondence below.

	$AnB\bullet$	$AnBx$
Plain	$A \rightarrow B : M$	$A \rightarrow B : (-\,|\,-\,|\,-) : M$
Authentic	$A \bullet\!\!\rightarrow B : M$	$A \rightarrow B : (A\,|\,B\,|\,-) : M$
Confidential	$A \rightarrow\!\!\bullet B : M$	$A \rightarrow B : (-\,|\,-\,|\,B) : M$
Secure	$A \bullet\!\!\rightarrow\!\!\bullet B : M$	$A \rightarrow B : (A\,|\,B\,|\,B) : M$

In addition to the $AnB\bullet$ exchanges exhibited above, the $AnBx$ modes allow additional generality. Specifically, $AnBx$ provides primitive support for message *forwarding*, a feature which is not supported by existing proposals, and that instead constitutes a recurrent communication pattern in practical applications. We will provide examples of concrete uses of forwarding in existing protocols; for the moment, we just illustrate this functionality with some simple examples.

The first example shows how authenticity guarantees can be preserved upon forwarding:

$$\begin{array}{lcl,lcl} A & \rightarrow & B & , & (A\,|\,B,C\,|\,-) & : & M \\ B & \rightarrow & C & , & (A\,|\,B,C\,|\,-) & : & M \end{array}$$

Here, the first action denotes an authentic exchange that originates from A and is meant to be delivered to both B and C. Upon receiving M, agent B forwards it to C in the second action, preserving the authenticity guarantees by A. Notice that the mode $(A\,|\,B,C\,|\,-)$ in the second exchange still mentions A as the source of the communication, even though the message is sent by B. This pattern cannot be encoded in the $AnB\bullet$ notation, since authentic messages are always assumed to be originated by the agent specified on the tail of the arrow.

Forwarding modes can be used also to implement a form of "blind" delivery, arising when an agent relays a message that is intended to remain confidential for a third party:

$$\begin{array}{lcl,lcl} A & \rightarrow & B & , & (-\,|\,-\,|\,C) & : & M \\ B & \rightarrow & C & , & (-\,|\,-\,|\,C) & : & M \end{array}$$

Here, A confidentially sends M to C, relying on B to deliver the message. Also this protocol cannot be expressed in the $AnB\bullet$ notation, as secret messages are always intended to be disclosed to the agent specified on the head of the arrow.

Message forwarding is also available for fresh exchanges, in various combinations. Assume the message M is sent freshly from A to B:

$$A \overset{@}{\rightarrow} B \quad , \quad (A\,|\,B,C\,|\,-) \quad : \quad M$$

Then both the following actions:

$$B \rightarrow C, (A\,|\,B,C\,|\,-) : M \quad \text{and} \quad B \overset{@}{\rightarrow} C, (A\,|\,B,C\,|\,-) : M$$

are legal. With the action on the left, M is forwarded to C without any freshness guarantee, whereas the action on the right allows C to verify the freshness of the transmission.

1.2.3 Protocol Goals

AnBx protocol specifications are analyzed and validated against a set of *goals*, that specify the expected properties for the protocol. Like its predecessors *AnB* and $AnB\bullet$, our language *AnBx* supports three standard kinds of security goals:

- *Weak Authentication* goals have the form B `weakly authenticates` A `on` M and are defined in terms of non-injective agreement [51] on the runs of the protocol. Namely, this kind of guarantee does not take into account replay attacks on the protocol exchanges;
- *Authentication* goals have the form B `authenticates` A `on` M and are defined in terms of injective agreement [51], assessing the freshness of the exchange.
- *Secrecy* goals have the form M `secret between` $A_1, \ldots, A_k$ and are intended to specify which agents are entitled to learn message M at the end of the protocol run;

Additionally we introduce a new goal – A `confidentially sends` M `to` B – to model a confidential transmission from one principal to another one in which the payload is not leaked by the intruder. It is worth noting that satisfying this goal, equivalent to the *AnB* goal $A \rightarrow\bullet B : M$, is a necessary but not sufficient condition to meet the goal M `secret between` A, B.

1.3 Abstract and Cryptographic Semantics of *AnBx*

Like existing protocol narration languages, *AnBx* specifications are best characterized, semantically, in terms of the execution traces of the processes that run the specification on behalf of the protocol roles: such traces encode the sequences of messages that each process sends and receives during a computation that involves the honest participants and an intruder. Both the honest agents and the intruder can impersonate any role of the protocol across different runs.

Following previous work by Mödersheim and Viganò [63], we formalize the semantics of *AnBx* as a translation to the AVISPA Intermediate Format IF [10], a more low-level language to describe state transition systems with a precise semantics. In this translation, the execution state of a protocol is formed as the disjoint union of the sets of terms (the *knowledge*) that each of the honest agents, as well as the intruder, accumulates along a protocol run. Each action in the protocol is interpreted as a state transformer that updates the current state by upgrading the agent's knowledge with the terms extracted from the message the agent receives during that action. The transformation also updates the intruder's knowledge as a result of the message emitted in the protocol step.

As in [63], we define the translation in more steps, exploiting the existing AnB2IF compiler. The outline of the translation is as follows. Given an *AnBx* specification, we translate it into a corresponding *AnB* specification in which the *AnBx* modes are expressed as *tags* included in the messages exchanged. It is worth noting that the resulting *AnB* specification is not intended to be directly model-checked by OFMC, but it is fed to the AnB2IF compiler, which extracts from the narration the actions associated with each protocol agent, and translates them as IF rewriting rules. At this stage, the resulting IF specification rules still include the tags from the annotated *AnB* narration: a further transformation step completes the translation, producing the abstract IF specification (*absIF*) and the cryptographic IF implementation (*cryptIF*), which can be model checked by OFMC. The details of the overall translation are reported in this section.

1.3.1 From *AnBx* to *AnB*

The first step of the translation transforms each action in the *AnBx* narration into a corresponding *AnB* action, bearing additional annotations. These annotations will drive the

subsequent stages of the translation involving the Intermediate Format.

The translation is conceptually simple, though the introduction of the forward modes, and their combination with the fresh modes, leads to some complications. For example, for the fresh modes, we rely on a mechanism that is unrealistic in practice, but straightforward to handle in formal verification: namely, we assume that the sender freshly generates a new nonce, and the recipient caches every nonce it receives, telling apart fresh messages from replicas by checking whether the received nonce is in the cache[1]. Correspondingly, in case of a *forward* exchange, we reuse the same nonce generated at the step which introduced the message being forwarded. In order to identify the nonce to be reused, we need some housekeeping along the translation. Specifically, we make the translation dependent on a store ξ, keeping track of all the introduced nonces, along with additional information which is necessary to perform the translation.

We proceed as follows:

- At each fresh exchange which is not a forward, we first scan ξ to find out an unused nonce identifier and then we store in ξ a 4-tuple that associates the new nonce with additional information, which allows its reuse in any possible forwarding of the message exchanged. The tuple has the form $(A, \tilde{V}, M, N)$ and includes the name A of the source agent, the tuple $\tilde{V}$ of the intended verifiers, the message M exchanged and the nonce N generated;

- At each authentic forward action, we lookup the store in search of a tuple whose first three components match the *Auth* and the *Verifiers* components of the mode and the message being forwarded; if the tuple exists, we are forwarding the result of a fresh exchange and we include the nonce generated at that exchange among the components of the forwarded message, irrespective of whether the forward is fresh or not (this choice is technically convenient in the definition of the translation). Instead, if the tuple does not exist, then the source action must be non-fresh, thus no nonce is included in the forward of the generated message. The translation is undefined when a fresh forward is performed, but no matching tuple was found in the store.

In addition to this issue, we must take into account another complication related to blind forwarding operations. In general the recipient A of a message M may differ from the intended receiver B, so the message M should not always be exposed upon reception. We address this point by wrapping M inside the constructor blind_B, to denote that it should be readable only by B.

The translation clauses are listed below: if none of the clauses applies, the translation is undefined and an error is reported. The absence of errors at this level does not imply that the protocol is runnable, as we discuss below. When more than one entry in ξ matches the required side-conditions, we always assume to consider the most recent one for performing the translation.

[1]This simple mechanism is only dictated by practical motivations, namely contributing to keep model checking time and space complexity under control. Other, more realistic, implementations based on challenge response are of course possible, as we discuss in Section 1.4. On the other hand, we remark that concrete implementations may involve a combination of a nonce and a timestamp, so that the recipient would have to store the nonce only for the validity period of the timestamp.

$$
\begin{array}{lcll}
[\![A \to B, (- \mid - \mid -) : M]\!]_\xi & = & A \to B : \mathsf{plain}, M & \\
[\![A \to B, (- \mid - \mid \hat{B}) : M]\!]_\xi & = & A \to B : \mathsf{ctag}, \mathsf{blind}_{\hat{B}}(M) & \\
[\![A \to B, (\hat{A} \mid \tilde{V} \mid -) : M]\!]_\xi & = & A \to B : \mathsf{atag}, \hat{A}, \tilde{V}, M, N & \textit{if } \hat{A} \neq A \textit{ and } (\hat{A}, \tilde{V}, M, N) \in \xi \\
 & = & A \to B : \mathsf{atag}, \hat{A}, \tilde{V}, M & \textit{otherwise} \\
[\![A \to B, (\hat{A} \mid \tilde{V} \mid \hat{B}) : M]\!]_\xi & = & A \to B : \mathsf{stag}, \hat{A}, \tilde{V}, \hat{M}, N & \textit{if } \hat{A} \neq A \textit{ and } (\hat{A}, \tilde{V}, M, N) \in \xi \\
 & = & A \to B : \mathsf{stag}, \hat{A}, \tilde{V}, \hat{M} & \textit{otherwise} \\
 & & & \textit{where } \hat{M} = \mathsf{blind}_{\hat{B}}(M) \\
[\![A \xrightarrow{@} B, (\hat{A} \mid \tilde{V} \mid -) : M]\!]_\xi & = & A \to B : \mathsf{fresh} :: \mathsf{atag}, \hat{A}, \tilde{V}, M, N & \textit{if } \hat{A} = A \textit{ and } N \textit{ fresh in } \xi \\
 & & & \textit{or } \hat{A} \neq A \textit{ and } (\hat{A}, \tilde{V}, M, N) \in \xi \\
[\![A \xrightarrow{@} B, (\hat{A} \mid \tilde{V} \mid \hat{B}) : M]\!]_\xi & = & A \to B : \mathsf{fresh} :: \mathsf{stag}, \hat{A}, \tilde{V}, \hat{M}, N & \textit{if } \hat{A} = A \textit{ and } N \textit{ fresh in } \xi \\
 & & & \textit{or } \hat{A} \neq A \textit{ and } (\hat{A}, \tilde{V}, M, N) \in \xi, \\
 & & & \textit{where } \hat{M} = \mathsf{blind}_{\hat{B}}(M)
\end{array}
$$

Tags are public constants in the target *AnB* specification, while blind_X constructors are public function symbols, so all this information is available to every agent, including the intruder. In addition, the specification is extended with private function symbols, $\mathsf{unblind}_X$, parameterized over the agents identity, which are used to extract the confidential messages. We return on this later in this section, where we outline the workings of the IF intermediate format.

Below, we report the *AnB* actions resulting from the translation of the *AnBx* specification of the Diffie-Hellman protocol in Figure 1.1.

$$
\begin{array}{lcll}
A & \to & B & : \ \mathsf{atag}, A, B, \mathsf{exp}(g, X) \\
B & \to & A & : \ \mathsf{atag}, B, A, \mathsf{exp}(g, Y) \\
A & \to & B & : \ \mathsf{plain}, \{\!|Msg|\!\}_{\mathsf{exp}(\mathsf{exp}(g,Y),X)}
\end{array}
$$

Some applications of forward modes are shown in the following examples taken from the e-payments protocols employed as case studies. At the current stage we focus on the communication patterns, therefore we abstract from the informative content of the payload *Msg*.

Example 1 (Blind forward - Revised 1KP, steps 3a,3b) *The customer C generates a message intended to be disclosed only to the acquirer A. The message is delivered to the final recipient by relying on the merchant M, who cannot check anything on the incoming message and chiefly forwards it upon reception.*

$$
\begin{array}{lcll}
C & \to & M, & (- \mid - \mid A) : Msg \\
M & \to & A, & (- \mid - \mid A) : Msg
\end{array}
$$

$$
\begin{array}{lcll}
C & \to & M : & \mathsf{ctag}, \mathsf{blind}_A(Msg) \\
M & \to & A : & \mathsf{ctag}, \mathsf{blind}_A(Msg)
\end{array}
$$

Example 2 (Fresh forward - Revised SET, steps 5,6) *The acquirer A generates a message, which should be authenticated by both the merchant M and the customer C. The message should also be a shared secret among the three parties. Freshness is preserved upon relaying*

by M, while confidentiality guarantees are specified according to the receiver of the specific exchange.

$$\begin{array}{lcll} A & \overset{@}{\rightarrow} & M, & (A \,|\, M, C \,|\, M) : Msg \\ M & \overset{@}{\rightarrow} & C, & (A \,|\, M, C \,|\, C) : Msg \end{array}$$

$$\begin{array}{lcll} A & \rightarrow & M: & \mathsf{fresh} :: \mathsf{stag}, A, (M, C), \mathsf{blind}_M(Msg), N \\ M & \rightarrow & C: & \mathsf{fresh} :: \mathsf{stag}, A, (M, C), \mathsf{blind}_C(Msg), N \end{array}$$

Example 3 (Fresh exchange followed by a non fresh one - Revised 2KP, steps 5,6) *The acquirer A generates a message, which should be authenticated by both the merchant M and the customer C. Freshness is lost upon relaying, as every confidentiality guarantee.*

$$\begin{array}{lcll} A & \overset{@}{\rightarrow} & M, & (A \,|\, M, C \,|\, M) : Msg \\ M & \rightarrow & C, & (A \,|\, M, C \,|\, -) : Msg \end{array}$$

$$\begin{array}{lcll} A & \rightarrow & M: & \mathsf{fresh} :: \mathsf{stag}, A, (M, C), \mathsf{blind}_M(Msg), N \\ M & \rightarrow & C: & \mathsf{fresh} :: \mathsf{atag}, A, (M, C), Msg, N \end{array}$$

One of the purposes of the translational semantics we are defining is to isolate and filter out inconsistent specifications. The sources of inconsistency are of different nature, and they are captured at various steps of the translation. As we mentioned, the step we just described fails under certain conditions: indeed, these are special cases of inconsistent narrations, like the following sequence of *AnBx* actions:

$$\begin{array}{lcllcl} A & \rightarrow & B & , & (A \,|\, B, C \,|\, -) & : & M \\ B & \overset{@}{\rightarrow} & C & , & (A \,|\, B, C \,|\, -) & : & M \end{array}$$

Indeed, the two steps are inconsistent, as B is forwarding M to C as a fresh authentic message from A, whereas the first exchange does not convey any freshness guarantee. The inconsistency is detected by the translation of the forward action, as in the store there is no match for the tuple $(A, (B, C), M)$ needed to retrieve the nonce that would be required at this stage.

Further inconsistencies are captured by the AnB2IF translation, which is the step at which the actions required by the protocol agents to compose, decompose and check messages are made explicit, whenever possible, in the IF representation of the protocol. For instance, the following sequence of *AnBx* actions can be translated according to the previous rules, but the translation leads to an non-executable protocol, since B is exposed only to $\mathsf{blind}_C(M)$ and not to M.

$$\begin{array}{lcllcl} A & \rightarrow & B & , & (- \,|\, - \,|\, C) & : & M \\ B & \rightarrow & C & , & (- \,|\, - \,|\, -) & : & M \end{array}$$

While we rely on the AnB2IF compiler to accomplish this translation phase, and use it as a black box transformer, in the next section we give a brief overview of IF and of the AnB2IF translation, as they are useful to understand the details of the last step of our translation.

1.3.2 The Intermediate Format IF

The AVISPA Intermediate Format IF [10] is a low-level language for specifying transition systems in a way that is fit for protocol analysis tools. An IF specification consists of an initial

state that collects the initial knowledge of each of the protocol agents, a set of transition rules over states for the protocol participants and the intruder, and a set of *attack rules* that specify which states count as attack states. A protocol is safe when no attack state is reachable using the transition relation induced by transition and attack rules.

An IF state is a set of facts, separated by dots ('.'), which encode the knowledge of the protocol agents. We distinguish two kind of facts: $\mathsf{iknows}(m_1, \ldots, m_k)$, which expresses that the intruder knows terms $m_1, \ldots, m_k$, and $\mathsf{state}_{\mathcal{A}}(A, m'_1, \ldots, m'_n)$ which characterizes the local state of an honest agent during the protocol execution by the terms $A, m'_1, \ldots, m'_n$. The constant $\mathcal{A}$ identifies the role of the agent, and, by convention, the first message A is the name of that agent[2]. We will later introduce further kinds of facts. The *transitions* of an IF specification are of the form:

$$L \mid \mathit{Cond} \quad =[\mathcal{X}]\Rightarrow \quad R$$

where L and R are states, *Cond* a set of conditions on the terms of L, $\mathcal{X}$ a set of variables fresh in L and *Cond*, and R only contains variables from L or $\mathcal{X}$. The semantics of this rule is defined by the state transitions it allows: from a state S the rule enables a transition to a state S' iff there is a substitution σ of all rule variables such that $L\sigma \subseteq S$, $S' = (S \setminus L\sigma) \cup R\sigma$, $\mathcal{X}\sigma$ are fresh constants (that do not appear in S) and all conditions in *Cond* are satisfied under the substitution σ. These transitions may be employed to give semantics to the honest protocol participants as well as to the intruder.

Intruder Transitions The intruder transitions are defined by a set of rules like the following ones:

$$\begin{aligned} \mathsf{iknows}(M).\mathsf{iknows}(K) &\Rightarrow \mathsf{iknows}(\{M\}_K) \\ \mathsf{iknows}(\{M\}_K).\mathsf{iknows}(\mathsf{inv}(K)) &\Rightarrow \mathsf{iknows}(M) \end{aligned}$$

For instance, the first rule above describes both asymmetric encryption and signing, while the second one expresses that the payload of a ciphertext can be obtained by knowledge of the decryption key. Additional rules may be included to represent the intruder's capability to compose and decompose messages. Specifically, the intruder may use all public function symbols to form new messages, according to the following rule (where f is one such function symbol):

$$\mathsf{iknows}(M_1) \ldots \mathsf{iknows}(M_n) \Rightarrow \mathsf{iknows}(f(M_1, \ldots, M_n))$$

We assume $\mathsf{iknows}(\cdot)$ facts to be persistent, so as to model that the intruder knowledge increases monotonically, without having to repeat the left-hand side occurrences of $\mathsf{iknows}(\cdot)$ to the right for every possible transition.

Agent Transitions The transition rules for the honest protocol participants are generated from a narration by projecting the narration actions onto each protocol role so as to construct

[2]In contrast to the convention used in the *AnBx* specification, IF makes a clear distinction between role names, noted by calligraphic letters such as $\mathcal{A}$, and agent names, noted with capital letters A.

role *strands* [45] of the following form, for each role R in the protocol:

$$\begin{array}{cccc} \_ & \to & R & : & i_1 \\ R & \to & \_ & : & o_1 \\ & & \vdots & & \\ \_ & \to & R & : & i_l \\ R & \to & \_ & : & o_l \end{array}$$

If R is the sender of the first message of the protocol, then we simply take i_1 to be a dummy message, and similarly for o_l, if R is the receiver of the last message. Each pair of input-output actions in this strand is associated with an IF transition, which accounts for the state updates resulting from the reception of the input message and the transmission of output.

The precise definition of the transition rules is given in [59]. Here we provide some example that is sufficient to get the intuition and understand the rest of the presentation.

Having projected the source *AnB* narration onto each role, the core of the AnB2IF translation is in the generation of the condition set *Cond*, as these conditions formalize the precise details of how agents decompose, check and compose messages based on their current local knowledge (cf. [59]). To illustrate, below we give the IF transitions for roles A and B from the *AnB* translation of the protocol in Figure 1.1.

$$\begin{array}{l} \mathsf{state}_{\mathcal{A}}(A,B,g).\mathsf{iknows}(\cdot) \\ \quad =[X]\Rightarrow \mathsf{state}_{\mathcal{A}}(A,B,g,X).\mathsf{iknows}(\mathsf{atag},A,B,\mathsf{exp}(g,X)) \end{array}$$

$$\begin{array}{l} \mathsf{state}_{\mathcal{A}}(A,B,g,X).\mathsf{iknows}(\mathsf{atag},B,A,GY) \\ \quad =[Msg]\Rightarrow \mathsf{state}_{\mathcal{A}}(A,B,g,X,GY,Msg).\mathsf{iknows}(\{|A,Msg|\}_{\mathsf{exp}(GY,X)}) \end{array}$$

$$\begin{array}{l} \mathsf{state}_{\mathcal{B}}(B,A,g).\mathsf{iknows}(\mathsf{atag},A,B,GX) \\ \quad =[Y]\Rightarrow \mathsf{state}_{\mathcal{B}}(B,A,g,GX,Y).\mathsf{iknows}(\mathsf{atag},B,A,\mathsf{exp}(g,Y)) \end{array}$$

$$\begin{array}{l} \mathsf{state}_{\mathcal{B}}(B,A,g,GX,Y).\mathsf{iknows}(\mathsf{plain},Z) \mid \ \pi_1(\{|Z|\}_{\mathsf{exp}(GX,Y)}) \approx A \\ \quad \Rightarrow \mathsf{state}_{\mathcal{B}}(B,A,g,GX,Y,Z,Msg).\mathsf{iknows}(\cdot) \end{array}$$

In the last rule, the freshly received message payload corresponds to the second component of Z, but B can realistically perform pattern matching only on the first component, as highlighted by the formalization. Indeed, the actual transitions are more verbose: first, we should explicitly introduce tags into the knowledge of honest agents. Moreover, and more importantly, the incoming messages are actually encoded with variables, which are then decomposed to access the message components with the help of the side conditions. To illustrate, the second transition for role A is indeed as follows:

$$\begin{array}{l} \mathsf{state}_{\mathcal{A}}(A,B,g,X).\mathsf{iknows}(W) \ \mid \ \mathsf{atag} \approx \pi_1(W), B \approx \pi_2(W), A \approx \pi_3(W) \\ \quad =[Msg]\Rightarrow \mathsf{state}_{\mathcal{A}}(A,B,g,X,W,Msg).\mathsf{iknows}(\{|Msg|\}_{\mathsf{exp}(\pi_4(W),X)}) \end{array}$$

We will disregard these subtleties in the rest of the presentation, and just use terms whenever there is no risk of confusion.

Attack Transitions The attack states are described by rules of the form

$$L \mid \mathit{Cond} \ \Rightarrow \ L.\mathsf{attack}$$

for a special fact symbol attack. These rules describe a set of states for which the attack rule applies, adding an attack fact: we may thus define attack states just as the states that contain an attack fact. The transitions corresponding to attacks naturally arise by the security goals described in Section 1.2.3, full details are reported in [59].

1.3.3 IF Semantics for *AnBx*

Given the output of the AnB2IF transformation, a further step is needed to produce an executable (and model-checkable) specification of the source *AnBx* code. As anticipated, we give two alternative characterizations of this transformation, yielding a cryptographic and an abstract semantics respectively. These transformations correspond to different views of the communication modes – either in terms of cryptographic primitives, or by means of abstract, idealized channel constructions – and provide corresponding transition rules that describe the behavior of the intruder.

Cryptographic Intermediate Format

The cryptographic semantics simply results from implementing authentic and confidential exchanges by means of digital signatures and public-key encryption, respectively.

Honest Agents For the honest agents, the translation is based on the mapping defined below. Tags are removed in the cryptographic translation, since their adoption would impose tight restrictions on the communication patterns we could express through forwarding. For instance, we could not cope with tags whenever a secure message is downgraded to authentic upon relaying.

IF	cryptIF
$\mathsf{iknows}(\mathsf{plain}, M)$	$\mathsf{iknows}(M)$
$\mathsf{iknows}(\mathsf{ctag}, \mathsf{blind}_B(M))$	$\mathsf{iknows}(\{M\}_{\mathsf{pk}(B)})$
$\mathsf{iknows}(\mathsf{atag}, A, \tilde{V}, M)$	$\mathsf{iknows}(\{\tilde{V}, M\}_{\mathsf{inv}(\mathsf{sk}(A))})$
$\mathsf{iknows}(\mathsf{stag}, A, \tilde{V}, \mathsf{blind}_B(M))$	$\mathsf{iknows}(\{\{\tilde{V}, M\}_{\mathsf{inv}(\mathsf{sk}(A))}\}_{\mathsf{pk}(B)})$
$\mathsf{iknows}(\mathsf{fresh} :: \mathsf{atag}, A, \tilde{V}, M, N)$	$\mathsf{iknows}(\{\tilde{V}, M, N\}_{\mathsf{inv}(\mathsf{sk}(A))})$
$\mathsf{iknows}(\mathsf{fresh} :: \mathsf{stag}, A, \tilde{V}, \mathsf{blind}_B(M), N)$	$\mathsf{iknows}(\{\{\tilde{V}, M, N\}_{\mathsf{inv}(\mathsf{sk}(A))}\}_{\mathsf{pk}(B)})$

The translation of the honest agents relies on the assumption that every agent A acting as the source of an authentic message or as the target of a confidential exchange initially knows two asymmetric key-pairs $(\mathsf{pk}(A), \mathsf{inv}(\mathsf{pk}(A)))$ for encryption, and $(\mathsf{sk}(A), \mathsf{inv}(\mathsf{sk}(A)))$ for signing. In this case we say that an agent is *certified*.

Given the correspondences established by the table, the translation is conceptually straightforward, as it amounts to rewriting all occurrences of facts on the left column with the corresponding facts of the right column for all the transition rules generated by the AnB2IF

compiler. In our running example, the translation produces the following set of transitions:

$$\begin{aligned}
&\mathsf{state}_{\mathcal{A}}(A,B,g).\mathsf{iknows}(\cdot)\\
&\quad =[X]\Rightarrow \mathsf{state}_{\mathcal{A}}(A,B,g,X).\mathsf{iknows}(\{B,\mathsf{exp}(g,X)\}_{\mathsf{inv}(\mathsf{sk}(A))})\\
&\mathsf{state}_{\mathcal{A}}(A,B,g,X).\mathsf{iknows}(\{A,GY\}_{\mathsf{inv}(\mathsf{sk}(B))})\\
&\quad =[Msg]\Rightarrow \mathsf{state}_{\mathcal{A}}(A,B,g,X,GY,Msg).\mathsf{iknows}(\{\!|A,Msg|\!\}_{\mathsf{exp}(GY,X)}))\\
&\mathsf{state}_{\mathcal{B}}(A,B,g).\mathsf{iknows}(\{B,GX\}_{\mathsf{inv}(\mathsf{sk}(A))})\\
&\quad =[Y]\Rightarrow \mathsf{state}_{\mathcal{B}}(A,B,g,GX,Y).\mathsf{iknows}(\{A,\mathsf{exp}(g,Y)\}_{\mathsf{inv}(\mathsf{sk}(B))})\\
&\mathsf{state}_{\mathcal{B}}(A,B,g,GX,Y).\mathsf{iknows}(\mathsf{plain},Z)\mid\ \pi_1(\{\!|Z|\!\}_{\mathsf{exp}(GX,Y)})\approx A\\
&\quad \Rightarrow \mathsf{state}_{\mathcal{B}}(A,B,g,GX,Y,Z).\mathsf{iknows}(\cdot)
\end{aligned}$$

In practice, the rewriting step is more complex, because, as we have noted, the terms listed as arguments of the $\mathsf{iknows}(\cdot)$ facts may indeed occur in the side conditions that decompose and check the incoming messages in the transitions of the IF code: though lengthy to explain, the details of how that can be accomplished arise as expected.

Example 4 (Blind forward) *Let us consider the following protocol*

$$\begin{array}{rcll} C & \rightarrow & M, & (-\,|\,-\,|\,A) : token\\ M & \rightarrow & A, & (-\,|\,-\,|\,A) : token \end{array}$$

Let us assume that "token" is known by both M and A. Interestingly, as we were informally discussing before, the relaying by M is performed irrespectively of the actual content of the message, since M is not able to perform any check on a confidential message for A, even if M already knows the payload of the message. This is shown by the IF transition rules produced by the translation of that exchange.

$$\begin{aligned}
&\mathsf{state}_{\mathcal{C}}(C,M,A,token).\mathsf{iknows}(\cdot)\\
&\quad \Rightarrow \mathsf{state}_{\mathcal{C}}(C,M,A,token).\mathsf{iknows}(\{token\}_{\mathsf{pk}(A)})\\
&\mathsf{state}_{\mathcal{M}}(M,A,C,token).\mathsf{iknows}(Z)\\
&\quad \Rightarrow \mathsf{state}_{\mathcal{M}}(C,M,A,token,Z)\\
&\mathsf{state}_{\mathcal{A}}(A,C,M,token).\mathsf{iknows}(Z)\mid \{Z\}_{\mathsf{inv}(\mathsf{pk}(A))}\approx token\\
&\quad \Rightarrow \mathsf{state}_{\mathcal{A}}(A,C,M,token,Z)
\end{aligned}$$

In the second transition rule, M accepts every variable Z provided by the intruder. Note that the $\mathsf{iknows}(Z)$ *fact is not reported explicitly on the right-hand side of the arrow, since those facts are permanent. In the last transition, instead, A can perform pattern matching on the received encrypted message.*

An additional measure is needed for translating to cryptIF the transitions expecting a fresh message on input. These transitions are easily identified in the annotated *AnB* code, as they have an occurrence of the fresh tag on their incoming message. For any such transition, let B be the receiver, and N the nonce associated with the fresh message: then we include the boolean formula $\mathsf{not}(\mathsf{seen}(B,N))$ as a side condition to the rule, and we introduce the fact

$\mathsf{seen}(B, N)$ in the right-hand side upon reception. This implements a simple mechanism of replay protection.

For instance, for the sender of the message $A \overset{@}{\to} B, (A \mid B \mid -) : Msg$ we have an IF rule that freshly creates a nonce for the transmission

$$\ldots \multimap\![N]\!\Rightarrow \mathsf{iknows}(\{B, Msg, N\}_{\mathsf{inv}(\mathsf{pk}(A))}) \ldots$$

and on the receiver side, we have accordingly

$$\ldots .\mathsf{iknows}(\{B, Msg, N\}_{\mathsf{inv}(\mathsf{pk}(A))}).N \notin \mathsf{seen}(B) \Rightarrow N \in \mathsf{seen}(B). \ldots .$$

i.e., the message M is received only if the nonce N was never seen before by the receiver. If this is the case and the message is accepted, the receiver updates the entries in its database by adding N.

Intruder rules The symbols sk and pk introduced before are public functions, hence every agent, including the intruder, can obtain the public keys of every other agent as soon as their name is known. Instead, the function $\mathsf{inv}(\cdot)$, providing the ability to construct signing and decryption keys is non-public. Making the functions sk and pk public provides the intruder with full capabilities to construct all messages prescribed by the Dolev-Yao intruder model. In addition, we assume a fact $\mathsf{dishonest}(A)$ to denote that A is a dishonest agent; while many tools assume that there is only a single dishonest agent i (the "intruder"), our model can support any number of collaborating dishonest agents – one may still think of one intruder who has compromised several agents and can now use their identities. Consequently, the intruder knows all private keys of dishonest agents.

Abstract Intermediate Format

The abstract semantics provides for a direct representation of the communication modes in terms of corresponding state facts that encode the types of channel where the exchanges take place. In particular, the abstract semantics draws on the following channel constructors athCh, cnfCh and secCh, around which we define persistent state facts that track the protocol exchanges (like the intruder knowledge facts of the form iknows).

Honest agents For honest agents, the Abstract Intermediate Format arises as the result of replacing the $\mathsf{iknows}(\cdot)$ facts that represent the input and output of messages in the IF rules generated by the AnB2IF transformation with corresponding channel facts according to the following table.

IF	absIF
$\mathsf{iknows}(\mathsf{plain}, M)$	$\mathsf{iknows}(M)$
$\mathsf{iknows}(\mathsf{ctag}, \mathsf{blind}_B(M))$	$\mathsf{cnfCh}(B; M)$
$\mathsf{iknows}(\mathsf{atag}, A, \tilde{V}, M)$	$\mathsf{athCh}(A; \tilde{V}; M)$
$\mathsf{iknows}(\mathsf{stag}, A, \tilde{V}, \mathsf{blind}_B(M))$	$\mathsf{secCh}(A; \tilde{V}; B; M)$
$\mathsf{iknows}(\mathsf{fresh} :: \mathsf{atag}, A, \tilde{V}, M, N)$	$\mathsf{athCh}(A; \tilde{V}; M, N)$
$\mathsf{iknows}(\mathsf{fresh} :: \mathsf{stag}, A, \tilde{V}, \mathsf{blind}_B(M), N)$	$\mathsf{secCh}(A; \tilde{V}; B; M, N)$

Two comments are in order. First, nonces are implicitly included in the payload of the message when freshness is lost upon forwarding: this choice reflects the corresponding behavior in cryptIF, where nonces cannot be realistically removed from signed packets. Secondly, given that the state channel facts employed in absIF are persistent, we need additional measures to protect against replicas in all transitions expecting a fresh message on input. Indeed, we may rely on the very same mechanism described earlier for the cryptographic transformation, based on the $\mathsf{seen}(\cdot,\cdot)$ facts.

Intruder Rules The intruder rules constitute the key component of the abstract semantics, as it is by way of these rules that we define the actual interpretation of the channel facts composing the specification. An intruder can forge a message over an authentic channel only if the associated sender identity is compromised, while he can learn every message sent over an authentic channel. Dually, an intruder can send over a confidential channel every message he can compose, but he can learn a message sent over a confidential channel only if the associated receiver identity is compromised.

Secure channels combine the guarantees of authentic and confidential channels, and may be downgraded to authentic channels by dishonest agents. This reflects the very same behavior in cryptIF, where an intruder may remove an encryption layer to get a signed message from an encoding of a secure channel.

$$\begin{aligned}
\mathsf{iknows}(\tilde{V}).\mathsf{iknows}(M).\mathsf{dishonest}(A) &\Rightarrow \mathsf{athCh}(A;\tilde{V};M)\\
\mathsf{athCh}(A;\tilde{V};M) &\Rightarrow \mathsf{iknows}(M).\mathsf{iknows}(\tilde{V})\\
\mathsf{iknows}(B).\mathsf{iknows}(M) &\Rightarrow \mathsf{cnfCh}(B;M)\\
\mathsf{cnfCh}(B;M).\mathsf{dishonest}(B) &\Rightarrow \mathsf{iknows}(M)\\
\mathsf{iknows}(\tilde{V}).\mathsf{iknows}(B).\mathsf{iknows}(M).\mathsf{dishonest}(A) &\Rightarrow \mathsf{secCh}(A;\tilde{V};B;M)\\
\mathsf{secCh}(A;\tilde{V};B;M).\mathsf{dishonest}(B) &\Rightarrow \mathsf{iknows}(M).\mathsf{iknows}(\tilde{V})\\
\mathsf{secCh}(A;\tilde{V};B;M).\mathsf{dishonest}(B) &\Rightarrow \mathsf{athCh}(A;\tilde{V};M)
\end{aligned}$$

Correspondence between cryptIF and absIF

There exists a bijective mapping between the cryptIF and absIF specifications of any given protocol. Formally, we can define a correspondence relation $\sim$ between cryptIF and absIF, according to the translation rules from annotated AnB to these two interpretations. Intuitively, two states are related by $\sim$ if they differ only for the channel encodings according to the correspondence depicted below: this implies that the knowledge of the honest agents and the intruder is the same for the two states.

cryptIF	absIF
$\mathsf{iknows}(M)$	$\mathsf{iknows}(M)$
$\mathsf{iknows}(\{M\}_{\mathsf{pk}(B)})$	$\mathsf{cnfCh}(B;M)$
$\mathsf{iknows}(\{\tilde{V},M\}_{\mathsf{inv}(\mathsf{sk}(A))})$	$\mathsf{athCh}(A;\tilde{V};M)$
$\mathsf{iknows}(\{\{\tilde{V},M\}_{\mathsf{inv}(\mathsf{sk}(A))}\}_{\mathsf{pk}(B)})$	$\mathsf{secCh}(A;\tilde{V};B;M)$

The formal definition is more complicated, due to, e.g., the knowledge of additional cryptographic material in cryptIF specifications. Complete details for a similar framework can be found in [63].

Given a cryptIF specification and the corresponding absIF specification we can formulate these hypothesis:

- For each reachable state S_1 of the absIF specification there exists a reachable state S_2 of the cryptIF specification such that $S_1 \sim S_2$;
- For each reachable attack state S_1 of the cryptIF specification there exists a reachable attack state S_2 of the absIF specification such that $S_1 \sim S_2$.

1.4 Cryptographic Channel API

The previous sections have used channels as a method to *abstract* from implementation details. This is the view of a protocol designer thinking about an application and not about cryptography. Thus, one can use the channel notation to design the application based on assumptions about the communication paths and ignore how to realize these assumptions. Note that this leaves room even for non-cryptographic implementations of channels (as sometimes used in device-pairing protocols for instance).

We now change to a slightly different view, namely that we do not want to entirely abstract from the realization of the channels, but only to *separate* the aspect of a high-level application protocol from the low-level mechanisms used to implement the assumed channels. In the spirit of abstraction, in the previous section the cryptIF was defined as the simple-most way to achieve channel properties from the point of view of formal verification tools – and that should be fine for any modeler who does not care about the concrete realization of channels. Now, in contrast, we will look at other possible realizations with a focus on the constraints of the practical implementation, available key infrastructure and so on.

The goal of this section is thus to still allow treating both the cryptographic primitives and the channel notation as a black box in the protocol description, but also – separately – to describe the implementation of these black boxes. So, basically, the channels and the cryptographic primitives become a high-level security API for protocol designers.

1.4.1 Cryptographic Toolbox

Our notation for symmetric and asymmetric cryptography, i.e., $\{|M|\}_K$ and $\{M\}_K$, consists of function symbols that abstractly represent combinations of more low-level cryptographic mechanisms. For instance, the term $\{|M|\}_K$ represents more than applying a symmetric encryption algorithm like AES that only protects confidentiality, but it also includes a MAC (message authentication code) to protect against manipulation of the encrypted message. Moreover, the message M may first be split into fixed-sized blocks and then be encrypted using, for instance, a chaining block cipher.

Similarly, for $\{M\}_K$, where K is a public key, we do not directly mean the application of an asymmetric cipher like RSA. Rather, we assume an hybrid scheme where a fresh symmetric key K' is created, only this key is asymmetrically encrypted using K, and the actual message M is symmetrically encrypted using K'; then the asymmetrically encrypted key is sent along with the symmetrically encrypted message.

A similar reasoning holds also for signatures, i.e., the notation $\{M\}_K$ (where K is a private key) does not denote that the entire message M is signed using, for instance, RSA, but rather than just a hash of M is signed and M is transmitted along with it as a plaintext (so that everybody can read it, even not knowing the corresponding public key).

We allow in *AnBx* to integrate this concrete cryptographic implementation into the translation process, reading $\{|M|\}_K$ and $\{M\}_K$ as the following abbreviations:

$$\{|M|\}_K = (\mathsf{senc}_K(M), \mathsf{hmac}_K(M))$$
$$\{M\}_K = \begin{cases} (\mathsf{aenc}_K(K'), \{|M|\}_{K'}) & \text{if } K \text{ is a public key } [K' \text{ fresh symmetric key}] \\ (\mathsf{sig}_K(\mathsf{hash}(M)), M) & \text{if } K \text{ is a private key} \end{cases}$$

where senc, aenc, hash, hmac, and sig represent standard cryptographic primitives, as they are found in typical crypto-libraries.

Another important aspect of our abstractions is the choice of the message formats. A message may be a name m, a tuple of messages $(\tilde{M})$, or a *message digest* $[M]$. Indeed, no explicit cryptographic operator is necessary for message formation, given our channel abstractions. The only operation on data needed in most e-commerce protocols, as we show in our case studies, turns out to be the creation of digests $[M]$ to prove the knowledge of M without leaking it. We may need also digests which are resistant to chosen-plaintext attacks, hence presuppose an implementation based on a hashing scheme that packages M together with a randomized quantity known to the principals that possess M, and is never leaked to any principal that do not have knowledge of M. We thus allow digests to be tagged with an annotation that specifies the intended verifier, as in $[M{:}A]$.

$$[M] = \mathsf{hash}(M)$$
$$[M{:}A] = (\mathsf{aenc}_{\mathsf{pk}(A)}(K'), \mathsf{hmac}_{K'}(M)) \qquad [K' \text{ fresh symmetric key}]$$

In any case, *AnBx* is designed as to be very flexible with respect to cryptographic primitives, since its terms are defined over an arbitrary set of function symbols with their specific algebraic properties. Thus, a designer may choose to work with a lower level of cryptographic primitives, i.e., using function symbols representing basic algorithms like AES and RSA, rather than our higher-level symbols $\{|M|\}_K$ and $\{M\}_K$.

1.4.2 Alternative Cryptographic Channel Models

We now consider a few alternatives for the implementation of channels in the cryptIF.

Challenge-Response for Freshness So far we have introduced a sender-generated unique number in fresh-authentic and fresh-secure transmissions to allow the receiver to check this number against a database of previously received numbers and filter out replicas. An alternative to this is using a challenge-response mechanism, where the nonce is generated by the receiver and no database is needed.

The transmission $A \overset{@}{\to} B, (A \,|\, \tilde{V} \,|\, -) : M$ may be translated into:

$$\begin{aligned} A &\to B, (- \,|\, - \,|\, -) : && A, B \\ B &\to A, (- \,|\, - \,|\, -) : && N_B \\ A &\to B, (A \,|\, \tilde{V} \,|\, -) : && M \\ A &\to B, (A \,|\, B \,|\, -) : && N_B, \mathsf{hash}(M) \end{aligned}$$

Note that the last two message are only separated since we use the notation for authentic channels and we want to specify different modes for these components.

In the implementation we have separated the authentication and the freshness part to allow B to easily forward just the non-fresh authentic component. One may alternatively choose to implement this slightly differently to include also the fresh nonce in non-fresh forwards, similarly to what we did in the cryptIF above for simplicity: in this case, the nonce does not play any specific role for the destination of a forwarding operation. Obviously, we can rely on a similar construction to implement fresh-secure channels.

Diffie-Hellman for Freshness Another alternative to implement freshness is to use a Diffie-Hellman scheme. For instance, a possible implementation could look like this:

$$\begin{array}{ll} A \to B, (-\,|\,-\,|\,-): & A, B \\ B \to A, (-\,|\,-\,|\,-): & \mathsf{exp}(g, Y) \\ A \to B, (A\,|\,\tilde{V}\,|\,-): & M \\ A \to B, (A\,|\,B\,|\,-): & \mathsf{exp}(g, X), \{|\mathsf{hash}(M)|\}_{\mathsf{exp}(\mathsf{exp}(g,X),Y)} \end{array}$$

where X and Y are exponents freshly created by A and B, respectively. Here, only the half-key $\mathsf{exp}(g, X)$ from A is authenticated, but B can be sure about the freshness of the transmission thanks to his freshly generated secret Y. Again, freshness is decoupled from authenticity in our implementation and we use two different messages by A to convey distinct guarantees.

Forward Modes The implementation of most forwarding modes is straightforward: a forwarder simply receives a message from the network and then sends it to the next receiver. There are only three cases that deserve special attention and a further explanation:

- Dropping freshness: we already mentioned that A can send a fresh-authentic message to B, and B can forward this to C preserving the authenticity from A, but downgrading any freshness guarantee. With the previous implementations, B can simply drop upon forwarding the freshness certificate received as the fourth message in the narration;
- Decryption/Re-encryption: forwarding a message may involve a decryption / an encryption when confidentiality guarantees are changed upon relaying;
- Fresh-forward: here, the implementation with challenge-response is a bit more complicated, as also the receiver of the forward must supply a nonce. The following exchange

$$\begin{array}{ll} A \overset{@}{\to} B, (A\,|\,C, B\,|\,-): & M \\ B \overset{@}{\to} C, (A\,|\,C, B\,|\,-): & M \end{array}$$

  is then implemented as follows:

$$\begin{array}{ll} A \overset{@}{\to} B, (A\,|\,C, B\,|\,-) & : M \\ B \to C, (A\,|\,C, B\,|\,-) & : M \\ C \to A, (-\,|\,-\,|\,-) & : N_C \\ A \to C, (A\,|\,C\,|\,-) & : N_C, \mathsf{hash}(M) \end{array}$$

  This applies the freshness mechanism to another challenge response.

2

AnBx Case Studies

2.1 Case Study: e-Commerce Protocols

We want to demonstrate that *AnBx* can be employed as a specification language for a purely declarative modelling of distributed protocols. To this end, we show that the *AnBx* channels modes result successful as a set of communication abstractions, which provide primitive support for the expected high-level security guarantees of a wide and interesting class of protocols, namely e-payment protocols.

Introducing the Case Studies In general, the design and specification of e-payment systems are complex, and their analysis is challenging. Despite of this complexity, abstracting from cryptographic details, we can isolate a common communication pattern among most of these protocols (e.g., *iKP* [15, 16], *SET* [12–14] and *3-D Secure* [80]) and outline a general e-payment scheme which captures the essential ingredients of them. Such scheme can help the designer focusing on the business logic of the protocol, specifying a template in which the different security properties enforced at every exchange are parameterized by *AnBx* modes: the instantiation of these parameters gives rise to different concrete protocols.

Although *iKP* and *SET* are no longer in use we still found interesting to focus on them because, beyond the specific reasons outlined below, they represent a significant benchmark for their complexity.

The first case study we propose is the aforementioned *iKP* family of e-payment protocols. We show in particular that one of the key features of *iKP* - the increasing levels of security according to the number of certified principals, - can be achieved in a very elegant and effective way through its *AnBx* specification. As a byproduct of this case study, we find a new flaw in the original *iKP* specification and propose a fix.

The second case study illustrates a revised version of *SET*, a protocol that for its complexity is considered a benchmark for protocol analysis. Here, we shift our attention to some known security flaws of the protocol and show that our revised version is immune to such defects.

The *SET* case study employs the fresh forward mode introduced in Section 1.2 to propose a solution to an issue outlined in [79]: even a successful and completed *SET* protocol run does not give the parties enough evidence to provide certain important transaction features. Namely, the customer is not able to have evidence that the payment has indeed been authorized by the acquirer; by means of the fresh forward mode, we can instead achieve such a goal.

Interestingly, in both case studies, our version of the protocols outperforms in term of

security guarantees the original one, giving evidence that using adequate abstractions results not only in simpler design, but also in a more robust implementation. This is largely a consequence of the declarative nature of the specification style supported by *AnBx*.

We verified the *AnBx* specifications of *iKP* and *SET* by compiling them into cryptIF, using our tool, and running OFMC on the generated transitions against the expected security goals, discussed below. We tested all the different possible implementations outlined in Chapter 1. We also encoded and verified the original versions of *iKP* and *SET*, and compared the results against those of the revised versions. In the following we report on the results of such tests. For all the tests we ran OFMC in classic mode with one and two sessions, using both the typed and the untyped mode: with two sessions we were sometimes unable to complete the verification due to search space explosion. We also ran intensive tests limiting the depth of the search space to remain within the available memory space.

In the following we assume that certified principals own dual key pairs, for encryption and digital signatures which are valid within a Public Key Infrastructure and were issued by a trusted third party (Certification Authority).

2.1.1 A General e-Payment Scheme

We now outline the general communication pattern considered in the case studies, which constitutes a bare-bone specification of an e-payment protocol. This abstraction allows us to introduce here most of the concepts which are common to both the examples we consider.

In our model each principal starts with an initial knowledge shared with other participants. Indeed, since most e-commerce protocols describe only the payment transaction and do not consider any preliminary phase, we assume that Customer and Merchant have already agreed on the details of the transaction. These details constitute a *contract* that includes an *order description* (*desc*) and a *price*. We also assume that payments are based on existing credit-card systems operated by an Acquirer, who shares with the Customer a *customer's account number* (*can*) comprising the credit card number and the associated PIN.

The initial knowledge of the three parties can thus be summarized as follows:

- Customer *C: price, desc, can*;
- Merchant *M: price, desc*;
- Acquirer *A: can*.

The transaction can be decomposed into the following steps:

1. $C \rightarrow M$: *Initiate*
2. $C \leftarrow M$: *Invoice*

   (In steps 1 and 2 the Customer and the Merchant exchange all the information which is necessary to compose the next payment messages.)
3. $C \rightarrow M$: *Payment Request*

4. $M \rightarrow A$: *Authorization Request*

   (In steps 3 and 4 the Customer sends a payment request to the Merchant. The Merchant uses this information to compose an authorization request for the Acquirer and try to collect the payment.)

5. $M \leftarrow A$: *Authorization Response*

6. $C \leftarrow M$: *Confirm*

   (In steps 5 and 6 the Acquirer processes the transaction information, and then relays the purchase data directly to the issuing bank, which actually authorizes the sale in accordance with the Customer's account. This interaction is not part of the narration. The Acquirer returns a response to the Merchant, indicating success or failure of the transaction. The Merchant then informs the Customer about the outcome of the process.)

Beside some elements of the initial knowledge, other information needs to be exchanged in the previous process. First, to make transactions univocally identifiable, the Merchant generates a fresh transaction ID (*tid*) for each different transaction. Second, the Merchant associates to the transaction also a *date* or any appropriate time information. Both pieces of information must be communicated to the other parties. The core information describing a transaction is then identified by the tuple $(price, tid, date, can, desc)$, which also constitutes the payment order information. This tuple plays the role of the *contract* among the three parties: if Customer and Merchant reach an agreement on it, and they can prove this to the Acquirer, then the transaction can be completed successfully. The transaction authorization result *auth* is then returned by the Acquirer, and communicated to the two other participants.

We note that two main confidentiality concerns arise in the previous process: on the one hand, the Customer typically wishes to avoid leaking credit-card information to the Merchant; on the other hand, the Customer and the Merchant would not let the Acquirer know the details of the order or the services involved in the transaction. Both these requirements can be enforced by protecting the exchange of *can* and *desc* with the digests we introduced in Section 1.4.1.

In the end, the structure of the payment protocol template can be specified by means of the *AnBx* messaging primitives as follows:

1. $C \rightarrow M, \eta_1 : [can{:}A], [desc{:}M]$
2. $C \leftarrow M, \eta_2 : price, tid, date, [contract]$
3. $C \rightarrow M, \eta_3 : price, tid, can, [can{:}A], [contract]$
4. $M \rightarrow A$ (decomposed in two steps to specify different communication modes, if necessary)
   (a) $M \rightarrow A, \eta_{4a} : price, tid, can, [can{:}A], [contract]$
   (b) $M \rightarrow A, \eta_{4b} : price, tid, date, [desc{:}M], [contract]$
5. $M \leftarrow A, \eta_5 : auth, tid, [contract]$
6. $C \leftarrow M, \eta_6 : auth, tid, [contract]$

The digests [*can:A*] and [*desc:M*] are annotated with their intended verifiers (if M is not certified, the second digest deserves some more care, as we will discuss shortly). Correspondingly, [*contract*] is actually the digest of a tuple $(price, tid, date, [can{:}A], [desc{:}M])$, where all the sensitive data is protected against chosen plaintext attacks by the usage of the nested digests. By instantiating the exchange modes η_i in the previous scheme, one may generate different versions of the scheme, achieving different security guarantees: this is exactly what we do in our case studies.

Protocol Goals A first goal we would like to achieve for an e-payment system is that all the involved parties agree on the contract they sign, and are able to verify this. In terms of OFMC goals, this corresponds to requiring that each participant can authenticate the other two parties on [*contract*]. Moreover, the Acquirer should be able to prove that the payment has indeed been authorized and the associated transaction performed: in OFMC this can be represented by requiring that M and C can authenticate A on the authorization *Auth*. In summary, the authentication goals we would like to achieve are the following:

1. *M authenticates C on [contract]*, to give evidence to M that C has authorized the payment to her;
2. *C authenticates M on [contract]*, to give evidence to C on the terms of the purchase that M has settled with her;
3. *A authenticates C on [contract]*, to give evidence to A that C authorized her to transfer the money from her account to M;
4. *A authenticates M on [contract]*, to give evidence to A that M has requested the transfer of the money to her account;
5. *C authenticates A on [contract],Auth*, to give evidence to C that A authorized the payment and performed the transaction;
6. *M authenticates A on [contract],Auth*, to give evidence to M that A authorized the payment and performed the transaction.

Since authentication is realized by means of the digital signature, it follows that, if a principal is not certified, it cannot provide the required evidence.

Finally, we are also interested in some secrecy goals, like verifying that the Customer's credit card information *can* is kept confidential, and transmitted only to the Acquirer. In general, we would like to keep the information exchanged by the principals secret among the expected parties.

All validated protocol goals are reported in the discussion about the case studies.

Non repudiation analysis A stronger variant of the authentication goals described above requires that, after completion of a transaction, each participant is able to provide a non-repudiable proof of the effective agreement by the other two parties on the terms of the transaction. In principle, each principal may wish to have sufficient proofs to convince an external verifier that the transaction was actually carried out as she claims. The lack of some of these provable authorizations does not necessarily make the protocol insecure, but it makes disputes between the parties more difficult to settle, requiring to rely on evidence provided by

other parties or to collect off-line information. We discuss this issue more in detail in Section 2.3.

2.2 The iKP Protocol Family

The *iKP* protocol family ($i \in \{1, 2, 3\}$) was developed at IBM Research [15, 16, 65] to support credit card-based transactions between customers and merchants (under the assumption that payment clearing and authorization may be handled securely off-line). All protocols in the family are based on public-key cryptography, and vary in the number of parties that own individual public key-pairs to generate digital signatures: this is reflected by the name of the different protocols – *1KP*, *2KP*, *3KP* – which offer increasing levels of security. We model this feature by a corresponding variation of the number of certified principals.

In Table 2.1 we instantiate the communication modes of the general e-payment scheme to define the *AnBx* counterpart of the *iKP* protocol family on the basis of the number of participants having public key-pairs at their disposal. The different instantiations elegantly capture the possibility of the protocol to scale upon the maturity of the underlying public key infrastructure, an important design feature of *iKP*. Here, to make the table more compact, we prefix the communication mode with @ to denote a fresh exchange.

The (common) narration of the three resulting protocols is the following:

- Step 1 and 2: Customer and Merchant exchange data that let them build independently the *contract*. They must tell their "own version of the story" to the acquirer. C declares [*can:A*], the credit card that will be used in *contract*, sending the protected digest of *can* to M. The Customer also informs (and agrees with) the Merchant on the digest of *desc* they will use to define the *contract*. M generates a (fresh) transaction ID (*tid*) and the *date* of transaction (C and M already had agreed on *price* and *desc*, being part of their initial knowledge). M can verify the integrity of [*desc:M*], form the *contract*, and then compute its digest. The tuple $(price, tid, date, [contract])$ is sent to C which, upon receiving it, can compute [*contract*] and verify that it matches the digest provided by M. If the match succeeds, the protocol execution continues, otherwise it stops.

- Step 3: the Customer prepares a secret message for the Acquirer containing the information necessary to complete the transaction: [*contract*], credit card number – *can* –, amount of the transaction – *price* – and transaction ID – *tid*. However this message is sent to the Merchant and not to the Acquirer, because often, as in *SET* and *iKP*, an e-commerce protocol does not allow direct interaction between Customers and Acquirers, but only through Merchant mediation. We assume that M is cooperating in delivering messages. Hence M receives an opaque message, and the only thing he can do is to blindly forward the bitstream to A (step 4a).

- Step 4: the Merchant sends the tuple $(price, tid, date, [desc:M], [contract])$ to the Acquirer (step 4b). This information is necessary to complete the payment. In particular *date* and [*desc:M*] are required by A to compute independently the digest of *contract*. Upon reception of the two versions of [*contract*] originating from the two other principals, A can also compute the same value autonomously. If all three match, the transaction can be authorized, since this is the proof of the complete agreement between the customer and the merchant.

mode/step	$\rightarrow$	1KP	2KP	3KP
η_1	$C \rightarrow M$	$(-\,\vert\,-\,\vert\,-)$	$(-\,\vert\,-\,\vert\,M)$	$@(C\,\vert\,M\,\vert\,M)$
η_2	$C \leftarrow M$	$(-\,\vert\,-\,\vert\,-)$	$@(M\,\vert\,C\,\vert\,-)$	$@(M\,\vert\,C\,\vert\,C)$
η_3	$C \rightarrow M$	$(-\,\vert\,-\,\vert\,A)$	$(-\,\vert\,-\,\vert\,A)$	$(C\,\vert\,A\,\vert\,A)$
η_{4a}	$M \rightarrow A$	$(-\,\vert\,-\,\vert\,A)$	$(-\,\vert\,-\,\vert\,A)$	$(C\,\vert\,A\,\vert\,A)$
η_{4b}	$M \rightarrow A$	$(-\,\vert\,-\,\vert\,A)$	$@(M\,\vert\,A\,\vert\,A)$	$@(M\,\vert\,A\,\vert\,A)$
η_5	$M \leftarrow A$	$@(A\,\vert\,C, M\,\vert\,-)$	$@(A\,\vert\,C, M\,\vert\,M)$	$@(A\,\vert\,C, M\,\vert\,M)$
η_6	$C \leftarrow M$	$(A\,\vert\,C, M\,\vert\,-)$	$(A\,\vert\,C, M\,\vert\,-)$	$(A\,\vert\,C, M\,\vert\,C)$
certified agents		A	M, A	C, M, A

Table 2.1: Exchange modes for the revisited *iKP* e-commerce protocol

- Step 5: the Acquirer sends the authorization response to the Merchant, within a fresh authentic message, containing also *tid* and [*contract*]. This is done in order to bind all the information, and produce a proof that the payment has been authorized for that specific transaction and that specific contract.

- Step 6: the Merchant forwards the message received from the Acquirer to the Customer This is a notification of the result of the transaction. In this way C receives, via M, a proof of payment from A. Since the message is signed by A, M cannot alter the message without being discovered.

2.2.1 Security Analysis

We verified the *AnBx* protocols described above and carried out a corresponding analysis of the original specifications of {1,2,3}*KP*, as amended in [64] with respect to the very first description in [15]. Below we refer to this amended version as the "original" *iKP*, to be contrasted with the "revised" *AnBx* version discussed above. In both cases we ran our tests assuming that the Acquirer is trusted (technically, modeled as a constant in the OFMC specification). This appears reasonable in an e-commerce application, since the Acquirer should act as a trusted third party. For *3KP*, the *AnBx* code for the revised and the original version of the protocol is shown in the Appendix (Tables A.2 and A.1).

As we mentioned earlier, the *AnBx* specification are not just more scalable: they provide stronger security guarantees, which are detailed in Table 2.2 and commented further below. In general, our implementation of the *iKP* protocols outperforms the original version, i.e., it satisfies more and stronger security goals, for all i's. This is largely due to the declarative nature of the *AnBx* messaging primitives, which, being defined as channel abstractions, provide strong encapsulation mechanisms on the messages they exchange. The price to pay with respect to the original *iKP* is that we need to split step 4 in two substeps, plus one additional step to return the initiative to the merchant after step 4a.

More in detail, during the analysis of the original *2KP* and *3KP* we found a (to the best of our knowledge) new flaw. It is related with the authenticity of the Authorization response *auth* that is generated by the acquirer and then sent to the other principals at step 5 and

certified agents	*A*		*M,A*		*C,M,A*	
	1KP		2KP		3KP	
Goal	O	R	O	R	O	R
can secret between *C,A*	-	-	-	-	+	+
C confidentially sends *can* to *A*	+	+	+	+	+	+
A weakly authenticates *C* on *can*	-	-	-	-	-	+
desc secret between *C,M*	+	+	+	+	+	+
auth secret between *C,M,A*	-	-	-	-	-	+
tid secret between *C,M,A*	-	-	-	-	-	+
price secret between *C,M,A*	-	-	-	-	-	+
[*contract*] secret between *C,M,A*	-	-	-	-	-	+
M authenticates *A* on *auth*	-	+	+*	+	+*	+
C authenticates *A* on *auth*	-	+	+*	+	+*	+
A authenticates *C* on [*contract*]	-	-	-	-	+	+
M authenticates *C* on [*contract*]	-	-	-	-	+	+
A authenticates *M* on [*contract*]	-	-	w	+	w	+
C authenticates *M* on [*contract*]	-	-	+	+	+	+
C authenticates *A* on [*contract*],*auth*	-	+	+	+	+	+
M authenticates *A* on [*contract*],*auth*	-	+	+	+	+	+

* goal satisfied only fixing the definition of Sig_A

w = only weak authentication

Table 2.2: Security goals satisfied by Original and Revised *iKP*

6. In particular, the starred goals in Table 2.2 are met only after changing the protocol by adding the identities of merchant and customer inside the signature Sig_A in the original specification (in *2KP*, since the customer is not certified, this can be done with an ephemeral identity derived from the credit card number). Namely, in the original specification [15], $Sig_A \triangleq \{\mathsf{hash}((Auth, \mathsf{hash}(Common))\}_{\mathsf{inv}(\mathsf{sk}(A))}$ should be replaced by

$$Sig_A \triangleq \{hash(C, M, Auth, hash(Common))\}_{\mathsf{inv}(\mathsf{sk}(A))}$$

Besides the previous discussion, it is interesting to point out that our revisited *2KP* behaves almost as good as the original *3KP* from a security point of view. A goal that is not satisfied by the former is "*can* secret between *C,A*", but this does not mean that the credit card is leaked, but only that it is not strongly authenticated by the acquirer, and indeed the weaker goal `C confidentially sends can to A` succeeds. As to authentication, in *3KP* the credit card number can be signed by *C*, whereas this is not possible in *2KP* and *1KP* (neither in the original, nor in the revised versions). In these protocols, the acquirer weakly authenticates the Customer by means of the credit card number, which is a shared secret between the two parties. Moreover, the fact the *C* is not certified prevents in *2KP* the possibility to authenticate this principal on [*contract*].

Finally, it is worth to point out that, after the completion of our version of *3KP* payment protocol, each party has evidence of transaction authorization by the other two parties. Therefore our version achieves all the goals that can ideally be requested, according to the number of certified principals. On the contrary even the original *3KP*, the strongest original version, fails in one goal: A can only weakly authenticate M on [*contract*].

2.3 SET Purchase Protocol

Secure Electronic Transaction (*SET*) is a family of protocols for securing credit card transactions over insecure networks. This standard was proposed by a consortium of credit card companies and software corporations led by Visa and MasterCard and involving companies like IBM, Microsoft, Netscape, RSA and Verisign. The main aim of *SET* was to enable customers to make purchases, having guarantees of authenticity of the transaction while keeping the customer's account details secret from the Merchant and his choice of goods secret from the Acquirer (payment gateway). Despite being backed by the major players of the IT and financial industry, *SET* failed to become the standard "de facto" for electronic payments. Reasons for its lack of adoption include the fact that the infrastructure required by *SET* is complex and requires cooperation of many parties to be established. Moreover, as we will see, *SET* fails to meet some of the desired protocol goals.

Nevertheless *SET* is still an interesting case study, as its complex structure makes it a benchmark for security protocols design and verification. It is a real-world protocol, arising from the industry, and its documentation comprises more than 1000 pages. Such a large protocol needs tool support to be verified effectively. Bella et al. [12–14] made a major endeavor analyzing and abstracting the *SET* protocol and we take their work as a reference for the protocol specification.

In the present case study we consider the *SET* purchase protocol as outlined in [14]. *SET* uses many optional data and, depending on which are taken into account, we may obtain different alternative versions of the purchase phase. The most difficult task is to find a version that is both simple and relatively close to reality. Following a common idea in literature [14, 26, 45, 49, 54, 56, 79], we consider a single transaction involving no optional data. We assume that the Merchant registration and the optional cardholder registration have been successfully completed. In *AnBx* this can be represented by the fact that agents C and M are certified. The certification of C is optional, in analogy to *2KP*, where only the Customer and the Acquirer are certified. This variant of the protocol is called the *unsigned* version of *SET*, in contrast with the *signed* version, where all principals are certified.

Introduction to SET To ease the comparison with other works on *SET*, in this presentation the information exchanged is denoted with the names commonly used in the *SET* specification. Now we introduce some basic concepts of the protocol and provide a mapping of the data to the general e-commerce template presented in Section 2.1.

As it is common in e-payment protocol specifications, we assume that Customer and Merchant have already completed the *initial shopping agreement*, and they have agreed on the order description – *OrderDesc* – and purchase amount – *PurchAmt*. This is not part of the *SET* specification and it can be done by any mean, even out of band, as long as the secrecy of *OrderDesc* and *PurchAmt* is guaranteed. The *primary account number – pan(C)* – is an abstraction of the credit card number belonging to the Customer. If the cardholder

registration (a subprotocol of *SET*) is completed, *pan(C)* is a public-key certificate that includes the hash of the credit card number and an optional PIN (*PanSecret*). Otherwise *pan(C)* could be the credit card number itself. In any case the Acquirer should be able to verify the validity of the payment information presented by the Customer. In the case of the unsigned version, the payment information may also be employed as a weak proof of identity of *C*, since *C* cannot digitally sign. he initial knowledge of each participant, similarly to *iKP*, is the following:

- Customer *C: PurchAmt,OrderDesc,pan(C)*;
- Merchant *M: PurchAmt,OrderDesc*;
- Acquirer *A: pan(C)*.

During the protocol run the principals generate some identifiers: *LIDM* is a local transaction identifier that the Customer sends to the Merchant in order to identify the transaction, while the Merchant generates *XID*, which is used in the rest of the protocol as a transaction ID. However there is debate on whether *XID* can be considered globally unique or not [26]; we will return later to this issue. In analogy with the *SET* specification we use some abbreviations, but compared with the original presentation (see [14,26] and Table A.3 in appendix) they are just a few:

- *TID: LIDM,XID*;
- *OIdata: OrderDesc*;
- *PIdata: pan(C)*;
- *HOD:* [*OIdata:M*],[*PIdata:A*].

HOD contains the evidence (digest) of the credit card the Customer intends to use, and the evidence of the order description that will later forward to the Acquirer. Being part of an authentic message, *HOD* binds its components similarly to what happens with the dual signature. The Merchant can verify the digest [*OIdata:M*] generated by the Customer, while *TID* contains information he already knows and that can be checked.

Dual Signature A key idea introduced in *SET* is the *dual signature*. Its purpose is to let several parties agree on a transaction without giving any of them full view of the details. The Merchant does not need the customer's credit card number to process an order, but he only needs to know that the payment has been approved by the Acquirer. Conversely, the Acquirer does not need to be aware of the details of the Customer's order, but he just needs evidence that a particular payment has been required for a specific order.

As showed in [14], a fragment of an e-commerce protocol using the dual signature, as *SET* does in steps 3–4, may be the following:

1. $C \to M : \{OIdata, \mathsf{hash}(PIdata)\}_{\mathsf{pk}(M)}, \mathsf{sign}_C(M, A, \mathsf{hash}(OIdata), \mathsf{hash}(PIdata)), \{PIdata\}_{\mathsf{pk}(A)}$
2. $M \to A : \{\mathsf{hash}(OIdata)\}_{\mathsf{pk}(A)}, \mathsf{sign}_C(M, A, \mathsf{hash}(OIdata), \mathsf{hash}(PIdata)), \{PIdata\}_{\mathsf{pk}(A)}$

where $\mathsf{sign}_C(m) = \{\mathsf{hash}(m)\}_{\mathsf{inv}(\mathsf{sk}(C))}$. The Customer computes the digital signature of the concatenation of the two digests – of *OIdata* and *PIdata* – obtaining what is called *dual signature*:

$$\mathsf{sign}_C(M, A, \mathsf{hash}(OIdata), \mathsf{hash}(PIdata))$$

However, in contrast to the example given in [14], we include the identities of the intended verifiers (*M,A*) to comply with the Abadi and Needham's [4] explicitness principle.

The Customer sends the dual signature to the Merchant along with two other components: the first is $\{OIdata, \mathsf{hash}(PIdata)\}_{\mathsf{pk}(M)}$ – secret for the Merchant – while the second is $\{PIdata\}_{\mathsf{pk}(A)}$ – secret for the Acquirer. The first component allows the Merchant to have evidence of the digest of *PIdata* without seeing *PIdata* itself (the payment information). Similarly in the second step the Merchant secretly sends the digest of $OIdata$ to the Acquirer – $\{\mathsf{hash}(OIdata)\}_{\mathsf{pk}(A)}$, along with $\{PIdata\}_{\mathsf{pk}(A)}$ and the dual signature, received from the customer.

Thus Merchant and Acquirer can independently verify that the content of the dual signature equals to $\mathsf{hash}(\mathsf{hash}(OIdata), \mathsf{hash}(PIdata))$, a value they can compute from their current knowledge. Clearly this mechanism works only if Merchant and Acquirer do not cooperate to cheat the Customer, exchanging their information. Here we assume that the Acquirer, typically a financial institution, is trusted and respects the privacy of the Customers.

While this communication pattern can be applied with the signed version of SET it is unfit with the unsigned version since the Customer is unable to sign. In this case the dual signature is replaced by the term $\mathsf{hash}(\mathsf{hash}(OIdata), \mathsf{hash}(PIdata))$, which binds the two hashes, but is built using a hash function rather than a digital signature. This implies that the intruder could attempt a chosen plaintext attack to forge the hashes, making the *OIdata* secrecy more vulnerable.

To counter this problem we propose to use the *AnBx* protected digests (Section 1.4), which have the nice property to be verifiable only by a predefined principal and therefore are immune from chosen plaintext attacks (the encryption key acts as a confounder). Instead of the dual signature, the Customer can build the pair $([OIdata{:}A], [PIdata{:}M])$, in which the two components are verifiable by the A and M respectively. The two components are bound together by the secret message that C sends to M as in the following code:

$$\begin{array}{lcl} C \to M, (- \mid - \mid M) & : & [OIdata{:}M], [PIdata{:}A], OIdata \\ C \to M, (- \mid - \mid A) & : & [OIdata{:}M], [PIdata{:}A], PIdata \\ M \to A, (- \mid - \mid A) & : & [OIdata{:}M], [PIdata{:}A], PIdata \\ M \to A, (M \mid A \mid A) & : & [OIdata{:}M], [PIdata{:}A] \end{array}$$

achieving the same purpose of the dual signature even if the Customer is not certified (as in the unsigned *SET*). With some slight adaptation this fragment of protocol will be the central part (steps 3–4) of the revised version of *SET* we are going to present. For scalability, this solution can also be applied in the signed version just tuning the exchange modes according to the Customer ability to digitally sign, i.e., setting the mode $(C \mid M \mid M)$ in the first step and $(C \mid A \mid A)$ in the second and third steps.

Revisiting SET Although many papers on *SET* [14, 26, 79] focused their attention mostly on the signed version of *SET* we analyzed both versions. Table 2.3 shows the communication modes to instantiate the following protocol template for the unsigned and signed versions of SET:

1. $C \to M, \eta_1 :$ *LIDM*
2. $M \to C, \eta_2 :$ *XID*

mode/step	$\rightarrow$	*unsigned* SET	*signed* SET
η_1	$C \rightarrow M$	$(-\,|\,-\,|\,M)$	$@(C\,|\,M\,|\,M)$
η_2	$C \leftarrow M$	$@(M\,|\,C\,|\,-)$	$@(M\,|\,C\,|\,C)$
η_{3a}	$C \rightarrow M$	$(-\,|\,-\,|\,M)$	$@(C\,|\,M\,|\,M)$
η_{3b}	$C \rightarrow M$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4a}	$M \rightarrow A$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4b}	$M \rightarrow A$	$@(M\,|\,A\,|\,A)$	$@(M\,|\,A\,|\,A)$
η_5	$M \leftarrow A$	$@(A\,|\,C, M\,|\,M)$	$@(A\,|\,C, M\,|\,M)$
η_6	$C \leftarrow M$	$@(A\,|\,C, M\,|\,-)$	$@(A\,|\,C, M\,|\,C)$
certified agents		M, A	C, M, A

Table 2.3: Exchange modes for the revisited *SET* e-commerce protocol

3. # Payment Request
   (a) $C \rightarrow M, \eta_{3a}$: *TID,HOD*
   (b) $C \rightarrow M, \eta_{3b}$: *TID,PurchAmt,HOD,PIdata*
4. # Authorization Request
   (a) $M \rightarrow A, \eta_{4a}$: *TID,PurchAmt,HOD,PIData*
   (b) $M \rightarrow A, \eta_{4b}$: *TID,PurchAmt,HOD*
5. $A \rightarrow M, \eta_5$: *TID,HOD,AuthCode*
6. $M \rightarrow C, \eta_6$: *TID,HOD,AuthCode*

Step by step the narration of the signed version of the protocol is:

- Step 1 and 2 - *Purchase Initialization Request/Response*: the Customer sends the local transaction identifier *LIDM*, then the Merchant replies with the transaction identifier *XID*. These steps have the sole aim to let C and M exchange data that is needed to build the messages exchanged in the following steps. In these steps the exchanges are secret, authentic and fresh.
- Step 3 - *Payment Request*:
  - (a) the Customer prepares a fresh authentic message secret for the Merchant containing *TID* and *HOD*. *TID* includes information to identify the transaction, while *HOD* contains the evidence (digest) of the credit card the Customer intends to use, and the evidence of the order description that will later forward to the Acquirer. Being part of an authentic message, *HOD* binds its components similarly to what happens with the dual signature. The Merchant can verify the digest [*OIdata:M*] generated by the Customer, while *TID* contains information he already knows and that can be checked. If the verification fails the protocol can stop since there is

no possibility of authorizing a payment if Customer and Merchant disagree on *OrderDesc* or *TID* or *PurchAmt* (see also step 5)

- (c) the Customer composes a fresh authentic and secret message for the Acquirer. Beside the information provided in step 3.a, the message includes the payments information – *PIdata* – which will be processed by the Acquirer in order to authorize the payment. The message is delivered to the Merchant, since the protocol does not allow direct interaction between customers and acquirers. Note that, being included in a secret message, the payment information is not captured by the Merchant.

- Step 4 - *Authorization Request*:
  - (a) the message received at 3.(a) by the Merchant is blindly forwarded to the Acquirer. The Merchant is not exposed to its content.
  - (c) the Merchant sends an authentic and secret message to the Acquirer containing (*TID, PurchAmt, HOD*), telling in this way "his own version of the story"
  - the Acquirer checks if the two versions of (*TID, PurchAmt, HOD*), provided independently by the Customer and the Merchant, are the same. If this is the case the Acquirer can be sure that the two principals have agreed on the terms of the transaction and then can verify the payment information – *PIdata* – originated by the customer. If the verification succeeds the payment can be authorized and the authorization code – *AuthCode* – is generated, otherwise the request is rejected *AuthCode* will include an error message explaining the reason of failure).

- Step 5 - *Authorization Response*: the Acquirer sends, in an authentic and secret message to the Merchant, the authorization code along with *TID* which identifies the transaction. The identity of the Customer is included among the verifiers to allow the forwarding of the message at step 6, maintaining the authenticity guarantee.

- Step 6 - *Purchase Response*: the Merchant forwards the message received from the Acquirer to the Customer. This is the notification of the result of the transaction. In this way *C* receives, via the Merchant, a proof of payment from the Acquirer. Since the message is signed by the Acquirer, the Merchant cannot alter the message without being detected. The structure of this message is quite different from the original *SET*. *HOD* is included to bind the digest of the order description and the digest of the credit card number, with the authorization code and the transaction ID.

2.3.1 Main Results of SET Security Verification

We verified the *AnBx* specifications of *signed* and *unsigned SET* purchase protocol and carried out a corresponding analysis of the original specifications, as reported in [14]. As for *iKP* we ran our tests assuming that the Acquirer is trusted. Indeed, though *SET* does not rely on such assumption, each customer can stop the protocol if she does not want to proceed with an Acquirer she does not trust. Therefore we can assume that, if the run of the protocol is completed, the customer decided to trust the Acquirer. For the signed version of the *SET* purchase protocol, the *AnBx* code of the revised and the original versions is shown in the Appendix (Tables A.4 and A.3).

certified agents	*M,A*		*C,M,A*	
	unsigned SET		*signed* SET	
Goal	O	R	O	R
pan(C) secret between *C,A*	-	-	+	+
C confidentially sends *pan(C)* to *A*	+	+	+	+
A weakly authenticates *C* on *pan(C)*	-	-	+	+
OrderDesc secret between *C,M*	+	+	+	+
PurchAmt secret between *C,M,A*	-	-	+	+
AuthCode secret between *C,M,A*	-	-	-	+
TID secret between *C,M,A*	-	-	-	+
HOD secret between *C,M,A*	-	-	-	+
M authenticates *A* on *AuthCode*	+	+	+	+
C authenticates *A* on *AuthCode*	-	+	-	+
C authenticates *M* on *AuthCode*	+*	-	+*	-
A authenticates *C* on *contract*	-	-	w	+
M authenticates *C* on *contract*	-	-	+	+
A authenticates *M* on *contract*	w	+	w	+
C authenticates *M* on *contract*	+	+	+	+
C authenticates *A* on *contract,AuthCode*	-	+	-	+
M authenticates *A* on *contract,AuthCode*	+	+	+	+

* goal satisfied only fixing step 5 as in [14]

w = only weak authentication

revised SET $\Rightarrow$ *contract* = *PriceAmt,LIDM,XID,*[*PIData:A*], [*OIData:M*]

original SET $\Rightarrow$ *contract* = *PriceAmt,LIDM,XID,*$\mathsf{hash}(PIData)$,$\mathsf{hash}(OIData)$

Table 2.4: Security goals satisfied by Original and Revised SET purchase protocol

In general, the security guarantees of our version of the *SET* purchase protocols outperform those of the original one both for the signed and the unsigned versions, as reported in Table 2.4. Similarly to what was already mentioned for *iKP* (Section 2.2) we benefit of the declarative nature of *AnBx* message primitives, which are defined as channel abstractions and offer stronger encapsulation mechanisms on the messages they exchange. It is worth noting that two known flaws affecting the *SET* original specification do not compromise our revised version.

The first flaw [14] involves the fifth step of the protocol: often in *SET* the signed messages lack of explicitness and therefore in step 5 it is not possible to link univocally the identity of the Acquirer and the Merchant with the particular transaction and authentication code. The original instruction

$A \rightarrow M : \{\{LIDM,XID,PurchAmt,AuthCode\}_{\mathsf{inv}(\mathsf{pk}(A))}\}_{\mathsf{pk}(M)}$

should then be amended to

$A \rightarrow M : \{\{M,LIDM,XID,PurchAmt,AuthCode\}_{\mathsf{inv}(\mathsf{pk}(A))}\}_{\mathsf{pk}(M)}$

otherwise the goal `C authenticates M on AuthCode` cannot be satisfied.

In our version the message at step 5 include both the identity of the Merchant and of the Customer. This happens because the payload is automatically compiled to a message that includes the following component:

$A \rightarrow M : \{\{M,C,LIDM,XID,PurchAmt,AuthCode\}_{\mathsf{inv}(\mathsf{pk}(A))}\}_{\mathsf{pk}(M)}$

Interestingly the same implementation also prevents the second flaw presented in [26]. In this thesis the specification of the protocol is more detailed than in [14] as it introduces an additional field $AuthRRTags$, which includes the identity of the Merchant. The attack is against the purchase phase and exploits a lack of verification in the payment authorization process. It may allow a dishonest Customer to cheat on the Merchant. This flaw is subtle: unlike the attack in [14], requiring the existence of a corrupted Acquirer, this attack requires only a collusion between a dishonest Merchant and a Customer. The attack is based on the fact that neither *LIDM* nor *XID* can be considered unique and they cannot be used to identify the Merchant. Therefore the customer can start a parallel purchase with an accomplice (playing the role of another merchant) . Both merchants generate their authorization requests and send them to the Acquirer, but the intruder intercepts the good merchant's request and destroys it. The Acquirer proceeds with the intruder's message where in the field $AuthRRTags$ the intruder has changed its identity with the identity of the good merchant.

Due to a faulty verification process (as specified by the *SET* documentation, and reported in [26]), the Acquirer does not compare the identity in $AuthRRTags$ with the other identities present in the message. The authorization message sent to the intruder at step 5 has the same structure of the message that the good merchant expects, so the Merchant can be convinced that he has been paid. Therefore the Merchant can then deliver the goods to the dishonest customer. The result is that the Acquirer has authorized the payment in favor of the intruder playing the role of a dishonest merchant. Full details about this attack can be found in [26]. As the authors suggest, the fix is to bind under the Acquirer signature the merchant's and the client's identities, and this is what our version does in step 5 and 6.

To check this solution, we tested the versions of *SET* presented in [26] and [14] with OFMC; we also verified that they also need to be fixed in the sixth (and final) step as already outlined in [79]. The identity of the Customer must be included in the message, otherwise the Customer cannot authenticate the Merchant on *AuthCode*. This issue also leads us to more interesting considerations on how to prove the authorization of the transaction.

Proving Authorization of the Transaction

Our general e-payment protocol (Section 2.1) defines $contract = price, tid, date, [can{:}A], [desc{:}M]$ as the terms of the transaction on which all participants should ideally agree. In the revised *SET* the equivalent term is

$PriceAmt,TID,[OIData{:}M],[PIData{:}A]$

which lacks only of the complementary information *date*. Expanding the definitions of $TID, OIdata$ and $PIdata$ we obtain

$contract = PriceAmt,LIDM,XID,[OrderDesc{:}M],[pan(C){:}A]$.

Analogously, in the original *SET* we can define

$contract = PriceAmt,LIDM,XID,\mathsf{hash}(PIData),\mathsf{hash}(OIData)$

Our analysis showed that some of the desired six authentication goals are not met by

the original signed *SET* (Table 2.4). A similar result, using a different analysis technique, was already described in [79], but we can show that amending our revised version of *SET*, it is possible to achieve all the six goals. The original problem arises from the fact that the Customer does not have an evidence of the origin of *AuthCode* by the Acquirer and she rather relies only on information provided by the Merchant. For example giving to C a proof that A authorized the payment requires substantial modification of the sixth step of the protocol. In fact, instead of letting the Merchant signing a message for C we exploit the *AnBx* forward mode to bring to the customer the authorization of the payment signed directly by the Acquirer. It is interesting to note that, employing the *fresh forward mode* in the sixth step, we can achieve the strong authenticity goal on $contract, AuthCode$, even though the transaction identifier is not unique (as in the second flaw [26]).

The resulting protocol is very similar to the general e-payment protocol, introduced in Section 2.1, and it confirms the results outlined in [79] showing that, while *iKP* meets all the non-repudiation of origin goals, the original specification of *SET* does not. It is important to notice that to achieve non-repudiation, each participant must have sufficient proofs to convince an external verifier that the transaction was actually carried out as she claims. A way to obtain this is to assume that the authentication is obtained by means of digital signatures computed with keys which are valid within a Public Key Infrastructure and are issued by a trusted third party (Certification Authority). In fact, such signatures can be verified not only by the intended recipient, but also by every other agent. Although this limits the way authentic channels in *AnBx* could be implemented, in practice it does not represent a significant restriction since, in the considered protocols, digital signature is the standard way meant to achieve authentication.

In summary we showed that revisiting *SET* in *AnBx* not only offers a simpler and clearer design, but also, compared to the original specification, stronger guarantees. The most interesting points are a new design of *SET* by means of *AnBx* primitives, produces a protocol that is immune to the known flaws which are present in the original *SET* and it is possible to fully prove authorization of transaction.

3

From AnBx to Executable Narrations

Narrations are a popular way to describe security protocols as a sequence of message exchanges among different principals. Despite being intuitive, this specification technique is semi-formal because it contains a lot of implicit concepts, leaving various aspects of the protocol vague or undefined. Moreover it just describes the ideal execution of a protocol, without saying what to do when something goes wrong. These facts cannot be simply ignored, in particular when translating narrations to real-world programming languages, because otherwise it would be difficult to program a computer to run such protocol.

For this purpose, a precise specification and a concrete semantics defining the translation from the source to the target language is necessary. Overcoming the limits of narrations requires both making assumptions, to explain some implicit facts, and adding further information to remove ambiguities. These aspects were already pointed out by Abadi [1], who identified four aspects that should be taken into account:

1. The *knowledge* of the principals before the protocol run should be stated explicitly. Additionally it is necessary to specify which values (and by whom) are freshly generated during the protocol execution. This is usually done adding a declarative preamble before the sequence of message exchanges.

2. The *checks* carried out by principals on reception of messages, should also be explicit. Usually an exchange like $A \rightarrow B : M$ is performed over an asynchronous insecure channel, which can be under control of an intruder [40]. Messages can be stopped, resent in a different order, or even forged by an attacker. Therefore agent B has to perform some informative checks on the received messages, with the aim to verify their consistency with respect to his current knowledge and the expected status of execution of the protocol.

3. *Concurrency*: despite the sequential idealized execution of the protocol, the principals behave in a concurrent way. This means that messages can be exchanged in a different order, due to the network conditions or depending on the behavior of the intruder.

4. Several *sessions* of the protocol can run in parallel, but narrations does not consider this fact, since a narration tells the story of a single session.

In this work we mainly focus on the first two issues, describing the translation from the abstract language *AnBx* (Chapter 1), to a concrete real-world programming language, namely Java. In fact, to generate and correctly run a Java program, any potential ambiguity must be removed. For this purpose, some additional information or assumption is needed. Beside the issues related with the modelling of the knowledge and the checks on reception, we should

consider, for instance, that *Alice & Bob narrations* are in general written in an untyped language, while Java is statically typed.

The automatic Java code generation of security protocols specified in *AnBx* is a process comprising several phases. A pictorial view of the whole process is shown in Figure 3.1 and 3.2. The work done by our tool can be summarized as follows:

1. *AnBx* $\rightarrow$ *AnB* $\rightarrow$ (*verification*): the *AnBx* protocol is compiled to *AnB* and then verified (model-checked) with OFMC [11].

2. *AnB* $\rightarrow$ *SpyerPN* $\rightarrow$ *Execnarr* $\rightarrow$ *Opt-Execnarr*: if the protocol is deemed safe, the *AnB* protocol is compiled to *SpyerPN*[1], a protocol narration language introduced in [25], and then to two subsequent intermediate formats: *executable narration* (*Execnarr*) and *optimized executable narration* (*Opt-Execnarr*). As a by-product, inherited from the `spyer` tool [25], an output to *Spi Calculus* is also available.

3. *Opt-Execnarr* $\rightarrow$ (*protocol logic*) + (*application logic*) $\rightarrow$ *Java*: the final outcome is the generation of the Java source code from the *optimized executable narration*. The process keeps apart the protocol logic from the application logic until code is actually generated.

We focused on the first phase of the process in chapters 1–2. The core of the second phase, described in the current chapter, is the generation of the informative checks, by means of an extension of the `spyer` tool which automatically generates the checks derivable from the static information of protocol narrations.

The underlying theory is a formal operational semantics for protocol narrations, proposed by Briais and Nestmann [25], that gives an interpretation on how the protocol participants are supposed to execute the protocol. The resulting processed narrations are called *executable*. The checks are expressed by means of consistency formulas and given the high number of generated formulas, the tool applies some simplification strategies, providing good results in practice. By contrast, although the *AnB* semantics [59] includes the notion of *consistency checks*, it does not provide an algorithm to compute in practice a finite sufficient set of checks. In fact, it could lead to an infinite set of checks, and "it is not clear if such a finite sufficient set exists in general" [59]. Although finding the solution to this problem can be interesting and useful, it is out of the scope of this work.

For these reasons, in the first part of this work (section 3.1), we decided to adopt the solution proposed by Briais and Nestmann making some extensions to address the following issues:

- Although the *SpyerPN* language shares many concepts with *AnB*, it exhibits some significant differences that require the implementation of a translator from *AnB* to *SpyerPN* (Section 3.1.2).

- The *SpyerPN* language is less expressive than *AnB*. For example, it does not support HMACs, Diffie-Hellmann key agreements, tuples and functions. Therefore we extended its semantics to support these primitives. In this way it is possible to model a wider and more realistic class of protocols.

[1] We refer to the language in which the protocol narrations are written in [25], with the name *SpyerPN*. This is the input language of the tool `spyer`.

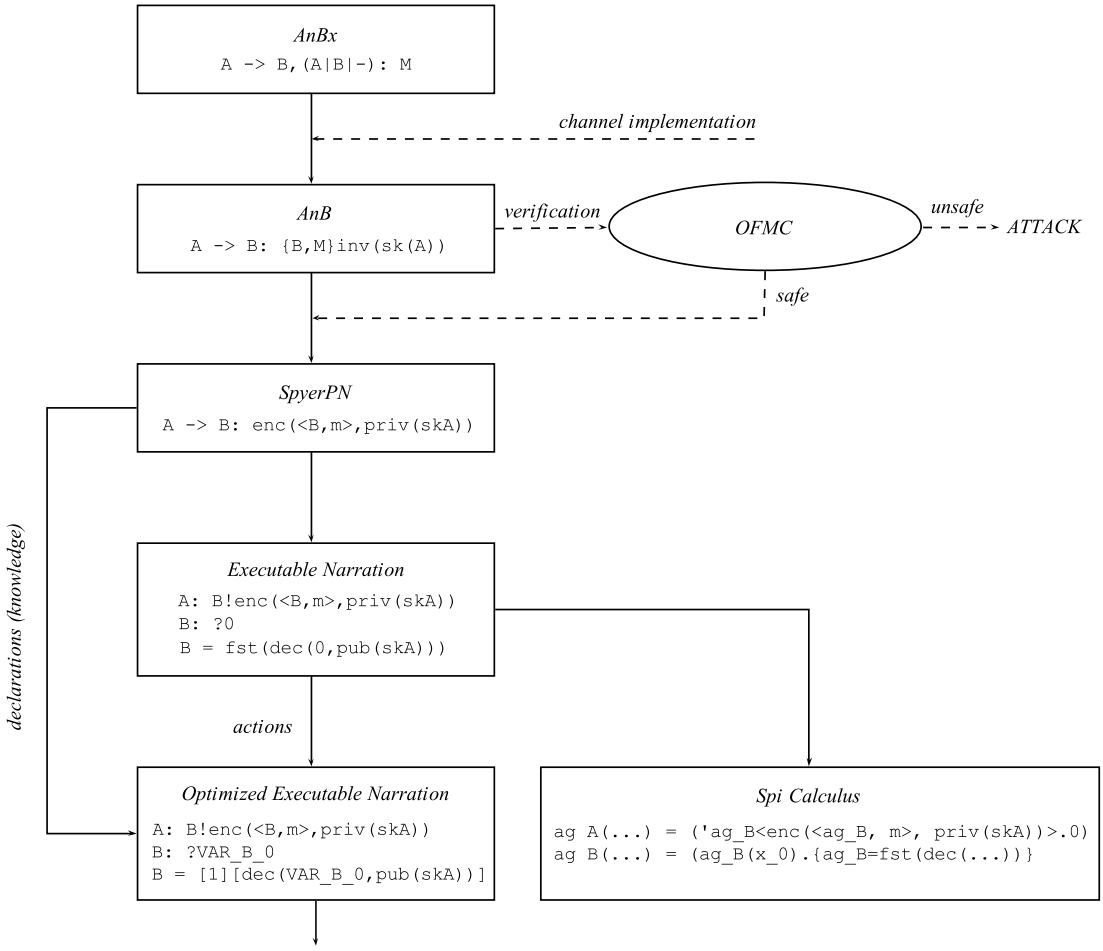

Figure 3.1: From *AnBx* to optimized executable narrations

- The *executable narration* code is not optimized, hence its translation to a real program would not be efficient. For example, the same cryptographic operations, e.g. encryption and decryption, could be performed, on the same data, many times during the protocol run. We identified the set of cryptographic operations, which in general are computationally expensive, and optimized the code to reduce the overall execution time, introducing variables storing partial results, and making some reordering with the purpose of minimizing the number of cryptographic operation performed. Moreover we delayed the generation of fresh values until they are really used, while in the executable narrations they are all created at the beginning. Our experimental results show that these optimizations (Section 3.2) reduce the execution time by 30-40% for the revised versions of the e-commerce protocols belonging to the *iKP* family (see Chapter 2).

To help the reader to get the whole picture we anticipate here that in chapter 4 we are going to describe the third and final phase, the generation of the Java source code from the optimized executable narrations (Figure 3.2). The solution we propose goes beyond the specific issues of Java, devising a strategy that we think can also be applied, with a reasonable effort, to other object oriented and procedural programming languages. The code generation, comprising

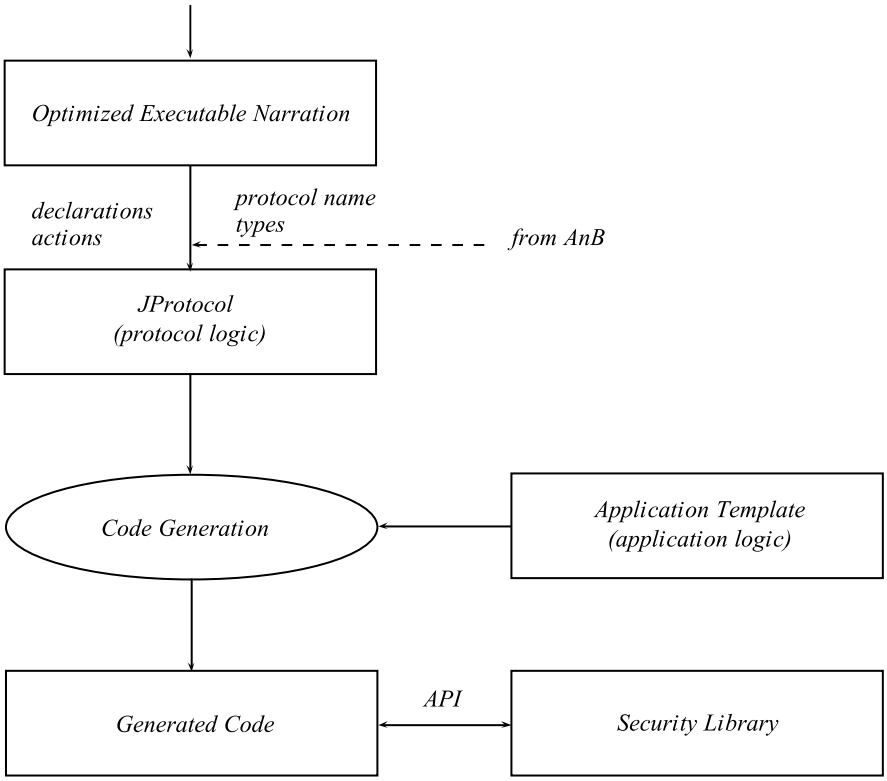

Figure 3.2: Code generation

several steps, required to address some interesting issues:

1. The definition of a code generation strategy. First of all, we make a distinction between the *protocol logic* and the *application logic*. The latter is implemented by means of a parametrized application template written in the target language (Section 4.1.3) which is instantiated with the information – *the protocol logic* – derived from the optimized executable narration. We model the *protocol logic* by means of a format called *JProtocol* (Section 4.1.1) which is still independent from the target language. It is important to underline that the application template is generic, i.e. independent from the specific protocol. This allows the tool to work as a one-click code generator because it can directly produce a runnable application from the *AnBx* specification.

2. The definition of a typed abstract representation of the security related portion of a generic procedural language. This also required the design of a type system (Section 4.1.2) to infer the type of expressions and variables. The correct implementation of the type system is necessary to avoid compile time errors in the generated code. Another issue to consider was the fact that the set of types in the *AnBx* is too limited to be directly and efficiently translated into a real programming language.

3. The design of an API for security (Section 4.2) exposing some of the cryptographic operations offered by the Java Cryptographic Architecture (JCA) [44,71]. The library wraps the JCA interface and implements the custom classes necessary to code the generated programs in Java. The library offers an high degree of generality and customization,

since the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Moreover the library provides the communication primitives used to exchange messages in the target network environment, the TCP/IP standard in our case.

4. The generation of the Java source code (Section 4.1.4) is performed instantiating the protocol template, i.e. the skeleton of the application, with the information derived from the protocol logic. It is worth noting that only at this stage, when the code is generated, the language specific features and their API calls are actually bound to the protocol logic.

Due to the complexity of the target language, we do not claim the formal correctness of the last part of the translation chain, from the executable narration to Java. However we think that this experimental work offers some interesting insights in the topic of security protocol design and automatic generation of implementations.

First of all, we show the effectiveness of *AnBx* as a language not only for abstract protocol prototyping but also for the generation of concrete implementations. With respect to some existing tools [58, 73] we produce Java code which automatically generate the checks on reception. Moreover in contrast to other tools [72, 78], using the Spi Calculus as specification formalism, we use a higher-level and more intuitive language, making our compiler more appealing to a wider spectrum of potential users. Additionally we propose a Java library for security which can be used not only with the code generated from *AnBx*, but also to code standard Java programs, even by programmers with a limited background in security. We discuss these issues and the related works in the final section (4.3) of chapter 4.

3.1 Compiling *AnB/AnBx* into Formalized Executable Narrations

In this section we show how protocols in *AnB* and *AnBx* can be compiled into executable narrations following the ideas presented in [24, 25]. First of all we explain how the agent knowledge is modeled in these languages (3.1.1) and how protocols can then be translated to *SpyerPN* (3.1.2). Next we recall the procedure proposed in [25] to compute the informative checks on reception of messages (3.1.4) showing how to extend it in order to include all *AnB/AnBx* language features which are not handled by `spyer`. In the next section, we propose some optimizations (3.2) that are useful for the translation to a real programming languages.

3.1.1 Modeling the Knowledge of the Agents

Before we proceed, it is useful to remind that a protocol specification in *AnB* and *AnBx* comprises four mandatory sections:

- `Types`: describes the entities (agents/principals) involved in the protocol, along with the protocol data and the operators on them, including the cryptographic functions;
- `Knowledge`: specifies the initial knowledge of each principal;
- `Actions`: specifies the sequence of statements that constitute the ideal, unattacked run of the protocol;

```
Protocol: Fresh_From_A
Types:
  Agent A,B;
  Certified A;
  Number Msg
Knowledge:
  A: A,B;
  B: A,B
Actions:
  A -> B,@(A|B|-):Msg
Goals:
  B authenticates A on Msg
```

Figure 3.3: *AnBx* specification of an authentic (fresh) exchange

```
Protocol: Fresh_From_A
Types:
  Agent A,B;
  Number Msg,N1;
  Function pk,sk,hash
Knowledge:
  A: A,B,pk,sk,inv(pk(A)),inv(sk(A));
  B: A,B,pk,sk
Actions:
  A -> B: A
  B -> A: {N1,B}pk(A)
  A -> B: {N1,hash({B,Msg}inv(sk(A)))}pk(B),{B,Msg}inv(sk(A))
Goals:
  B authenticates A on Msg
```

Figure 3.4: A challenge-response implementation in *AnB* of the protocol of Figure `3.3`

- `Goals`: specifies the goals that the protocol is meant to convey.

An example of a simple *AnBx* protocol and its translation in *AnB* is given in Figures 3.3 and 3.4. The full syntax of these languages is shown in Tables 4.7 and 4.8.

In Chapter 1 we already explained the main enhancements of *AnBx* with respect to *AnB*. Here we detail on some syntactic features introduced in *AnBx* in the sections `Types` and `Knowledge`.

Types In this section of an *AnB* protocol, the involved agents (type `Agent`) and the functions used (type `Function`) are specified. Moreover the type `Number` identifies constants and values which are freshly generated. The distinction between constants and fresh values is done by means of the first letter of the identifier (lowercase and uppercase respectively). Types `PublicKey` and `Symmetric_key` designate variables of the eponymous type.

In *AnBx* we introduced two additional types:

1. `SeqNumber` identifies the fresh values which are expected to be received by an agent no more than once during the protocol execution. An example is the transaction identifier

in an e-commerce protocol.

2. `Certified` denotes the agents possessing the public/private key pairs for encryption and signature. This is a syntactic sugar that simplifies the notation, making redundant writing in the knowledge section the keys possessed by those agents. Such information is added by the *AnBx* compiler to the agent's knowledge, along with some cryptographic functions which are used during the protocol execution (compare Figures 3.3 and 3.4). Moreover the notion of certified agent becomes handy in the translation process to real-world programming languages, where we deal with cryptographic objects like certificates and keys (see Section 4.1).

We should observe that in *AnB* the notion of which agent generates the fresh values is implicit. In the `Type` section the fresh values are simply listed without any reference to the agent who is generating them (for example variables `N1` and `Msg` in Figure 3.4). Here the underlying assumption is that these values are freshly generated by the agent that uses them first. This is coherent with the semantics of OFMC [11], i.e. the translation from *AnB* to IF [10]. In fact in IF the freshly generated values are added to the current knowledge of the agent who introduces the variable in the protocol.

Knowledge This section is used to specify the initial knowledge of each principal. The original specification of *AnB* permits to include in the initial knowledge, agents names, constants, functions, and their combination. Freshly generated values are not allowed since they must be actually generated during the protocol execution. Although this is fully reasonable, it could be a problem in protocols where the initial knowledge should include some non constant pre-shared values.

For example, in e-commerce protocols like *iKP* [15, 16] and *SET* [13], the description of the payment process, assumes that the price and the order details are agreed between the customer C and the merchant M before the protocol execution.

This schema is supported by *AnBx* by means of a new construct in section `Knowledge` allowing the specification of values shared among a set of agents as in the statement `C,M share Price,Desc`, which models the aforementioned scenario.

3.1.2 From *AnB* to *SpyerPN*

Now we show how to translate an *AnB* protocol to a narration in *SpyerPN* (the syntax is shown in Table 3.1).

A protocol in *SpyerPN* is composed by two sections. The first one, named *declaration,* is a header of the actual *narration* and includes the initial knowledge of each agent, the names generated by them and the names that are assumed to be initially known only by a subset of agents. The latter is similar to the `share` construct we introduced in *AnBx*, and it is useful to simulate a first pass where shared values are securely distributed among some agents.

The *agents* are taken from set of agent names $\mathbf{A}$ and the *messages* are built upon set of names $\mathbf{N}$. It is assumed that $\mathbf{A} \bigcap \mathbf{N} = \oslash$.

To handle asymmetric cryptography, the inverse key $inv(M)$ of a message M is defined as follows:

messages **M**			
M, N	::=	a	*name*
	|	A	*agent name*
	|	$hash(M)$	*hashing*
	|	$pub(M)$	*public key*
	|	$priv(M)$	*private key*
	|	$(M.N)$	*pair*
	|	$enc(M, N)$	*encryption*
	|	$hmac(M, N)$	*hmac* *
	|	$kap(M, N)$	*key agreement half key* *
	|	$kas(M, N)$	*key agreement full key* *
	|	$M(N)$	*function* *
exchanges			
T	::=	$A \to B : M$	*exchanges*
narrations			
L	::=	ϵ	*empty narration*
	|	$T; L$	*non empty narration*
declarations			
D	::=	A **knows** M	*initial knowledge*
	|	A **generates** n	*fresh name generation*
	|	**private** k	*private name*
protocol narrations			
P	::=	$D; P$	*sequence of declarations*
	|	L	*narration*

Table 3.1: Syntax of *SpyerPN* protocol narrations (plus extensions *)

$$inv(M) = \begin{cases} pub(M') & if\ M = priv(M') \\ priv(M') & if\ M = pub(M') \\ \bot & otherwise \end{cases}$$

The agents can verify if two messages M_1 and M_2 are inverse keys one of each other, trying to decrypt with M_2 a message encrypted with M_1 and conversely.

The statement **private** k means that k is a name which is initially only available for the agents involved in the protocol. For instance, this is useful to simulate that an agent A and a server S initially share a secret key kAS:

private kAS
A **knows** kAS
S **knows** kAS

A **knows** m denotes that, initially, the agent A knows the message m. The statement A **generates** n implies that the agent A generates a fresh name n (a nonce or a freshly generated key). All fresh names must be declared explicitly.

The meaning of actions in section *narration* is intuitive. $A \to B : M$ denotes that the agent A sends a message M to the agent B. Messages are built according to the syntactic rules

shown in Table 3.1. With respect to the original *SpyerPN* syntax, we introduced the support of operators like *hmac*, *kap*, *kas* and generic functions. *kap* and *kas* are used to model the basic operations on keys which are available in key agreement protocols like Diffie-Hellman. These functions are borrowed from [36]:

- $kap(g, x)$ is the half key computed from secret x
- $kas(k, y)$ is the full key computed from an half-key k and a secret y

They satisfy the algebraic property $kas(kap(g, x), y) \approx kas(kap(g, y), x)$, given the pre-shared parameter g.

Agent and Name conventions As a convention, the syntax of *SpyerPN* distinguishes elements of **A** and **N**, by means of the first letter of the identifier, upper and lowercase respectively. This contrasts with the syntactic rules of *AnB*. We handle such differences and later, when generating the Java code, we restore the original case of the identifiers.

Public Keys conventions *SpyerPN* offers two operators for modeling public and private keys, *pub* and *priv* respectively. For instance $(pub(A), priv(B))$ denotes the public/private key pair of agent A. However since in a Public Key Infrastructure (PKI) it is customary to use one key pair for signing and another distinct key pair for encryption. *AnBx* supports this schema by means of the `pk` and `sk` functions (Figure 3.4). Key pairs are translated to *SpyerPN* as $(pub(pkA), priv(pkA)$ and $(pub(skA), priv(skA)$ respectively.

3.1.3 Protocol Translation

The first step is to define a function $\tau : \mathbf{M_{AnB}} \rightarrow \mathbf{M}$, the translation of *AnB* messages to their equivalent in *SpyerPN*, where $\mathbf{M_{AnB}}$ and $\mathbf{M}$ are the sets of messages in the two languages (Table 3.2). To compute τ, we also need the type information available from the protocol header. For this purpose, it is necessary to denote, beside the set of identifiers **Ident**, the subsets of identifiers of a given type: **Agent**, **Number**, **SeqNumber**, **Function**, **PubkicKey** and **Symmetric_key.**

AnB messages of type **Ident** are translated to the equivalent identifiers in *SpyerPN*, adjusting the names and agent according to the name conventions mentioned before. The other messages are translated following the rules shown in Table 3.2. Again, here public and private keys are mapped according to the key name convention seen earlier. Finally tuples in *AnB* are mapped to nested ordered pairs in *SpyerPN* because the target language does not support tuples, but just pairs.

Declarations - Share statements As we have seen in 3.1.1, that statements like

```
A,B share Msg
```

can be used in section `Knowledge` of *AnBx* protocols, to model pre-shared information among agents. In general an *AnBx* knowledge statement $A_1, .., A_n$ `share` M, where A_i are the agent names and M is a message, will be equivalent to the *SpyerPN* declarations (we do not show here the intermediate statements in *AnB*):

private $\tau(M)$
A_1 **knows** $\tau(M)$

$$\tau(\cdot) : \mathbf{M_{AnB}} \rightarrow \{\perp\} \cup \mathbf{M}$$

$\tau(a)$	:=	$toUpper(a)$	$if\ a \in$ **Agent**
$\tau(a)$	:=	$toLower(a)$	$if\ a \in$ **Ident** $\wedge\ a \notin$ **Agent**
$\tau(\{\|m\|\}_k)$	:=	$enc(\tau(m), \tau(k))$	
$\tau(\{m\}_k)$	:=	$enc(\tau(m), \tau(k))$	
$\tau(op(a))$	:=	$pub(op + \tau(a))$	$if\ a \in$ **Agent** $\wedge\ op \in$ **Function** $\wedge\ op \in \{pk, sk\}$
$\tau(inv(op(a)))$	:=	$priv(op + \tau(a))$	$if\ a \in$ **Agent** $\wedge\ op \in$ **Function** $\wedge\ op \in \{pk, sk\}$
$\tau(exp(g, x))$	:=	$kap(\tau(g), \tau(x))$	$if\ g, x \in$ **Number**
$\tau(exp(exp(g, x), y)$	:=	$kas(kap(\tau(g), \tau(x)), \tau(y))$	$if\ g, x, y \in$ **Number**
$\tau(hmac_k(m))$	:=	$hmac(\tau(m), \tau(k))$	$if\ k \in$ **Symmetric_key**
$\tau(hash(m))$	:=	$hash(\tau(m))$	
$\tau(f(x))$	:=	$F(\tau(x))$	$if\ \tau(f) = F \in \mathbf{M} \wedge\ f \in$ **Function**
$\tau(E)$	:=	$\perp$	$otherwise$

Table 3.2: Translation of *AnB* messages to *SpyerPN* (+ is the concatenation of names)

$\vdots$

A_n **knows** $\tau(M)$

The first statement is generated only if M is an identifier, but not an agent name. Recall that *SpyerPN* allows declaring **private** only the names, but not any other kind of message.

Declarations - Types Identifiers of type `Number`, `SeqNumber`, `Symmetric_key` and `PublicKey` in *AnB* are taken into account to generate declarations in *SpyerPN*. First of all, it should be noted that the *AnB* `Types` declaration does not contain an explicit information to identify the agent who generates or knows the declared variable or constant. Therefore, first we have to analyze the actions of the protocols to discover the name of the agent who uses the identifier first. Second, we must detect if an identifier represents a value which is a freshly generated during the protocol execution or not. This is done checking if an identifier is a variable or not, with the exception of pre-shared private names. If the answer is affirmative, we add a **generate** fact, otherwise we add a **knows** fact. For example, the code fragment

```
Types
Number K1,M,z
```

will be translated to

A_i **generates** $k1$
A_j **knows** m
A_k **knows** z

where A_i, A_j, A_k are the first agents to use each identifier[2]. We assume here that m is a pre-shared value and therefore a **knows** fact is generated.

[2] $k1$ and m are now lowercase according to the syntactical conventions of *SpyerPN*.

Declarations - Knowledge The mapping of the knowledge is rather straightforward. We analyze the protocol section `Knowledge` and for every agent A_i we create a fact A_i **knows** m for each message m included in the agent knowledge. Messages are translated by means of the function τ. Some attention is required to map the public/private keys possessed by each agent. Since in *AnBx* we have the notion of certified agents, we assume that their names and public keys are available to all agents. Moreover we make the reasonable assumption that a certified agent knows his own private key.

For example, this fragment of *AnB* code

```
Knowledge
A: A,B,hash,pk,pk(A),inv(pk(A)),pk(B)
```

is translated to *SpyerPN* as follows

A **knows** A
A **knows** B
A **knows** hash
A **knows** pk
A **knows** pub(pkA)
A **knows** priv(pkA)
A **knows** pub(pkB)

Narrations Each *AnB* action on a plain channel $A \to B : M$ is simply translated to an equivalent action $A \to B : \tau(M)$. Actions on other channel types are not translated and an error is raised. This is not a limitation since it is always possible to use the *AnBx* channels which are compiled in *AnB* in actions over a plain channel.

3.1.4 Compiling Protocol Narrations

Having translated the protocol from *AnB* to *SpyerPN*, it is now possible to compute the checks on reception applying the ideas proposed by Briais and Nestmann [25]. In this section we summarize their approach giving an overview of the issues related to the translation process from *SpyerPN* to *executable narrations* (Table 3.3). This representation will also be used as the input format for the next step in code generation.

Exchanges in the executable narrations (set **X**, in Table 3.3) are decomposed making more explicit the behavior of every single agent. Atomic exchanges of the form $A \to B : M$ can hence be compiled to four more specific basic actions (non-terminal I in Table 3.3):

1. *emission* $A : B!E$ of a message expression E (evaluating to M);
2. *reception* $B : ?x$ of a message and its binding to a variable x;
3. *check* $B : \phi$ for the validity of the formula ϕ from the point of view of agent B;
4. *scoping* νk, represents the creation and scope of private names. Scoping is decoupled from agent identities, allowing to use a single construct for names that are **private** and **generated** according to the declarations in *SpyerPN*.

When an agent receives a message, he binds that message to a fresh variable for reference in subsequent processing. For this purpose, a set $x, y, z, ...$ of variables **V** is introduced. Such set is assumed to be disjoint from $\mathbf{N} \cup \mathbf{A}$.

expressions $\mathbf{E}$			
E, F	$::=$	a	*name*
	$\mid$	A	*agent name*
	$\mid$	x	*variable*
	$\mid$	$hash(E)$	*hashing*
	$\mid$	$pub(E)$	*public key*
	$\mid$	$priv(E)$	*private key*
	$\mid$	$(E.F)$	*pair*
	$\mid$	$\pi_1(E)$	*first projection*
	$\mid$	$\pi_2(E)$	*second projection*
	$\mid$	$enc(E, F)$	*encryption*
	$\mid$	$dec(E, F)$	*decryption*
	$\mid$	$hmac(E, F)$	*hmac* *
	$\mid$	$kap(E, F)$	*key agreement half key* *
	$\mid$	$kas(E, F)$	*key agreement full key* *
	$\mid$	$E(F)$	*function* *
formulae $\mathbf{F}$			
ϕ, ψ	$::=$	$[E = F]$	*matching*
	$\mid$	$[E : \mathbf{M}]$	*well − formedness test*
	$\mid$	$inv(E, F)$	*inverse key test*
		$\phi \wedge \phi$	*conjunction*
		tt	*always true*
simple actions $\mathbf{I}$			
I	$::=$	νk	*fresh name generation*
	$\mid$	$A : B!E$	*message emission*
	$\mid$	$A :?x$	*message reception*
		$A : \phi$	*checks*
executable narrations $\mathbf{X}$			
X	$::=$	ϵ	*empty narration*
	$\mid$	$I; X$	*non empty narration*

Table 3.3: Syntax of executable narrations (plus extensions *)

Since an agent does not only handle messages but also variables, the notion of *message expressions* ($\mathbf{E}$) is introduced, along with the operations to construct and deconstruct messages. Messages received during the protocol execution, and stored in variables x, are closely related to the statically intended messages M described in the narration. For this reason, bindings $(M, x) \in \mathbf{M} \times \mathbf{E}$ are used.

The process of finding out whether some expression represents a particular message, is formalized by means of an evaluation function which is shown in Table 3.4. Note that if an expression contains variables the evaluation fails.

Formulas ϕ on received messages are described by a conjunctions of three kinds of checks:

- *equality tests* $[E = F]$ on expressions denoting the comparison of two bit-streams of E and F;
- *well-formedness tests* $[E : \mathbf{M}]$ denoting the verification of whether the projections and

$$[\![\cdot]\!] : E \to \{\bot\} \cup \mathbf{M}$$

$$\begin{array}{rcll}
[\![E]\!] & := & E & if\ E \in \mathbf{N} \cup \mathbf{A} \\
[\![(E.F)]\!] & := & ([\![E]\!].[\![F]\!]) & \\
[\![\pi_1(E)]\!] & := & M & if\ [\![E]\!] = (M.N) \in \mathbf{M} \\
[\![\pi_2(E)]\!] & := & N & if\ [\![E]\!] = (M.N) \in \mathbf{M} \\
[\![enc(E,F)]\!] & := & enc([\![E]\!],[\![F]\!]) & \\
[\![dec(E,F)]\!] & := & M & if\ [\![E]\!] = enc(M,N) \in \mathbf{M} \wedge [\![F]\!] = N \in \mathbf{M} \\
[\![op(E)]\!] & := & op([\![E]\!]) & if\ op \in \{pub, priv, hash\} \\
[\![op(E,F)]\!] & := & op([\![E]\!],[\![F]\!]) & if\ op \in \{hmac, kas, kap\} \\
[\![E(F)]\!] & := & M([\![F]\!]) & if\ [\![E]\!] = M \in \mathbf{M} \\
[\![E]\!] & := & \bot & otherwise
\end{array}$$

$$[\![\cdot]\!] : F \to \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{array}{rcll}
[\![tt]\!] & := & \mathbf{true} & \\
[\![\phi \wedge \psi]\!] & := & \mathbf{true} & if\ [\![\phi]\!] = [\![\psi]\!] = true \\
[\![[E = F]]\!] & := & \mathbf{true} & if\ [\![E]\!] = [\![F]\!] = M \in \mathbf{M} \\
[\![[E : \mathbf{M}]]\!] & := & \mathbf{true} & if\ [\![E]\!] = M \in \mathbf{M} \\
[\![inv(E,F)]\!] & := & \mathbf{true} & if\ [\![E]\!] = M \in \mathbf{M} \wedge [\![F]\!] = inv(N) \in \mathbf{M} \\
[\![\phi]\!] & := & \mathbf{false} & otherwise
\end{array}$$

Table 3.4: Definition of the evaluation of expressions and formulas

decryption contained in E are likely to succeed;

- *inversion tests* $inv(E, F)$ denoting the verification that E and F evaluate to inverse messages.

The evaluation function is extended to formulas; it can be seen that, $[E : \mathbf{M}]$ is just a macro for $[E = E]$. Similarly, $inv(E, F)$ can be encoded (for example) as $[dec(enc((E.F), E), F) : \mathbf{M}]$.

Since consistency checks will have to operate on *(message,expression)* pairs, the representation of the agent knowledge must be generalized to finite subsets of $\mathbf{M} \times \mathbf{E}$. The underlying idea is that a pair (M, E) denotes that an expression E is equivalent to the message M.

For this reason is it necessary to introduce the notion of *knowledge sets*, and two operations on them:

- *synthesis* reflecting the closure of knowledge sets using message constructors;
- *analysis* reflecting the exhaustive recursive decomposition of knowledge pairs on as enabled by the currently available knowledge.

Formally these sets and operations are defined as follows:

Definition 1 (Knowledge)

- *Knowledge sets* $K \in \mathbf{K}$ are finite subsets of $\mathbf{M} \times \mathbf{E}$.
- The *synthesis* $\mathcal{S}(K)$ of K is the smallest subset of $\mathbf{M} \times \mathbf{E}$ containing K and satisfying the SYN-rules in Table 3.5.

$$\textsf{SYN-OP1}\ \frac{(M,E)\in\mathcal{S}(K)}{(op(M),op(E))\in\mathcal{S}(K)}\ op\in\{pub,priv,hash\}$$

$$\textsf{SYN-OP2}\ \frac{(M,E)\in\mathcal{S}(K)\qquad(N,F)\in\mathcal{S}(K)}{(op(M,N),op(E,F))\in\mathcal{S}(K)}\ op\in\{enc,hmac,kas,kap\}$$

$$\textsf{SYN-PAIR}\ \frac{(M,E)\in\mathcal{S}(K)\qquad(N,F)\in\mathcal{S}(K)}{((M.N),(E.F))\in\mathcal{S}(K)}$$

$$\textsf{SYN-FUN}\ \frac{(M,E)\in\mathcal{S}(K)\qquad(M,F)\in\mathcal{S}(K)\qquad M\in\mathbf{N}}{(M(N),(E(F))\in\mathcal{S}(K)}$$

$$\textsf{SYN-KAP}\ \frac{(M,E)\in\mathcal{S}(K)\qquad(N,F)\in\mathcal{S}(K)\qquad M\in\mathbf{N}}{(kap(M,N),kap(E,F))\in\mathcal{S}(K)}$$

$$\textsf{SYN-KA-EQ}\ \frac{(kas(kap(M,N),O),kas(kap(E,F),G)\in\mathcal{S}(K)\qquad M\in\mathbf{N}}{(kas(kap(M,O),N),kas(kap(E,G),F)\in\mathcal{S}(K)}$$

Table 3.5: Synthesis SYN-rules

- The *analysis* $\mathcal{A}(K)$ of K is $\bigcup_{n\in\mathbb{N}}\mathcal{A}_n(K)$ where the sets $\mathcal{A}_i(K)$ are the smallest sets satisfying the ANA-rules in Table 3.6.

With respect to the original work [25] we added the SYN-rules SYN-OP2, SYN-FUN, SYN-KAP, SYN-KA-EQ and the ANA-rules ANA-OP2, ANA-FUN. These new rules are necessary to generalize the notion of synthesis and analysis with functions and operators defined in *AnB/AnBx*, and previously unavailable in *SpyerPN*. It is worth noting that the SYN-KA-EQ rule is necessary to model the algebraic equivalence $(kas(kap(g,x),y)\approx(kas(kap(g,y),x)$.

The above knowledge representation allows generating the checks required on message reception. Assuming that agents have some initial knowledge (usually of the form (M,M) where M are the messages corresponding to the known facts as stated in the declaration section) the knowledge set is extended during the protocol execution, according to the information collected during the reception actions: the expected message and the corresponding expression. The checks must verify:

1. if the expectation of the recipient on the received message (as expressed statically in the narration) is matched by the recipient's current knowledge;

2. if the knowledge increase obtained receiving the message is consistent with the previously acquired knowledge.

Some checks depend on the structure of messages: for instance, if an agent receives an encrypted message he should be able to decrypt the corresponding expression if he knows the correct key.

Other checks result instead from the fact that a message M may occur more than once in a protocol narration. In this case the same message M could be associated to two different expressions E_1 and E_2. Since the term M is used in the protocol specification to refer to the

$$\textsf{ANA}-\textsf{INI}\ \frac{(M,E)\in K}{(M,E)\in\mathcal{A}_0(K)}$$

$$\textsf{ANA}-\textsf{OP1}\ \frac{(op(M),E)\in\mathcal{A}_n(K)}{(op(M),E)\in\mathcal{A}_{n+1}(K)}\ op\in\{pub,priv,hash\}$$

$$\textsf{ANA}-\textsf{OP2}\ \frac{(op(M,N),E)\in\mathcal{A}_n(K)}{(op(M,N),E)\in\mathcal{A}_{n+1}(K)}\ op\in\{hmac,kap,kas\}$$

$$\textsf{ANA}-\textsf{FUN}\ \frac{(M(N)),E)\in\mathcal{A}_n(K)}{((M(N)),E)\in\mathcal{A}_{n+1}(K)}$$

$$\textsf{ANA}-\textsf{FST}\ \frac{((M.N),E)\in\mathcal{A}_n(K)}{((M,\pi_1(E))\in\mathcal{A}_{n+1}(K)}$$

$$\textsf{ANA}-\textsf{SND}\ \frac{((M.N),E)\in\mathcal{A}_n(K)}{((N,\pi_2(E))\in\mathcal{A}_{n+1}(K)}$$

$$\textsf{ANA}-\textsf{DEC}\ \frac{(enc(M,N),E)\in\mathcal{A}_n(K)\qquad(inv(N),F)\in\mathcal{S}(\mathcal{A}_n(K))}{(M,dec(E,F)\in\mathcal{A}_{n+1}(K)}$$

$$\textsf{ANA}-\textsf{DEC}-\textsf{REC}\ \frac{(enc(M,N),E)\in\mathcal{A}_n(K)\qquad(inv(N),F)\notin\mathcal{S}(\mathcal{A}_n(K))}{(enc(M,N),E)\in\mathcal{A}_{n+1}(K)}$$

$$\textsf{ANA}-\textsf{NAM}-\textsf{REC}\ \frac{(M,E)\in\mathcal{A}_n(K)\qquad M\in\mathbf{N}\cup\mathbf{A}}{(M,E)\in\mathcal{A}_{n+1}(K)}$$

Table 3.6: Analysis ANA-rules

very same message, the current knowledge set can be considered consistent only if the two expressions evaluate to the same message. In case of asymmetric keys, it can also happen that, in some knowledge set, there is a combination of (M_1,E_1) and (M_2,E_2) where $M_1=inv(M_2)$. In this case, $inv(E_1,E_2)$ should also be satisfied.

These requirements are formalized in the definition of consistency formula:

Definition 2 (Consistency Checks) *Let K be a knowledge set. Its consistency formula $\Phi(K)$ is defined as follows:*

$$\begin{aligned}\Phi(K)\ :=\quad & \textstyle\bigwedge_{(M,E)\in K}[E:\mathbf{M}]\\ \wedge\ & \textstyle\bigwedge_{(M,E_i)\in K\wedge(M,E_j)\in\mathcal{S}(K)\wedge E_i\neq E_j}[E_i=E_j]\\ \wedge\ & \textstyle\bigwedge_{(M,E_i)\in K\wedge(inv(M),E_j)\in\mathcal{S}(K)} inv\,(E_i,E_j)\end{aligned}$$

The first conjunction clause checks that all expressions can be evaluated, the second checks that if there are several ways to build a message M, then all the corresponding expressions must evaluate to the same value. The third conjunction clause checks that if it was possible to generate a message M and its inverse $inv(M)$, then the corresponding expressions must also be mutually inverse.

The generation of the consistency formulas, implies comparing pairs taken from K with pairs taken from $\mathcal{S}(K)$. It can be shown that comparing pairs only in K is not sufficient. On

the other hand, comparing any possible combination of pairs taken from $\mathcal{S}(K)$, would lead to an infinite formula. For this reason, knowledge sets can often be simplified without loss of information, i.e. without undermining the computation of the consistency formula (for the details refer to [25]).

Compilation The above notions are the elements required to compile a *SpyerPN* narration into an executable protocol narration. The translation function keeps track of the global information on the used variables and hidden names, as well as the agent local information, about their knowledge on generated names.

In detail, statements like **private** k and A **generates** n check both that the local (or generated) name is fresh, but they are handled differently: whereas the construction A **generates** n increases the knowledge of A, the name k of **private** k is not added to any knowledge; this task is deferred to the explicit A **knows** k clauses for the intended A.

The compilation of $A \rightarrow B : M$ checks that M can be synthesized by A, instantiate a new variable x and adds the pair (M, x) to the knowledge of B.

The consistency formula $\Phi(\mathcal{A}(K'_B))$ of the analysis of the updated knowledge K'_B defines the checks ϕ to be performed by B at run-time.

The compilation process is formalized in [25]. The same paper contains various extended examples to illustrate the concepts behind this translation.

3.2 Executable Narrations Optimization

The code generated in the previous steps describes precisely the actions and the checks to be performed by every agent. While the *executable narration* is fine for defining the semantics of the protocols, it is far from being efficient when applied in practice. We address here some issues that can improve the performance of the protocol execution or can make the generated code, in the target language, more compact and readable.

Additionally, in the final output that we call *optimized executable narration* (*Opt-Execnarr*), we include, along with the mapping of the actions, the declaration section available for the *SpyerPN* format. This is useful because the declarations model the knowledge of the agents, as such information will be analyzed generating of the code. It worth noting that in the declarations only names or ground expressions are present. The syntax of the optimized executable narration is shown in Table 3.7.

Since the meaning of these optimizations is rather intuitive, we present them informally with the support of an example. Let's consider the first *AnBx* step of the revised 2KP protocol. Agent `C` sends to agent `M` a message composed by two verifiable digests:

```
C->M,(-|-|M):[can(C):A],[Desc:M]
```

The statement is translated to *AnB* as follows:

```
C->M:{{hmac(can(C),HA},{HA}pk(A),hmac(Desc,HM),{HM}pk(M)}pk(M)
```

which is further translated to executable narration, in an emit and a receive action

```
C: M!enc(<hmac(can(C),ha),<enc(ha,pub(pkA)),
          <hmac(desc,hm),enc(hm,pub(pkM))>>>,pub(pkM)
M ?0
```

where 0 is the variable representing the message received by M. On reception the agent M has to perform several checks. One of the equality checks is
$\pi_1(\pi_2(\pi_2(dec(0, priv(pkM))))) = hmac(desc, dec(\pi_2(\pi_2(\pi_2(dec(0, priv(pkM))))), priv(pkM)))$

expressions			
E, F	::=	$[...]$	*same as Execnarr*
	|	$(E_1, ..., E_n)$	*tuple*
	|	$\pi_i(E)$	$i-th\ projection$
$formulae$			
ϕ, ψ	::=	$[...]$	*same as Execnarr*
simple actions			
I	::=	A : **new** k	$fresh\ name\ generation$
	|	$A : send(B, E)$	*message emission*
	|	$A : receive(x)$	*message reception*
	|	$A : x := E$	*assignment*
narrations			
L	::=	ϵ	*empty narration*
	|	$T; L$	*non empty narration*
declarations			
D	::=	A **knows** M	*initial knowledge* (M *is a ground expression*)
	|	A **generates** n	$fresh\ name\ generation$
	|	**private** k	*private name*
protocol narrations			
P	::=	$D; P$	$sequence\ of\ declarations$
	|	L	*narration*

Table 3.7: Syntax of the optimized executable narrations

Pair ⇒ Tuples As we have seen (3.1.3), *SpyerPN* allows composing messages using pairs but not tuples. This is a design choice of the authors of [25] and we maintained the compatibility with their tool. One good practical reason is the possibility to convert the protocol to Spi Calculus (see the *intermediate output formats* paragraph below). However, if we look at the above example we understand at first sight that the presence of many nested projections makes the code is not well readable. For this reason we transformed the pairs in tuples. The projection operator $\pi_i[\cdot]$ is commonly available in many programming languages, for example the retrieval of the *i-th* element of an array. The above check becomes:

$$\pi_3[dec(0, priv(pkM))] = hmac(desc, dec(\pi_4[dec(0, priv(pkM))], priv(pkM)))$$

Variable Generation and Reordering From the previous example we observe that in the equality checks, the decryption of the variable 0 – $dec(0, priv(pkM))$ – is performed twice. In general, during the protocol execution, operations like encryption, decryption, hashing can be repeated on the same data several times.

This phenomenon affects both actions and checks. For example, in sequence of actions, it may be required to resend pieces of data received earlier. They may be the result of some complex expressions involving nested encryption and decryption operations. Similarly in an equality checks we have to take into account all the different ways to compute the values that are compared. This implies that on the two sides of different equality checks, the same

expression or common sub-expression can appear repeatedly.

Since it is well known [41, 57, 74] that cryptographic operations are computationally expensive, it is convenient to store the results of common sub-expressions in variables that can be later retrieved when they need to be employed in the remaining part of the protocol execution. We make here the reasonable assumption that reading from and writing to the RAM is less expensive in time and space than performing additional cryptographic operations on the same data.

Thus, the above example can be amended

```
C: VAR_C_MCANCH2A:=hmac(can(C),ha)
C: VAR_C_MDESCH2M:=hmac(desc,hm)
C: M!enc(<VAR_C_MCANCH2A,enc(ha,pub(pkA)),
          VAR_C_MDESCH2M,enc(hm,pub(pkM))>,pub(pkM))
M: ?VAR_M_0
M: VAR_M_DJI4DM0VPMVPM:=dec(proj[4][dec(VAR_M_0,priv(pkM))],priv(
   pkM))
```

Variables, with prefix `VAR_X_` where `X` is the agent name, are created and expressions are assigned to them. In general, in order to decide if an expression is worth to be stored in a variable it is necessary to check all the following statements of the protocol.

Our tool performs this in two passes. During the first pass (*discovery phase*) the statements of the protocol are analyzed and the candidate expressions are found. Candidate expressions must include at least one of the following CPU intensive operations: encryption, decryption, hashing, hmac, functions and operations on keys like those performed during key agreements. It is clear that at this level of abstraction, is it impossible to assess the computational complexity of symbolic non cryptographic functions, but we preferred to include them, considering their presence in the abstract model, as an index of their relevance.

If, during the protocol execution, the candidate expression is computed by an agent more than once, a new variable is created and the expression is assigned to it. The expression is then replaced by the variable in all its occurrences. The mapping of variables and expression is stored.

In the second and final pass (*reordering phase*), the variable assignments are analyzed and reordered to further decrease the number of cryptographic operations avoiding recomputing more than once the same values. This pass is needed because during the first pass the statements are analyzed in the sequential order, therefore a second pass may discover dependencies among variables, and thus anticipate the assignment of variables which are sub-expression of other variables. For example, this sequence of statements

```
C: VAR_C_1 := dec(proj[i:4][dec(VAR_C_0,priv(pkC))],priv(pkC))
C: VAR_C_2 := dec(VAR_C_0,priv(pkC))
```

can be reordered as follows, saving one decryption operation

```
C: VAR_C_2 := dec(VAR_C_0,priv(pkC))
C: VAR_C_1 := dec(proj[i:4][VAR_C_2],priv(pkC))
```

Deferred Generation of Fresh Values In an executable narration, the actions associated with the generation of fresh values, if present, are placed by `spyer` at the beginning of the protocol, regardless the actual step in which they are used. It is hence useful to rearrange the order of actions to defer the creation of fresh values until it is really necessary. In practice

Protocol	*Avg exec time (ms)*	*Exec time reduction*
1KP (opt)	1681	-39%
1KP	2768	
2KP (opt)	2320	-29%
2KP	3257	
3KP (opt)	2775	-36%
3KP	4076	

Table 3.8: Experimental results of the optimization

the effect is to place each **new** action just above the action employing for the first time the fresh value. This could reduce to load of the processor, for example when a protocol session is aborted, due to a check failure.

Experimental Results In order to verify the benefit of the optimization we generated the Java code of the three variants of *iKP* (*1KP,2KP* and *3KP*). For each protocol we considered two different settings: one with the optimization and the other without. In both cases, pairs were already converted to tuples, so this aspect is not taken into account in the experimental results. Within the Eclipse 3.7 IDE, we run five times each protocol on a computer with these features: CPU AMD Athlon 64 X2 (dual-core) 3.0 GHz, RAM 4 GB, operating system Windows 7 64-bit SP1. The three instances (one for each role) were running in parallel on the same machine. The experimental data (Table 3.8) show that the optimization pays off reducing the execution time between 30% and 40% for the three versions of *iKP*. The translation from *AnBx* to *AnB* we used here was the "lightest", doing the minimum number of exchanges and cryptographic operations, with respect to more realistic setting (for example, those using challenge-response to achieve the freshness). We think that in these cases the benefit of the optimization will be even greater.

Intermediate Output Formats In this section we have illustrated the steps to translate an *AnBx* / *AnB* protocol to an optimized executable narration. As we have seen this process comprises these several phases, as showed in Figure 3.1, and summarized here:

$AnBx \rightarrow AnB \rightarrow SpyerPN \rightarrow Execnarr \rightarrow Opt\text{-}Execnarr$

Our tool provides an output for each of these intermediate formats. This can be useful not only for making easier the debugging, but also for allowing interoperability with other tools including `spyer` itself. Moreover, as a by-product inherited from the `spyer` tool, we can also output to a sub-calculus of the Spi Calculus [3], that can then be used with the *Symbolic Bisimulator Checker* (SBC) [22].

4

Automatic Java Code Generation of Security Protocols

4.1 From Executable Narrations to Java Code

As we have seen in the previous chapter we began from a specification of a security protocol in *AnBx* and through a series of translations we obtained a more detailed, but still abstract, representation of the protocol, called *optimized executable narration*. This intermediate protocol scheme, that formalizes and refines the original specification, is an important step because it makes explicit the actions and the checks upon reception that each agent has to perform during the protocol execution. The way such actions and checks are written is suitable to be translated in an imperative programming language because they can be expressed by means of constructs that are generally available in such languages: variable assignments, functions and procedure calls, logical conditions such as equalities.

It should be clear that before to automatically generate the source code of an application, we need to fill several implementation details that are not caught by the formal model. If we consider the *optimized executable narration* as a complete, language independent, abstract representation of the protocol, we can devise a design approach that keeps apart the *protocol logic* from the *application logic*.

Informally we can say that, given a target programming language, the application logic is the part of code which is added to the protocol logic to generate the concrete and complete application. Separating the two logics (Figure 3.2) has the advantage to allow the potential generation of source code in different programming languages. Although here we are focusing on Java we think that, applying the same strategy, generating code in other object-oriented or procedural typed languages might be done with a limited effort. The main idea is to plug the protocol logic in the application skeleton/template which is independent from the protocol itself.

To implement such approach, from the practical point of view, we need at least the following ingredients:

- a parametrized application template – the *application logic*
- the information derived from the protocol – the *protocol logic* – employed in the preparation of the code generation. We denote such intermediate format as *JProtocol*.
- a software library accessible through an application programming interface

Parametrized Application Template - Application Logic This component, written in the target language, is the skeleton of the application. The functions implemented by this template can include, for example, the parsing of command line arguments, the initialization of the agents, the configuration of the communication channels and the software components offering access to the cryptographic resources. These functionalities are typically used by every security application regardless the concrete protocol and hence they are suitable to be included in the application template. Clearly the concrete code must be parametrized according to the protocol specification, hence the protocol steps and relative actions will depend on the specific protocol. Additionally we can consider the fact that every agent must open at least a communication channel, but the exact number of channels depends on the protocol logic.

***JProtocol* - Protocol Logic** From the optimized protocol narration we must derive the information that will be used to instantiate the application template, and hence a translation process must be defined. We found useful to introduce here another intermediate step because it makes simpler to extract from the protocol narration the information, i.e. the parameters, that will be used to instantiate the application template during the code generation. This format is still independent from the target language, and this comes out to be handy for the generation of the code in the target language.

The information extracted includes the parameters needed to instantiate each agent: the channels used, the local variables and the local procedures. It is worth noting that almost all the information can be directly derived from the optimized protocol narration, but, in a few cases, it might be necessary, or just more convenient, to retrieve the information from the *AnB* specification.

Since it reasonable to assume that the target language must be a typed language, we need to infer the type of variables and expressions. In general the *AnB* types are too vague to be used in a effective way in a real programming language. Therefore, in addition to the type system (4.1.2), we will need some convention on the variable names.

The intermediate protocol format, that we call *JProtocol*, it is a 7-tuple including the following elements:

$$(name, roles, steps, channels, fields, methods, actions)$$

1. The *protocol name.* It is used thoroughly in the code generation. First of all it gives the name to the application. It is retrieved from the *AnB* specification, since the *SpyerPN* and the executable narration do not carry this data.

2. The *roles* (agents) running the application. The role names are used by the agents to refer to their peers in the narration. They will also be bound to the concrete agent names/aliases by means of a mapping defined in the *application configuration file* (see 4.1.3). Since the set of roles is built analyzing the actions, agent names included in the agent knowledge, but not involved in the protocol, are ignored.

3. The protocol *steps*. To improve the readability of the generated code, instead of writing the actions performed by each agent as sequence of statements, it is more convenient to group corresponding consecutive instructions according to the protocol steps in which they are executed. Each step includes only one send or one receive action, and zero or more actions of type assignment or check. The idea is to have a main procedure calling to

a subroutine parametrized on the protocol step. This allows generating more meaningful logs and make easier the debugging the code. The steps are retrieved analyzing the protocol actions.

4. The communication *channels* employed during the protocol execution. Each pair of communicating agents needs a channel. It will be, in a standard network environment, a TCP/IP socket where one agent will act as client and the other as server. As a design choice, the one who starts the communication will be the client, the other the server. If an agent is exchanging messages with more than one agent, he will open a different channel with each peer. Again, the channel parameters are inferred from the protocol actions.

5. For each role, the local variables, procedures and functions. In an object oriented terminology, the class *fields* and *methods* that are declared by each *role* class. These element will be used in statements that will implement the protocol actions in the target language.

6. The *actions* performed by each agent during the protocol run. There are two issues to consider in the translation from the optimized executable narrations to the intermediate format: the translation of *actions*, and the translation of *expressions* used in such actions.

How local methods and field are retrieved and how actions are translated is explained in the next section.

Software Library and its Application Programming Interface Instead of fully generating the source code of every function used by the application, it is convenient to isolate a set of software components which can be reused among different applications. This approach is feasible because the class of programs we are considering performs actions that can share a common set of standard functions. Think about sending and receiving messages, encrypting and decrypting data, and so on. Therefore it is advantageous to group and standardize these common functions and data structures, in what is usually called a *software library*. The access to the resources of the library is, as customary, defined by means of an application programming interface (API) defining how the software components can communicate among them. This design strategy has several clear advantages:

- it relieves the application from dealing with a lot of implementation details;
- it makes considerably simpler the specification of the application templates;
- it makes more modular and cleaner the shape to the application;
- the maintenance of the code is simpler and more efficient: every program can benefit from the improvements and bug fixes in the library;
- it gives the chance to use alternative implementations of the library, provided the same standard API.

4.1.1 From Optimized Executable Narration to *JProtocol*

As we have seen, the intermediate representation of the protocol is a 7-tuple:

$$(name, roles, steps, channels, fields, methods, actions)$$

In the previous section we shortly explained how the first four components are retrieved from the optimized executable narration. The procedure is simple and does not deserve any further explanation. Instead the last components require some care:

Local Procedures/Methods The set of local (private to each role) *methods* is built analyzing the *declarations* of the optimized protocol narration. If a function is part of the agent knowledge, excluding functions belonging to the library, such function must be available to the agent. In practice, in an object oriented language, a private method must be implemented in the role class. However, since in the protocol narrations the functions are abstract, only the skeleton of the method can be generated and a dummy value is returned. In this way, the generated code can be compiled and the application will be runnable, but the duty of filling the skeleton with a concrete implementation of the function is left to the user. This is the only "unfinished" portion of code produced by the code generator.

Local Variables/Fields The definition of the local variables, or class *fields*, is obtained by two components:

1. The first one derives from the declarations, and includes, for each agent, the known and the generated names. For each name a variable is declared, excluding the role names and the function names. For example, given this portion of the knowledge of agent `A`:

   ```
   M knows hash
   M knows hmac
   M knows C
   M knows price
   M generates tid
   ```

   the translation will include the declaration of two variables `price` and `tid`. The remaining names are ignored because they are either role names (`C`) or functions included in the library (`hash` and `hmac`). Along with the variable identifier, the type, as inferred by the type system (see 4.1.2), is stored.

2. The second component is built gathering information from the action list. For each assignment or reception action a variable will be declared. For example, these statements

   ```
   M: ?VAR_M_0
   M: VAR_M_1 := dec(VAR_M_0,priv(pkM))
   ```

   will imply two variables declarations: `VAR_M_0` and `VAR_M_1`. They will, respectively, store the value received by agent `M` and the result on an expression, namely the computation of the decryption of the first variable with the private key of agent `M`. Again, the type of the variable is evaluated by the type system.

E, F	::=	$encrypt(E, F)$	*encryption*
	|	$decrypt(E, F)$	*decryption*
	|	$sign(E, F)$	*signature*
	|	$verify(E, F)$	*verify*
	|	$hmac(E, F)$	*hmac*
	|	$hash(E)$	*hash*
	|	$DHPubKey(E, F)$	*DH public key*
	|	$DHSecKey(E, F)$	*DH secret key*
	|	$(E_1, ..E_n)$	*tuple*
	|	$\pi_i(E)$	*projection*
	|	$f(E)$	*function*
	|	$var(x, E)$	*variable binding*
	|	id	*identifier*

Table 4.1: Syntax of *jexpressions*

Mapping of Expressions The last component of *JProtocol*, the mapping of the actions, requires the translation of *expressions* (Table 3.7) to a format that we call *jexpressions* (Table 4.1). In many cases the translation is straightforward, but when the expression includes functions like encryption and decryption some care is needed. The operators *enc* and *dec* are commonly used to model the symmetric and asymmetric encryption. Moreover in the latter, we distinguish, as it happens in practice, between the digital signature operation and the standard encryption applied to achieve secrecy. The discrimination is done by means of the *public keys conventions* (see 3.1.2): key identifier = prefix *sk* + agent name, for the signature and key identifier = prefix *pk* + agent name, for the encryption. Key patterns not falling in these two classes are treated as a symmetric encryption. The type checker is in charge of verifying if the key type is coherent with the structure of the expression.

Here some examples showing how expressions are translated to *jexpressions*:

```
dec(X,priv(pkM)) => dec(X,priv(pk(M))
dec(X,pub(skM)) => verify(X,pub(sk(M))
enc(X,pub(pkM)) => enc(X,pub(sk(M))
enc(X,priv(skM)) => sign(X,priv(sk(M))
dec(X,K) => dec(X,K)
enc(X,K) => enc(X,K)
```

The translation of identifiers (names) and agent names maps each identifier to a pair *(identifier, type)*. Since *SpyerPN* and the subsequent narrations are untyped, we rely on the declarations statements included in the *AnBx* specification. No ambiguity arises for types `Agent`, `Certified`, and `SeqNumber`. Identifiers of type `PublicKey` are conventionally mapped to public keys for encryption. Instead types like `Number` and `Symmetric_key` are too generic to be employed directly. For example, nonces and key agreement parameters including half-keys are all declared in *AnBx* as `Number` but in the target language they have, in general, a different type.

To overcome any possible ambiguity, the type is inferred by means of a naming convention (see Table 4.2), provided that these identifiers where declared as `Number` or `Symmetric_key`. The same name convention is used to denote identifier added in the compilation from *AnBx* to *AnB*, hence the original type is preserved.

prefix	*type*
`Nx`	nonce
`Kx`	symmetric key
`Xx,Yx`	Diffie-Hellman parameters and half keys
`Hx`	hmac keys
`SQNx`	sequence number

Table 4.2: Naming convention for identifiers of type `Number` and `Symmetric_key`

$$
\begin{array}{rcll}
I & ::= & A, i : \textbf{new}\ k : T & fresh\ name\ generation \\
 & | & A, i : send(B, ch, E) & message\ emission \\
 & | & A, i : receive(B, ch, x : T) & message\ reception \\
 & | & A, : x : T := E & assignment
\end{array}
$$

where A,B agents, x:T variable of type T, E expression, ch channel, i step

Table 4.3: Syntax of *jaction*

Identifiers are also changed restoring their original case as declared in *AnBx*. Recall that *SpyerPN* introduces some convention on identifiers and agent names (see 3.1.2).

Mapping of Actions Having translated the expressions is now possible to translate the actions to the format we call *jaction* (Table 4.3). First of all, every action is labeled with the protocol step. As we mentioned earlier each step includes one send or one receive action, and zero or more actions of type assignment or check. Therefore the protocol step number is not unique but it is used to group consecutive actions in the generated code. Moreover emission and reception actions are enriched with the reference to the communication channel used by them. This is necessary because in the concrete code the communications actions must be bound to a specific communication channel. The *steps* and the *channels* lists are available from the previous steps. Finally, the type of the variables created in actions – reception, assignment and fresh name generation – is inferred by the type system and stored along with the variable identifier.

4.1.2 The Type System

AnB is a typed language, but its set of types it is too poor compared with the needs typical in coding a security protocol. Although the variety of types depends on the specific language, in general any real typed programming languages has a range of types richer an abstract language like *AnB*. Although in some cases, it could be possible to have a minimal set of types in the target language, this does not bring any benefit. On the contrary, it is well known that types help to code better programs [70].

Considering the application field of *AnB*, the verification of security protocols, types like `Number` and `Symmetric_key` are sufficient to represent the terms used in protocols. However terms belonging to the same type in *AnB*, as we have seen, might be translated differently in target language. For instance, while `Number` is used in *AnB* to model both nonces and key

agreement half-keys, in Java nonces may be instances of the class `ByteArray`, while public half keys may be instances of the class `PublicKey`.

The naming convention we devised (Table 4.2) helps to discriminate identifiers of different type to some extent, but in general it is not sufficient for handling complex terms. Let's look at the encryption and decryption operations. Agents should be able to encrypt terms of any type[1]; in turn, decryption should output a term of the original type, the type of the data before the encryption. For example, in Java we could have a cryptographic engine exposing the following methods for symmetric encryption and decryption:

```
public SealedObject encrypt(Object obj, Key symmetricKey) {...}
public Object decrypt(SealedObject so, Key symmetricKey) {...}
```

The first methods encrypts any `Object` with an appropriate `Key` and outputs an encrypted object of type `SealedObject`. A snippet of code using such methods is:

```
private SealedObject SO = null;
private String Msg = new String ("msg");
private Key myKey = ... // the key is retrieved from the key store
SO = encrypt(Msg, myKey);
ch.send(SO); // the data is send to the network
```

It should be noted that the call to `encrypt()` does not require an explicit cast of `Msg`. In fact `String`, as any other type in Java, is a subtype of `Object`. On the contrary, the `decrypt()` method outputs a value of type `Object`. Therefore in order to assign that value to a variable of type `String` an explicit cast is required. Otherwise the program will not compile.

```
private SealedObject SO = null;
SO = ch.receive(); // a SealedObject is received from the network
private String Msg;
private Key myKey = ... // the key is retrieved from the key store
Msg = (String) decrypt(SO, myKey);
```

However this does not guarantee the absence from run-time errors. For example, in the above code, we could have a run-time error (and the raise of cast exception) if the decrypted object is not of the expected type (`String`) or a subtype of it.

What we need, in general, is the ability to infer the type of terms. This is used to declare new variables or to compute the appropriate cast in assignments or within expressions. Such task can be accomplished by the type system and its type inference algorithm. As a first step we must to declare the types we want to include in the type system. Types in Table 4.4 are those typically used in a wide range of security applications. It is worth noting that these types are still abstract, in the sense that their mapping to concrete types in the target language is postponed until the actual code is generated. We employ here an object oriented terminology because Java is our main target language but this type system could be employed also with other typed procedural languages like C or Pascal.

Some types (*Agent*,*Certified*,*SeqNumber*,*SymmetricKey*) simply map the *AnBx* types. *PublicKey* and *PrivateKey* are parametric types where the parameter is used to distinguish keys with different purpose (signing and encryption). Three more parametric types are used for encrypted (sealed) and signed objects. In this case the parameter is needed to keep track the type of the original object before the cryptographic operation. The parameter is then

[1]Some restriction my apply but it's not relevant at this context

T	::=	$Agent$	*agent*
	|	$Certified$	*certified agent*
	|	$Nonce$	*nonce*
	|	$SeqNumber$	*sequence number*
	|	$SymmeticKey$	*symmetic encryption key*
	|	$PublicKey\,\langle S\rangle$	*public key of type* S
	|	$PrivateKey\,\langle S\rangle$	*private key of type* S
	|	$SealedObject\,\langle T\rangle$	*sealed object of type* T
	|	$SealedPair\,\langle T\rangle$	*sealed object of type* T *+ key*
	|	$SignedObject\,\langle T\rangle$	*signed object of type* T
	|	$Hash$	*hash*
	|	$HmacKey$	*hmac encryption key*
	|	$HmacPair$	*hmac + key*
	|	$DHBase$	*DH parameter spec*
	|	$DHKeyPair$	*DH key pair*
	|	$DHPubKey$	*DH public key*
	|	$DHSecKey$	*DH secret key*
	|	$\{T_i{}^{i\in\{1..n\}}\}$	*tuple*
	|	$T \to T$	*function*
	|	$String$	*base type*
	|	$Object$	*base type*
S	::=	PK	*key pair type for encryption*
	|	SK	*key pair type for signing*

Table 4.4: Type system - Types

employed when the inverse operation is performed. In details, *SealedObject* is an encrypted object, *SealedPair* is used when a key is encrypted along with data as in the hybrid cryptography, *SignedObject* is a digitally signed object. *Hash* and *HmacPair* are used to model digests supported by *AnBx*. *HmacPair* has the option of storing an optional key for verifiable digests. Keys used to compute the HMACs should have type *HmacKey*. A set of DH-types is used for values employed in key agreements. Tuples and functions are standard features of the language and they have their type counterparts here. Finally *Object* and *String* are the base types. *Object* can be thought as the default type, while *String* is a type commonly available in many programming languages, and it is useful in the generated code to produce human readable output.

The type inference algorithm is based on the typing rules shown in Table 4.5.

The T-HASH and T-HMAC rules model the creation of digests. The original type is obfuscated and cannot be retrieved since hashing and macs are not invertible. T-FUN models functions, T-CAT the concatenation and T-PROJ the projection. T-ENC-* rules model encryption. It should be noted that we impose some constraints on the type of the key, depending on whether asymmetric or symmetric encryption is used. The resulting type is parametrized on

$$\text{T-HASH}\ \frac{t : T}{hash(t) : Hash}$$

$$\text{T-HMAC}\ \frac{t_1 : T_1 \quad t_2 : HmacKey}{hmac(t_1, t_2) : HmacPair}$$

$$\text{T-FUN}\ \frac{f : T_1 \to T_2 \quad t : T_1}{f(t) : T_2}$$

$$\text{T-CAT}\ \frac{for\,each\,i \quad t_i : T_i \quad i \in \{1..n\}}{\left\{t_i\ ^{i \in \{1..n\}}\right\} : \left\{T_i\ ^{i \in \{1..n\}}\right\}}$$

$$\text{T-PROJ}\ \frac{t_1 : \left\{T_i\ ^{i \in \{1..n\}}\right\}}{t_{1 \cdot j} : T_j}$$

$$\text{T-ENC-ASYM}\ \frac{t_1 : T_1 \quad t2 : PublicKey\,\langle PK \rangle}{enc(t_1, t_2) : SealedPair\,\langle T_1 \rangle}$$

$$\text{T-ENC-SYM}\ \frac{t_1 : T_1 \quad t2 : T_2}{enc(t_1, t_2) : SealedObject\,\langle T_1 \rangle} \qquad T_2 \in \{SymmetricKey, DHSecKey\}$$

$$\text{T-DEC-ASYM}\ \frac{t_1 : SealedPair\,\langle T_1 \rangle \quad t2 : PrivateKey\,\langle PK \rangle}{dec(t_1, t_2) : T_1}$$

$$\text{T-DEC-SYM}\ \frac{t_1 : SealedObject\,\langle T_1 \rangle \quad t2 : T_2}{dec(t_1, t_2) : T_1} \qquad T_2 \in \{SymmetricKey, DHSecKey\}$$

$$\text{T-SIGN}\ \frac{t_1 : T_1 \quad t2 : PrivateKey\,\langle SK \rangle}{sign(t_1, t_2) : SignedObject\,\langle T_1 \rangle}$$

$$\text{T-VERIFY}\ \frac{t_1 : SignedObject\,\langle T_1 \rangle \quad t2 : PublicKey\,\langle SK \rangle}{verify(t_1, t_2) : T_1}$$

$$\text{T-KAP}\ \frac{t_1 : DHBase \quad t2 : DHKeyPair}{kap(t_1, t_2) : DHPubKey}$$

$$\text{T-KAS-1}\ \frac{t_1 : DHPubKey \quad t2 : DHKeyPair}{kas(t_1, t_2) : DHSecKey}$$

$$\text{T-KAS-2}\ \frac{t_1 : DHKeyPair \quad t2 : DHPubKey}{kas(t_1, t_2) : DHSecKey}$$

Table 4.5: Type system - Typing rules

T_1, the type of the original object before encryption. Symmetrically the T-DEC-* rules model the decryption. Here the parameter is used to determine the type of the object after the decryption. Similarly, some checks are done on the type of the keys employed for decryption.

T-SIGN and T-VERIFY behave like T-ENC-* and T-DEC-*, but they are used to model the digital signature and its verification. The last three rules T-KAP, T-KAS-1 and T-KAS-2 are used to set the typing rules of the operations performed during the key agreements. Two T-KAS-* rules are provided because they model two ways the agents have to compute the shared secret keys, at the end of the key agreement protocol.

For the practical implementation of the type system we found useful to take inspiration from the examples included in "*Types and Programming Languages*" (TAPL) by Benjamin C. Pierce [70]. In particular we adapted some portion of the Haskell port of the original OCaml implementations. The port is done by Ryan W. Porter, and it is available at `http://code.google.com/p/tapl-haskell/`

4.1.3 Application Template

The last component of our toolbox is the application template. This set of files represents the skeleton of the Java application, and we refer to it as the *application logic*. The template is filled with the *protocol logic*, the data synthesized from the specification of the executable narration, and stored in the *JProtocol* data structure. The template files must be written in the target language, Java in our case.

As a running example, we show portions of code taken from the revised *2KP* (protocol name: `Rev_2KP`). Additionally, to help orienting the reader, the UML class diagram of the resulting application is shown in Figure 4.1.

In general, assuming that the protocol name is `ProtName`, the full application is composed by the following Java file (classes):

`ProtName.java`

The main file of the application defines the `ProtName` class, which implements only the method `main()`. This method is invoked when the application is started: the level of debugging messages is set and the command line parameters are passed, by means of the `Parse()` method, to the final class `ProtName_CommandLine_Parser`. The command line parameters `args` must include, along with other settings, the agent role that this instance of `ProtName` is playing during the protocol execution. For each agent an instance of `ProtName` must be created.

```
public class Rev_2KP {
 public static void main(String[] args) {
  AnBx_Debug.setAPPLICATION(true);
  AnBx_Debug.setPROTOCOL(true);
  Rev_2KP_CommandLine_Parser.Parse(args,Rev_2KP.class.toString());
  }
}
```

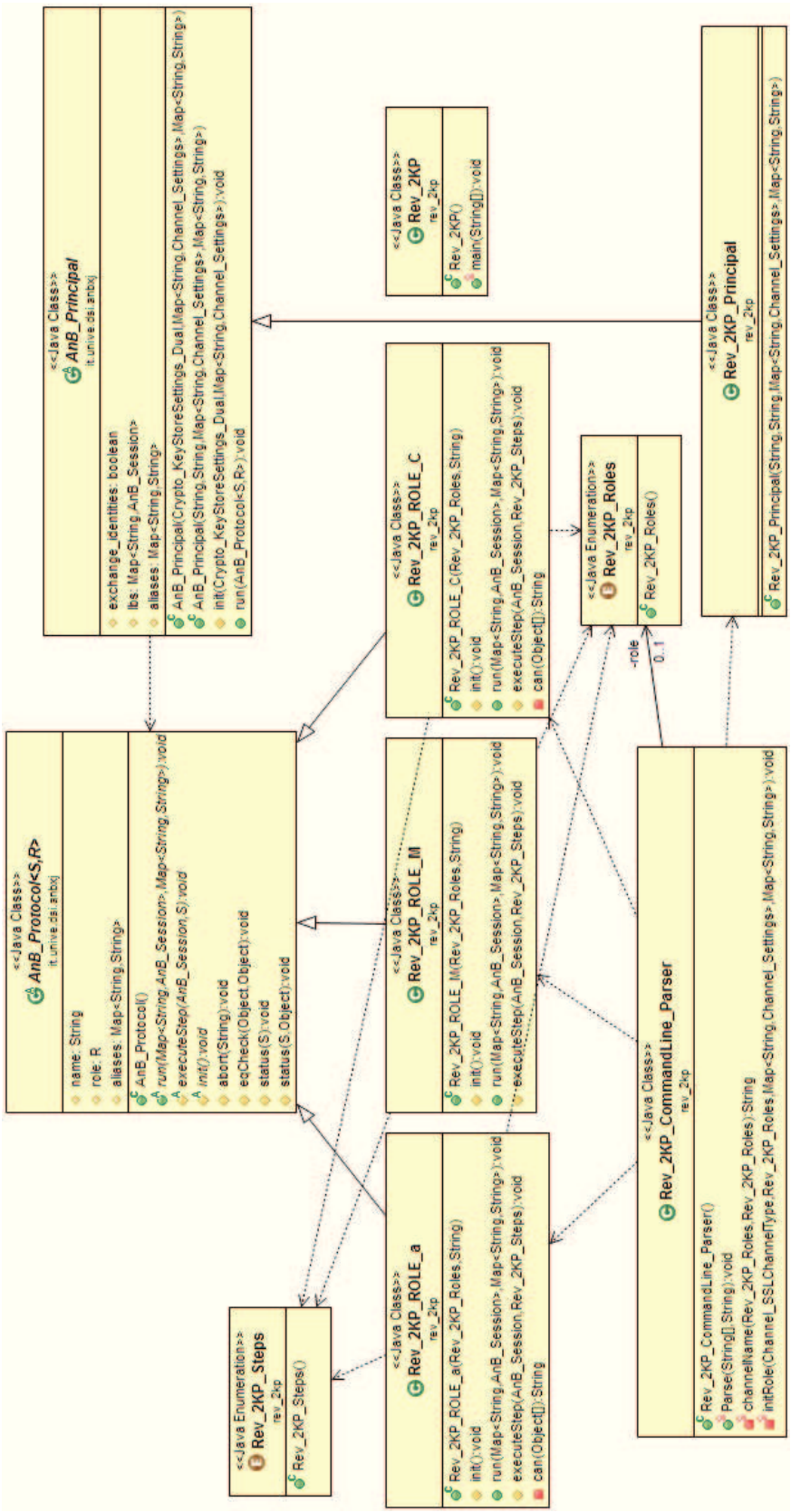

Figure 4.1: UML class diagram of revised $2KP$ in Java

`ProtName_CommandLine_Parser.java`

The purpose of the `ProtName_CommandLine_Parser` class is to initialize the application. The invocation of method `Parse()` causes the processing of the configuration file (details about its content are given below). The command line arguments are used to set up the parameters of the application. In detail, the main actions performed by this class are:

1. the mapping between the roles in the protocol and the identities of principals playing those roles. This is done by the `initRole` method

   ```
   Channel_SSLChannelType ct = Channel_SSLChannelType.SSL_PLAIN;
   Map<String, Channel_Settings> cs = new HashMap <String,
       Channel_Settings>();
   Map<String, String> aliases = new HashMap <String,String>();
   Rev_2KP_Principal Rev_2KP_pr = null;
   initRole(ct, role, cs, aliases);
   ```

   Here `ct` is SSL/TLS channel type we are running the application on top of it . The default value is the plain channel.

2. reading the configuration file

   ```
   private static void initRole(Channel_SSLChannelType ct,
       Revised_2KP_Roles role, Map<String, Channel_Settings> cs, Map<
       String, String> aliases) {
   [...]
   Properties configFile = new Properties();
   try {
     AnBx_Debug.out(layer, "Reading config file: " + configFileName.
         toString());
     configFile.load(Rev_2KP_CommandLine_Parser.class.
         getResourceAsStream(configFileName));
    } catch (IOException e1) {
     AnBx_Debug.out(layer, "Error reading config file: " +
         configFileName.toString());
     e1.printStackTrace();
    }
   }
   ```

3. reading, from the configuration file, the key path of the keystore, where keys and certificates can be retrieved. Additionally the alias of the agent is set; this is necessary for self identification, for example for the retrieval of the private keys

   ```
   private static void initRole(Channel_SSLChannelType ct,
       Revised_2KP_Roles role, Map<String, Channel_Settings> cs, Map<
       String, String> aliases) {
   [...]
     myAlias = configFile.getProperty(role.toString());
     keypath = configFile.getProperty("keypath");
   [...]
   }
   ```

4. the creation and the initialization of the communication channels. Parameters like the hostname, the port, and the role played in the channel (client or server) are retrieved.

```
private static void initRole(Channel_SSLChannelType ct,
   Revised_2KP_Roles role, Map<String, Channel_Settings> cs, Map<
   String, String> aliases) {
[...]
for (Rev_2KP_Roles peer : Rev_2KP_Roles.values()) {
 if (configFile.getProperty(peer.toString()) != null) {
    aliases.put(peer.toString(), configFile.getProperty(peer.
       toString()));
  }
  if (!peer.equals(role)) {
    ch = channelName(role,peer);
    String host = configFile.getProperty(ch + "_host");
    if (host != null) {
      Integer port = Integer.parseInt(configFile.getProperty(ch +
          "_port"));
      if (configFile.getProperty(ch + "_role").equalsIgnoreCase("
         Client")) {
        cs.put(peer.toString(),new Channel_Settings(ct,
           Channel_Roles.CLIENT, host, port));
      } else {
        cs.put(peer.toString(),new Channel_Settings(ct,
           Channel_Roles.SERVER, host, port));
      }
    }
  }
}
```

5. the creation of an object of class `ProtName_Principal`, according to the role the agent is playing, and the invocation of its method `run()`.

```
Rev_2KP_Principal Rev_2KP_pr = null;
Properties configFile = new Properties();
initRole(ct, role, cs, configFile, aliases);
if (myAlias != null && keypath != null) {
 Rev_2KP_pr = new Rev_2KP_Principal(myAlias, keypath, cs, aliases
    );
   switch (role) {
     case ROLE_C:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_C(role, protname));
          break;
     case ROLE_M:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_M(role, protname));
          break;
     case ROLE_a:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_a(role, protname));
          break;
        }
        } else {
               AnBx_Debug.out(layer, "Unable to initialize
                  Rev_2KP Principal");
               }
        }
```

`ProtName_Principal.java`

This class extends the library class `AnB_Principal` and holds the agent knowledge about channels and cryptographic material (keys, certificates, aliases). Once the principal is initialized a call to the superclass method `run()` executes in sequence the steps the protocol, that the agent has to perform .

```
class Rev_2KP_Principal extends AnB_Principal {
  public Rev_2KP_Principal(String myAlias, String path, Map<String,
     Channel_Settings> cs, Map<String, String> aliases) {
    super(myAlias, path, cs, aliases);
  }
}
```

`ProtName_Role_<X>.java`

This class extending the library class `AnB_Protocol` (Figure 4.7) is the core of the application. A different file, where X is replaced with the role name, is generated for each role. The method `run()` initialize the communication channels and executes the steps of the protocol. The two parameters of `run()` are the mapping of the communication channels and the mapping of role/aliases. Here we are showing portions of code of the agent playing the Merchant role (`ROLE_M`) who directly communicate with two other agents, the customer (`ROLE_C`) and the acquirer (`ROLE_a`), and therefore he needs to open two channels.

```
public void run(Map<String, AnB_Session> lbs, Map<String, String>
   aliases) {
  this.aliases = aliases;
  AnB_Session ROLE_M_channel_ROLE_C = lbs.get("ROLE_C");
  AnB_Session ROLE_M_channel_ROLE_a = lbs.get("ROLE_a");
    try {
          init();
      ROLE_M_channel_ROLE_C.Open();
      ROLE_M_channel_ROLE_a.Open();
      do {
        executeStep(ROLE_M_channel_ROLE_C, Revised_2KP_Steps.STEP_0);
        [...]
        executeStep(ROLE_M_channel_ROLE_C, Revised_2KP_Steps.STEP_7);
         } while (loop);
      ROLE_M_channel_ROLE_C.Close();
      ROLE_M_channel_ROLE_a.Close();
        } catch (Exception e) {
        [...]
                }
}
```

The method `executeStep()` executes the actions specified by each steps (sending, receiving, variable assignments, checks on reception). The two parameters are the session `s` and the protocol `step` to be executed. The `AnB_Session` (Figure 4.6) is a library class giving access both to the communication channel and to the cryptographic functions. Several communication and cryptographic operations are performed in these two protocol steps.

```
private void executeStep(AnB_Session s, Rev_2KP_Steps step) {
  status(step);
```

```
  switch (step) {
  case STEP_0:
  // C -> M,(-|-|M): [can(C):a],[Desc:M]
  // C -> M: {hmac(can(C),H3a),{H3a}pk(a),hmac(Desc,H3M),{H3M}pk(M)}pk
     (M)
    VAR_M_0 = (Crypto_SealedPair) s.Receive();
    VAR_M_DM0VPM = (AnBx_Params) s.decrypt(VAR_M_0);
    VAR_M_DJI4DM0VPMVPM = (SecretKey) s.decrypt((Crypto_SealedPair)
       VAR_M_DM0VPM.getValue(3));
    VAR_M_MDESCDJI4DM0VPMVPM = s.makeHmac(Desc,VAR_M_DJI4DM0VPMVPM);
    eqCheck((Crypto_HmacPair) VAR_M_DM0VPM.getValue(2),
       VAR_M_MDESCDJI4DM0VPMVPM);
    break;
  case STEP_1:
    //  M -> C,(@M|C|-): TID,[Price,TID,[Desc:M],[can(C):a]]
    //  M -> C: {C,SQN4,TID,hash(Price,TID,hmac(Desc,H3M),hmac(can(C),
       H3a))}inv(sk(M))
    TID = s.getSeqNumber();
    SQN4 = s.getSeqNumber();
    s.Send(s.sign(new AnBx_Params(aliases.get("ROLE_C"),SQN4,TID,s.
       makeDigest(new AnBx_Params(Price,TID,VAR_M_MDESCDJI4DM0VPMVPM,(
       Crypto_HmacPair) VAR_M_DM0VPM.getValue(0))))));
    break;
  case STEP_2:
        [...]
   }
  status(step);
}
```

ProtName_Roles.java

This enumeration class contains the list of *roles* (agents) participating in the protocol.

```
public enum Revised_2KP_Roles {
  ROLE_C, ROLE_M, ROLE_a
}
```

ProtName_Steps.java

This enumeration class contains the list of *steps* of the protocols.

```
public enum Rev_2KP_Steps {
  STEP_0, STEP_1, STEP_2, STEP_3, STEP_4, STEP_5, STEP_6, STEP_7
}
```

Configuration File

The last generated file is the configuration file which contains a set parameters that, along with the command line arguments, are used to initialize the application. This file can be easily modified by the end user without need to re-generate the code of the application. An example of configuration file is shown in Figure 4.2. The parameters include the path where the keystore is located in the file system. The keystore contains the keys and certificates of

```
# Protocol: Rev_2KP
# Java Config File: "C:/genAnBx/src/Rev_2KP/Rev_2KP.properties"
# Roles/Aliases
ROLE_C = alice
ROLE_M = bob
ROLE_a = charlie
# Channels
ROLE_C_channel_ROLE_M_role = Client
ROLE_C_channel_ROLE_M_host = 127.0.0.1
ROLE_C_channel_ROLE_M_port = 6666
ROLE_M_channel_ROLE_C_role = Server
ROLE_M_channel_ROLE_C_host = 127.0.0.1
ROLE_M_channel_ROLE_C_port = 6666
ROLE_M_channel_ROLE_a_role = Client
ROLE_M_channel_ROLE_a_host = 127.0.0.1
ROLE_M_channel_ROLE_a_port = 6669
ROLE_a_channel_ROLE_M_role = Server
ROLE_a_channel_ROLE_M_host = 127.0.0.1
ROLE_a_channel_ROLE_M_port = 6669
# Paths
keypath = C:/JavaProjects/demos/src/demos/keygen_dual/
```

Figure 4.2: Protocol.Properties configuration file

known agent. Moreover the configuration file includes the mapping between protocol roles and agent aliases and the setting of the communication channels.

The path of the keystore, is used by the application to retrieve the key material which is used during the execution. It is assumed that the format of the key store is compatible with the cryptographic settings of the application. In general the application is designed to use the appropriate cryptographic algorithms based on the keys type available in the keystore. In this database, keys and certificates are associated to an alias which is used as an index to access to that cryptographic objects. It is hence necessary to provide an explicit mapping between protocol roles and the alias of the agent who is actually playing that specific role.

Channel parameters are used to initialize the TCP/IP sockets and they include the network role (client or server), the port and the hostname. Note that the hostname is used only for the client channels, because servers listen on a port in their own system, and they do not need that parameter to setup the socket. The default value for the hostname is the localhost IP address (127.0.0.1) but it can be changed freely if the user wants to run processes on different machines. In this case each every machine running the protocol must have its own copy of the configuration file. For security reasons it is not advisable to share the same file among different agents. Moreover the client port and server port must match in order to establish a communication.

4.1.4 Code Generation

The last phase in the process is the code generation. As we have seen, we do not only produce the security related code (the protocol logic) but also a complete application combining the information derived from the optimized executable narration, with the application template

abstract API call	*AnBxJ/Java API calls*	*abstract Type*	*AnBxJ/Java Types*
APISend	`Send`	SealedPair	`Crypto_SealedPair`
APIReceive	`Receive`	SealedObject	`SealedObject`
APIEncrypt	`encrypt`	SignedObject	`SignedObject`
APIDecrypt	`decrypt`	HmacPair	`Crypto_HmacPair`
APISign	`sign`	JHash	`Crypto_ByteArray`
APIVerify	`verify`	AnBx_Params	`AnBx_Params`
APIHash	`makeDigest`	JString	`String`
APIHmac	`makeHmac`	JObject	`Object`
APISQN	`getSeqNumber`	JHmacKey	`SecretKey`
APINonce	`getNonce`	JSymmetricKey	`SecretKey`
APISymKey	`getSymmetricKey`	JNonce	`Crypto_ByteArray`
APIHmacKey	`getHmacKey`	JSeqNumber	`Crypto_ByteArray`
APIDHPubKey	`getKeyEx_PublicKey`	JDHBase	`DHParameterSpec`
APIDHKeyPair	`getKeyEx_KeyPair`	JDHPubKey	`PublicKey`
APIDHSecKey	`getKeyEx_SecretKey`	JDHKeyPair	`KeyPair`
APIEqCheck	`eqCheck`	JDHSecKey	`SecretKey`
		JVarArgs	`Object ...`

Table 4.6: API and type bindings

(the application logic).

As a first step we must reconcile the two logics. Concretely this is done binding the abstract view of the types and the API calls with a concrete one. The binding depends on the target programming language and on the support library. In our case the bindings are defined by the maps shown in Table 4.6. The abstract API calls are mapped to the concrete Java calls implemented by our *AnBxJ* library (section 4.2); the abstract Types are mapped to the concrete Java and library types.

Next, we use the *JProtocol* data and the bindings to generate syntactically correct Java statements (or portions of them) to be injected in the application template. This task is performed with the support of the HStringTemplate [32] library.

This library, written by Sterling Clover, is the Haskell port of the StringTemplate Java library by Terrence Parr [67, 68]. StringTemplate is a template engine (with ports also to C#, Python, Ruby, and Scala) for generating source code, web pages, emails, or any other formatted text output. It has been successfully used for multi-targeted code generators, multiple site skins, and internationalization/localization [69].

Figure 4.4 shows the template file (`ROLE_x.st`) for the ROLE_x class. Terms between the

$ delimiters are the templates which are instantiated during the protocol generation. The task performed by StringTemplate is to replace these templates with the actual code. For example

```
$fields:{n|private $n.typeof$ $n.name$ = null;
}$
```

is a template used to declare the private fields of the ROLE_x class. It is filled with the type and the name of the each element taken from the fields component of *JProtocol*.

The resulting code for role *Merchant* in the revised 2KP protocol is the following:

```
private String Price = null;
private String Desc = null;
private Crypto_ByteArray TID = null;
private Crypto_ByteArray SQN4 = null;
private Crypto_ByteArray SQN8 = null;
private Crypto_SealedPair VAR_M_0 = null;
private AnBx_Params VAR_M_DM0VPM = null;
private SecretKey VAR_M_DJI4DM0VPMVPM = null;
private Crypto_HmacPair VAR_M_MDESCDJI4DM0VPMVPM = null;
private Crypto_SealedPair VAR_M_2 = null;
private String VAR_M_4 = null;
private Crypto_SealedPair VAR_M_6 = null;
private SignedObject VAR_M_DM6VPM = null;
private AnBx_Params VAR_M_DDM6VPMUSA = null;
```

This example shows that StringTemplate does not provide just a simple string substitution but makes also possible to use more complex patterns like attributes with properties (`$n.name$`). Moreover it allows applying an anonymous template (`{n|private $n.typeof$ $n.name$ = null;}`) to each element of an attribute (`$fields:<anonymoustemplate>$`).

Following the same approach it possible to generate even more structured code. This is how the method `executeStep()` is specified in the same template file `ROLE_X.st`

```
protected void executeStep(AnB_Session $sessname$, $prot$_Steps step)
   {
  status(step);
  switch (step) {
    $stepactions:{n|
      case $n.astep$:
        $n.action$
        break;
                }$
        }
        status(step);
}
```

In this case, along with simple substitutions of the session name (`$sessnane$`) and of the protocol name (`$prot$`), we can use an anonymous template - `{n|...}` - to generate the cases of a switch statement, namely the actions (`$n.action$`) to be performed in each step (`$n.astep$`). Here we found more productive, to generate sequence of actions directly as a string in Haskell and pass it to the template property `$n.action$` rather than managing templates for all the kind of possible actions.

Finally the generated configuration and application files are written to disk. As an example of this, we show the names of files (left side the template name, right side the application

name) for the revised *2KP* protocol (roles are: A acquirer, M merchant, C customer):

- `CommandLine_Parser.st -> Rev_2KP_CommandLine_Parser.java`
- `Principal.st -> Rev_2KP_Principal.java`
- `ROLE_x.st -> Rev_2KP_ROLE_A.java, Rev_2KP_ROLE_M.java, Rev_2KP_ROLE_C.java`
- `Roles.st -> Rev_2KP_Roles.java`
- `Steps.st -> Rev_2KP_Steps.java`
- `main.st -> Rev_2KP.java`
- `[] -> Rev_2KP.properties`

All the Java files must belong to the same package, `rev_2kp` in this example.

In conclusion, one of the advantages of this way of generating the source code is that it is possible to make modification to the application template without making changes to the tool, as long as the template interface (parameters) are maintained. We think it is an advantage for the user, the possibility to work on the application template editing directly in the target language. The only exception are, of course, the template parameters, which require the StringTemplate syntax.

One the other hand, the option of generating code in another procedural typed language would require a reasonable extra effort and will consist of the following tasks:

- writing new templates files
- defining the syntax of statements like variable declaration, variables initialization (constructors), imperative actions
- setting the binding between the abstract types and the API calls and the concrete one.

The most delicate issue is the availability of a security and communication library as the one we designed. Since all the modern programming languages offer these features, the work will consist of building a wrapper library, around the existing language features, replicating the interface of ours (Figures 4.5 and 4.6)

4.2 API - Java Security Library (AnBxJ)

We developed a Java library to experiment with our approach and validate its practical effectiveness. The package provides an application programming interface (API) that implements the primitives discussed in the previous sections.

To support an high degree of flexibility, the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Instead it is structured as a modular, easily configurable framework that leaves the developer free (at compile, deployment or even at run-time) to decide which cryptographic scheme to use, depending on the requirements of security, robustness and performance the application must satisfy. By default, the system uses the algorithms and the key lengths specified in the digital certificates of the public keys used for encryption and signature. This simplifies a lot the standard usage of the library.

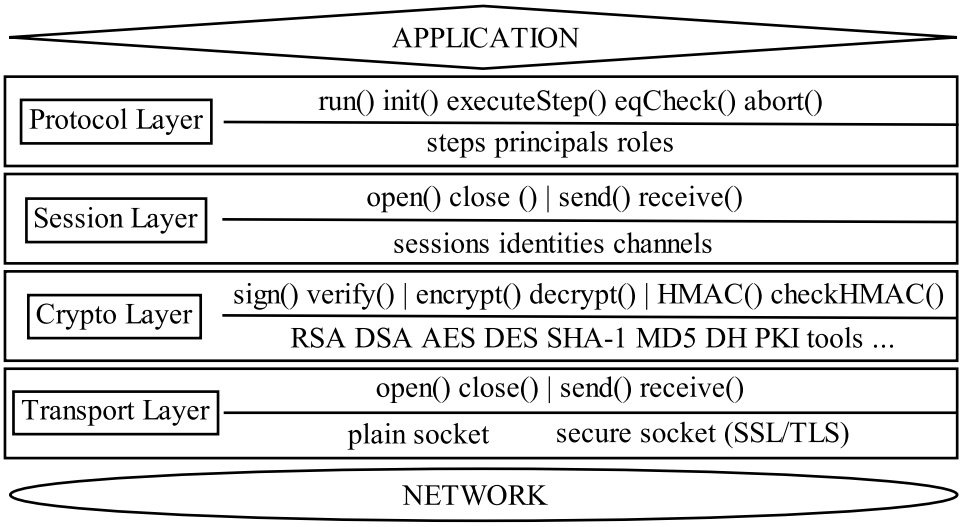

Figure 4.3: *AnBxJ* Java Library Architecture

Several encryption and digital signature algorithms, hash and digest functions, and different key sizes are made available to the application by means of a standard interface. This is done by interacting with the cryptographic libraries, almost transparently, using the standard interface specified by the Java Cryptographic Architecture [44, 71]. Changing the cryptographic protocols and settings is easy, because it does not modify the source code of the application, but only the configuration of the encryption engine in use. The framework is extensible making possible to add new cryptographic libraries, or replacing faulty implementation, or compromised algorithms.

This approach leads to a clear design of the application, focusing on the application logic, abstracting the programmer from the complexity of the underlying network protocols and infrastructure.

The API is structured in the layered architecture described in Figure 4.3, whose main components are described as follows:

- The *transport layer* provides all the networking functionality necessary to transport messages over the network, using both plain and secure sockets (SSL/TLS [38]). Although the enforcement of the security properties is often delegated to the cryptographic layer, is it also possible to run applications over a secured channel rather than over a plain one.

- The *cryptographic layer* essentially provides procedures to encrypt and decrypt, sign and verify, digest messages using the facilities included in libraries like `java.security` and `javax.crypto`. The public key infrastructure (PKI) binds public keys with their respective user identities by means of a certificate authority (CA). Trusted certificates are stored in keystores, and *identities* are defined associating *aliases* with a pair of public keys (one for encryption and one for digital signature).

- The *session layer* offers to the programmer the functions `send()` and `receive()`, which map, respectively, the output and the input primitives. Any *serializable* object can be a *message* exchanged by means of these primitives, thus it is possible to transmit a

wide range of object classes across a network connection link. Primitives to `open()` and `close()` sessions are also provided. Moreover, shielding the details of the cryptographic layer, the `AnB_session` class (Figure 4.6) offers methods for calling, in a simplified manner, the primitives of the cryptographic layer extending the class `AnB_Crypto_Wrapper` (Figure 4.5). Although in the generated code the messages are composed in the *AnB* style (with calls to the crypto API - Figure 4.5), another class could be implemented to provide direct support for the communication modes and digests which are available in *AnBx*, hiding the calls to the cryptographic methods).

- The *protocol layer* gives an abstract description of the protocol: data flow and control flow, steps and principal roles, check on reception. The main class is the abstract generic class `AnB_protocol` (Figure 4.7) which must be extended by the each role class.

4.3 Related Work and Conclusions

Other Java code generators for security protocols has been proposed in the past. An early project [58] allows automatic generation of Java code from the specification of a protocol written in CASPL [37] or in its intermediate language CIL. Although standard Java cryptographic providers are used, this tool has some noticeable limitations. Since at that time a RSA implementation was not publicly available in Java, the tool does not handle public-key encryption.

A couple of tools, both called *Spi2Java* [72, 78] generate Java code from Spi Calculus specifications.

Spi2Java (Pozza et al.) [72], is a framework to semi-automatically generate Java security protocol implementations from verified Spi Calculus formal specifications of such protocols. The aim of the framework is to provide high correctness confidence on the generated code, thus making a step towards bridging the gap between the verified abstract formal models, and their concrete implementations. *JavaSPI* [9] is an evolution of Spi2Java. The main novelty of this approach stands in the use of Java as both a modeling language and an implementation language.

Spi2Java (Tobler et al.) [78] is implemented in Prolog. It can input a formal security protocol specification in a variation of the Spi Calculus, and generate a Java code implementation of that protocol. By defining a Security Protocol Implementation API that abstracts cryptographic and network communication functionality the authors show that the protocol logic code can be separated from the underlying cryptographic algorithm and network stack implementation concerns. Our work shares the same idea although our implementation is different.

Protocol Implementation Generation (PiG) [73] is another tool using a process calculus - *LySa* [20] - as input language. The framework enables the sharing, verification, and translation of communication protocols. With it, partners can suggest a new protocol by sending its specification. After formally verifying the specification, each partner generates an implementation, which can then be used for establishing communication. The target language can be to C or Java.

In summary, in the present work we described a tool for the automatic generation of the Java code of security protocols specified in *AnBx*, an enhanced version of the popular *Alice & Bob* language for narrations. *AnBx* protocols can be model-checked with OFMC in

order to verify their safety. Extending the work of Briais and Nestmann [25], we generate an optimized executable narration, which includes the checks on reception derivable from the static information. Our optimization, as the experimental results has confirmed, improves the protocol execution speed, avoiding repeating the same cryptographic operations on the same data. The generation of the source code keeps apart the protocol logic from the application logic, making possible to extend this work, with a reasonable effort, to other object-oriented or procedural languages. Our experiments showed that our framework can be applied effectively to real-world industrial protocols, like the e-payments applications.

Although, due to the complexity of the target language, we cannot prove the formal correctness of the last phase of the translation (from the executable narration to Java), we think that this experimental work offers some interesting insights in the topic of protocol design and automatic application generation. First of all, we showed the effectiveness of *AnBx* as a language not just for abstract protocol prototyping but also for the generation of concrete implementations. With respect to some of the mentioned tools [58, 73], we take benefit of check generation algorithm presented in [25], to produce Java code which includes checks on reception, which are missing on the other tools. Moreover the same tools do not include a type system to handle complex messages as the one we implemented.

Moreover in contrast to other tools [72, 73, 78] which are using a process calculus as input language, we propose a higher-level and more intuitive language, making our tool suitable for a larger audience. Additionally we designed a Java library for security which can be used not only in conjunction with *AnBx*, but also, in a broader context, even by programmers without a deep knowledge of the security foundations. Last but not least, having an high degree of automation, the tool is suitable for agile prototyping and rapid development of security protocols.

Future work could take several directions. One the one hand it would be important to formally prove the correctness of the generated Java source code with respect to the original specification, or at least extend the formal reasoning to cover all the intermediate formats. On the other hand it would be possible to work on a better engineering of the tool, with the aim to improve its modularity. For example, adding support for more target languages and offering more customization options to the user. A further opportunity will be to plug the tool into an existing Integrated Development Environment, such as Eclipse [42] for example, making the tool suitable to be used in a professional environment.

$Alpha$	::=	$['A'..'Z'] \cup ['a'..'z']$	*alpha chars*
			$Alpha \subset String$
$Ident$	::=	$Alpha + String$	*identifier*
$Agent$	::=	$Ident$	*agent's name*
$IdentList$	::=	$Ident$	*list of identifiers*
	|	$Ident$, $IdentList$	
$Opererator$	::=	**inv**	*inverse function*
	|	**exp**	*exponential function*
	|	**crypt**	*asymmetric encryption*
	|	**scrypt**	*symmetric encryption*
	|	**cat**	*concatenation*
	|	**xor**	*xor*
	|	**apply**	*function application*
Msg	::=	$Ident$	*identifier*
	|	$Operator\ MsgList$	*operator on msg list*
$MsgList$	::=	Msg	*list of messages*
	|	Msg, $MsgList$	
$Type$	::=	**Agent**	*base types*
	|	**Number**	
	|	**Function**	
	|	**PublicKey**	
	|	**Symmetic_key**	
$TypeList$	::=	$Type\ IdentList$	*list of types*
	|	$Type\ IdentList$; $TypeList$	
$Types$	::=	**Types** : $TypeList$	*types*
$KnowItem$	::=	$Agent$: Msg	
$KnowList$	::=	$KnowItem$	*list of agents' know.*
		$KnowItem$; $KnowList$	
$Knowledge$	::=	**Knowledge** : $KnowList$	*knowledge*
$Action$	::=	$Agent\ ChType\ Agent : Msg$	*action*
$ActionList$	::=	$Action$	*list of actions*
	|	$Action$; $ActionList$	
$Actions$	::=	**Actions** : $ActionsList$	*actions*
$ChType$	::=	$\rightarrow$ | $\bullet \rightarrow$	*plain* | *authentic*
	|	$\rightarrow \bullet$ | $\bullet \rightarrow \bullet$	*secret* | *secure*
	|	$\bullet \twoheadrightarrow$	*fresh authentic*
	|	$\bullet \twoheadrightarrow \bullet$	*fresh secure*
$Goal$	::=	$Agent\ ChMode\ Agent : Msg$	*channel goal*
	|	$Agent$ **weakly authenticates** $Agent$ **on** Msg	*weak authentication goal*
	|	$Agent$ **authenticates** $Agent$ **on** Msg	*authentication goal*
	|	Msg **secret between** $AgentList$	*secrecy goal*
$Goals$	::=	$Goal$	*goals*
		$Goal$; $Goals$	
$Protocol$	::=	**Protocol** $Ident$	*protocol definition*
		$Types\ Knowledge\ Actions\ Goals$	

Table 4.7: Syntax of *AnB*

$Alpha$	::=	$[\ldots]$	*alpha chars*
$Ident$	::=	$[\ldots]$	*identifier*
$IdentList$	::=	$[\ldots]$	*list of identifiers*
$Opererator$	::=	$[\ldots]$	*operators*
$Agent$	::=	$Ident$	*agent's name*
	|	$'\_'$	*null agent*
$AgentList$	::=	$Agent$	*list of agents*
	|	$Agent,\ AgentList$	
$Type$	::=	$[\ldots]$	*base types*
	|	**Certified**	*certified agent*
Def	::=	$Ident\,(IdentList) :\ Msg$	*definition w pars*
		$Ident :\ Msg$	*definition w/o pars*
$DefList$	::=	Def	*list of definitions*
		$Def;\ DefList$	
$Defs$	::=	ϵ	*empty definitions*
	|	**Definitions** : $DefList$	*non empty definitions*
$Digest$	::=	$[MsgList]$	*standard digest*
	|	$[MsgList : Agent]$	*verifiable digest*
Msg	::=	$Ident$	*identifier*
	|	$Operator\ MsgList$	*operator on msg list*
	|	$Digest$	*digest*
	|	$PPar\ Ident\ MsgList$	*definition w pars in msg*
	|	$PParId\ Ident$	*definition w/o pars in msg*
$KnowItem$	::=	$Agent :\ Msg$	*agent's knowledge*
	|	$IdentList$ **share** $MsgList$	
$KnowList$	::=	$KnowItem$	*list of agent's knowledge*
	|	$KnowItem;\ KnowList$	
$Knowledge$	::=	**Knowledge** : $KnowList$	*knowledge*
$Action$	::=	$Agent\ ChType\ Agent : Msg$	*AnB action*
	|	$Agent \rightarrow Agent,\ ChMode : Msg$	*AnBx action*
$ActionList$	::=	$Action$	*list of actions*
	|	$Action;\ ActionList$	
$Actions$	::=	**Actions** : $ActionsList$	*actions*
$fresh$	::=	$\epsilon \mid @$	*emtpy* | $fresh$
$forward$	::=	$\epsilon \mid \uparrow$	*empty* | $forward$ *(opt.)*
$Vers$	::=	$AgentList$	*verifiers*
$ChMode$	::=	$forward\ fresh(Agent, Vers, Agent)$	*AnBx channel modes*
$ChType$	::=	$[\ldots]$	*AnB channel types*
$Goal$	::=	$[\ldots]$	*goal*
	|	$Agent$ **confidentially sends** Msg **to** $Agent$	*confidential exchange*
$Goals$	::=	$[\ldots]$	*goals*
$Protocol$	::=	**Protocol** $Ident$	*protocol definition*
		$Defs\ Types\ Knowledge\ Actions\ Goals$	

Table 4.8: Syntax of *AnBx* defined as an extension to the standard syntax of *AnB* ($[\ldots]$)

Reserved identifiers	::=	**pk**	*public key function for encryption*
	|	**sk**	*public key function for encryption*
	|	**hash**	*hash function*
	|	**hmac**	*hmac function*
	|	**g**	*DH base*
	|	**empty**	*sync message*

Table 4.9: Syntax of *AnBx* - Reserved identifiers

```
public final class $prot$_$role$ extends AnB_Protocol<$prot$_Steps,
    $prot$_Roles> {
  private static boolean loop = false;
  // local vars & knowledge
  $fields:{n|private $n.typeof$ $n.name$ = null;
  }$
public $prot$_$role$($prot$_Roles role, String name) {
    super();
    this.role = role;
    this.name = name;
  }
  protected void init() {
     // init local vars
       $fieldsinit:{n| $n.name$ = new $n.typeof$($n.pars$);
     }$
  };
  public void run(Map<String, AnB_Session> lbs, Map<String, String>
     aliases) {
    this.aliases = aliases;
    $channelroles:{n|AnB_Session $n.chname$ = lbs.get(\"$n.chrole$\");
    }$
    try {
      init();
      $channels:{n|$n$.Open();
      }$
      do {
        $channelsteps:{n|executeStep($n.channel$, $prot$_Steps.$n.
            step$);
        }$
      } while (loop);
          $channels:{n|$n$.Close();
          }$

     } catch (Exception e) {
       e.printStackTrace();
       return;
     }
  };
  protected void executeStep(AnB_Session $sessname$, $prot$_Steps step
     ) {
    status(step);
    switch (step) {
      $stepactions:{n|
        case $n.astep$:
              $n.action$
                  break;
      }$
    }
    status(step);
  }
  $rolemethods:{n|private $n.rettype$ $n.mname$($n.mpars$) {
    // TODO Auto-generated method stub
          return ($n.rettype$) $n.retvalue$;
  \}}$
}
```

Figure 4.4: `Role_X.st` file template

```
public class AnB_Crypto_Wrapper {
// implements a class supporting cryptographic operations
// a wrapper for the encryption engine

protected Crypto_EncryptionEngine ee;
protected AnBx_Agent me;
private final static AnBx_Layers layer = AnBx_Layers.LANGUAGE;
public AnB_Crypto_Wrapper(Crypto_EncryptionEngine ee)
public AnB_Crypto_Wrapper(Crypto_KeyStoreSettings_Dual kssd)
public Crypto_KeyStoreSettings_Dual getKeyStoreSettings_Dual()
public void Setup(Crypto_KeyStoreSettings_Dual kssd)

// ------------------------ cert/identities -----------------------
protected Certificate getRemoteCertificate_enc(String alias)
protected Certificate getRemoteCertificate_sig(String alias)
protected AnBx_Agent getMyIdentity()
protected void setMyIdentity()
// ------------------------ send/receive -----------------------
protected void Send_Id(AnBx_Agent id, Channel_Abstraction c)
protected void Send(Object obj, AnBx_Agent id, Channel_Abstraction c)
protected Object Receive(AnBx_Agent id, Channel_Abstraction c)
protected AnBx_Agent Receive_RemoteId(Channel_Abstraction c)
// ------------------------ encrypt/decrypt -----------------------
public Object decrypt(Crypto_SealedPair sc)
public Crypto_SealedPair encrypt(Object object, String alias)
public Object decrypt(SealedObject so, Key symmetricKey)
public SealedObject encrypt(Object object, Key symmetricKey)
// ------------------------ sign/verify -----------------------
public SignedObject sign(Object object)
public Object verify(SignedObject so, String alias)
// ------------------------ nonces, keys, seqnumbers
   -------------------
public Crypto_ByteArray getNonce()
public Crypto_ByteArray getSeqNumber()
public SecretKey getSymmetricKey()
public SecretKey getHmacKey()
// ------------------------ key exchange
   -------------------------------
public KeyPair getKeyEx_KeyPair()
public PublicKey getKeyEx_PublicKey(KeyPair keyPair)
public SecretKey getKeyEx_SecretKey(KeyPair keyPair, PublicKey
   publicKey)
public SecretKey getKeyEx_SecretKey(PublicKey publicKey, KeyPair
   keyPair)
// ------------------------ digest hash/hmac
   -----------------------------
public Crypto_ByteArray makeDigest(Object obj)
public Crypto_ByteArray makeDigest(Object obj, String str)
public Crypto_HmacPair makeHmac(Object obj, SecretKey sk)
public boolean checkHmacPair(Object obj, Crypto_HmacPair hmac,
   SecretKey sk)
}
```

Figure 4.5: *AnBxJ*: Crypto API (`AnB_Crypto_Wrapper` class)

```
public class AnB_Session extends AnB_Crypto_Wrapper {
// implements a session supporting cryptographic operations

private final static AnBx_Layers layer = AnBx_Layers.SESSION;
private Channel_Abstraction c;
private AnBx_Agent id_Remote = null;
private Boolean exchange_id = false;
// allow agents to exchange their aliases

public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings
    cs, boolean exchange_id)
public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings
    cs, AnBx_Agent id_Remote)
public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings
    cs, String id_Remote_alias)

// ------------------------ open/close ----------------------
public void Open()
public void Close()
// ------------------------ send/receive --------------------
public Object Receive()
public void Send(Object obj)
public AnBx_Agent Receive_RemoteId()
public void Send_Id()
// ------------------------ setters/getters -----------------------
public Channel_Abstraction getC()
public AnBx_Agent getId_Remote()
public void setC(Channel_Abstraction c)
public void setId_Remote(AnBx_Agent id_Remote)
}
```

Figure 4.6: *AnBxJ*: Communication API (`AnB_Session` class)

```
public abstract class AnB_Protocol<S, R> {

private final static AnBx_Layers layer = AnBx_Layers.PROTOCOL;
protected String name = null;
protected R role;
protected Map<String, String> aliases;
private boolean abortOnFail = false;

abstract public void run(Map<String, AnB_Session> lbs, Map<String,
   String> aliases);
abstract protected void executeStep(AnB_Session lbs, S step);
abstract protected void init();

protected void abort(String msg)
protected void eqCheck(Object obj1, Object obj2)
protected void status(S step)
}
```

Figure 4.7: *AnBxJ*: `AnB_Protocol` class

II

Sevecom

5

Verifying Sevecom using Set-based Abstraction

5.1 Introduction

The EU-project SeVeCom [66,75] proposes a modern system for secure vehicle communication that shall satisfy two seemingly conflicting goals, namely on the one hand authentication and accountability for vehicle communication and protecting the privacy on the other hand. To that end, each car contains a tamper-proof *hardware security module* (HSM) that holds all private keys of the car and that performs all encryption, decryption, and verification operations. For ordinary communication, this includes a number of short-term key-pairs that are registered with, and certified by, a trusted certification authority (CA). While the CA is thus able to link all communication back to a particular car (e.g. in case of police investigation), the other participants cannot see this relation, but only link actions that are performed using the same *pseudonym* (i.e. short-term key).

There is a growing number of automated tools for protocol verification that represent the cryptographic messages by abstract terms (and thereby ignore cryptographic attacks). Many such tools like Scyther [34] are restricted to "simple" protocols that consist only of a message exchange and therefore cannot analyze a system like SeVeCom that requires the participants to maintain databases of keys. Other tools like AVISPA [6] allow for modelling databases but require restricting the number of steps that honest agents can execute, and do not scale well with this number. There are several abstraction-based approaches [18, 21, 23], most notably the tool ProVerif [19], which completely avoid this problem and allow for verification with an unbounded number of steps. These techniques however have a kind of *monotonicity* built-in: what is true at some point cannot become false later, forbidding to model revocation for instance and are thus not suitable for analyzing SeVeCom.

The AIF framework [60] is an extension of the abstraction approach that allows for modeling databases (or sets) of messages that do not necessarily monotonically grow (allowing for revocation) and that inherits the benefits from the classical abstraction approaches, namely verification without bounding the number of steps that honest and dishonest participants can make.

We show that AIF is indeed well-suited for modelling and verifying the SeVeCom system. We consider several models of the system for different intruder models: one model considers the revocation-update protocols of the CA's root keys in the presence of an intruder with direct access to the HSMs. The second, comprehensive model of all the protocols needs to consider an intruder who is either outside a particular car or at least does not have access to

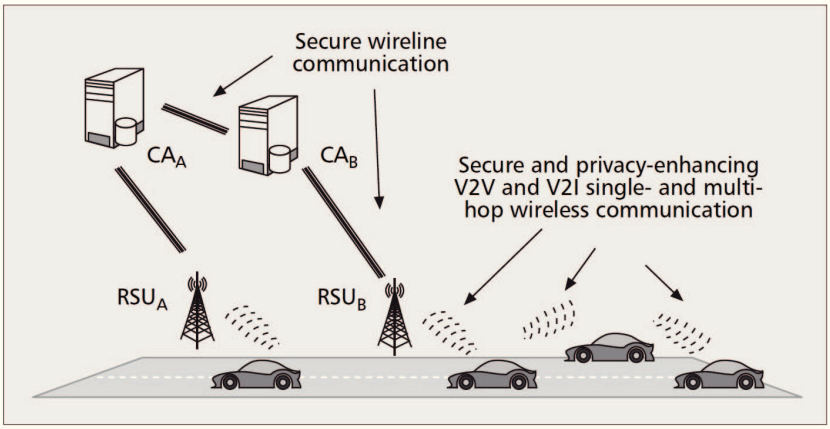

Figure 5.1: A Vehicular Communication System

the signing function of the HSM (which would lead to trivial attacks).

In our models we assume the hardware and software components to be reliable, and the cryptographic functions secure. Therefore we do not consider events like communication errors or attacks that can be performed exploiting weaknesses of the cryptographic algorithms. We present two novel attacks that were found by our analysis in the root key update protocol, and discuss some reasonable assumptions to prevent them. We verify the security properties of the system under these assumptions. Beyond the verification of the concrete system SeVeCom, this work demonstrates how the relevant aspects of time can be modeled in verification approaches that actually *abstract from time*.

5.2 Secure Vehicle Communication

A *vehicular communication* (VC) system (Figure 5.1)[1] comprises several network nodes: vehicles and *road-side* infrastructure *units* (RSU) which are equipped with on-board sensory, processors, and wireless communication modules. It is customary to distinguish between *vehicle to vehicle* (V2V) and *vehicle to infrastructure* (V2I) communications. The *Certification Authorities* (CA) are trusted entities responsible for the issuance and management of identities and credentials for parties involved in the vehicular network operation. In general, the authorities can be multiple and distinct in their roles and jurisdiction over a subset of network.

The *OnBoard Units* (OBU) are the computing devices which are placed on-board of vehicles. In SeVeCom public key operations are performed by the OBU, but all private key operations are performed by the HSM, the *Hardware Security Module* which is the trusted computing base of SeVeCom security architecture. The HSM stores the private cryptographic key material and provides cryptographic functions employed by other modules.

The HSM has a CPU, a memory module, and some non-volatile storage. In addition, in order to ensure the freshness of the encrypted or signed messages, the HSM must also include a real-time clock, and consequently, a battery module that ensures the clock can operate independently from the rest of the system. The HSM has also a hardware random number generator that is used for key generation purpose. The HSM is physically separated from the OBU, and it has some tamper-resistant properties in order to protect the private key

[1] Figures 5.1, 5.3, 5.4 are borrowed from [66]

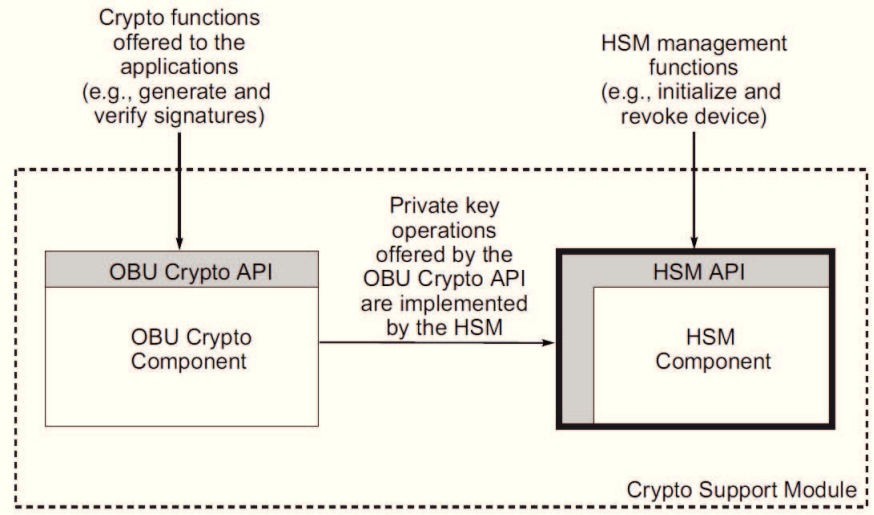

Figure 5.2: Crypto Support Module

material against physical attacks. The Crypto Support Module [75, §7] (Figure 5.2)[2] provides the implementation of the cryptographic operations needed by the applications running on the OBU. These applications can call the cryptographic operations through an APIs offered by the components of the Crypto Support Module.

The SeVeCom architecture (Figure 5.3) addresses the following fundamental issues:

- *Identity, credential, and key management*: each node is registered with only one CA, and has a unique long-term identity and a pair of private and public cryptographic keys, and it is equipped with a long-term certificate. A list of node attributes and a lifetime are included in the certificate that the CA issues upon node registration and upon certificate expiration. The CA is also responsible for the eviction of nodes or the withdrawal of compromised cryptographic keys via the revocation of the corresponding certificates.

- *Secure communication* (Figure 5.4): digital signatures are the basic tools to secure communications and are used for all messages. To satisfy both the security and anonymity requirements, SeVeCom relies on a pseudonymous authentication approach. Rather than using the same long-term public and private keys for securing communications, each vehicle employs multiple short-term private-public key pairs and certificates. A mapping between the short-term credentials and the long-term identity of each node is maintained by the pseudonym provider (PP), which is a particular instance of a Certification Authority within the PKI. On the contrary, since privacy is not an issue in this case, RSUs always use the long-term keys.

5.3 AIF

The AIF framework consists of the AIF specification language and a translator from AIF to first-order Horn clauses that incorporates the set-abstraction technique; it is connected to

[2]Figure 5.2 is borrowed from [75]

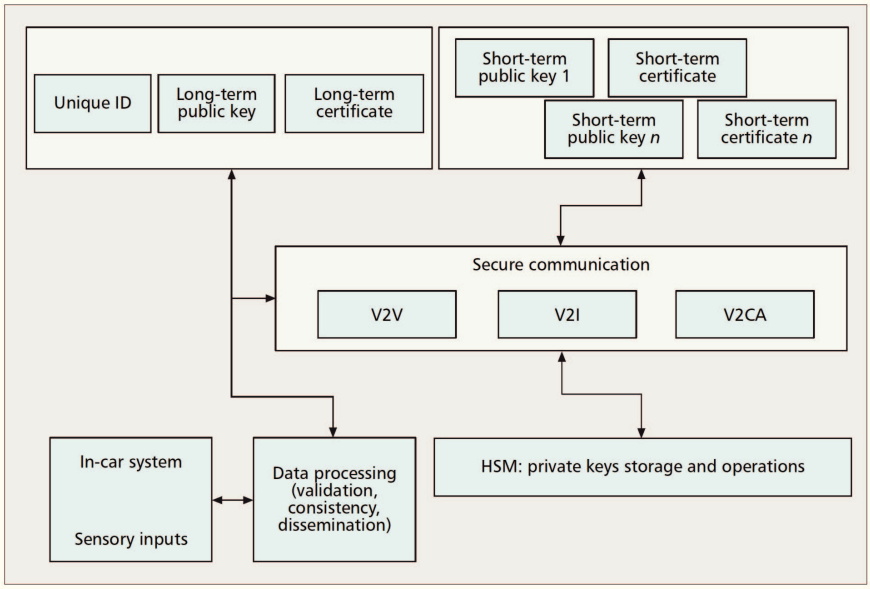

Figure 5.3: SeVeCom Architecture

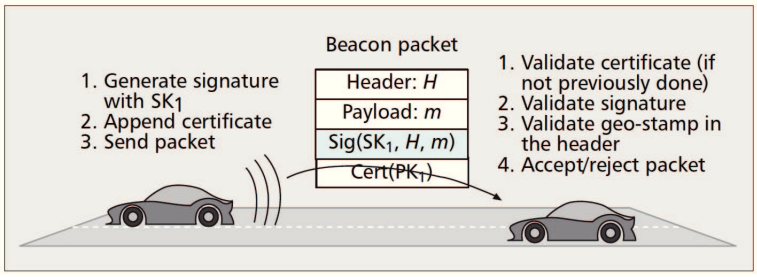

Figure 5.4: Secure V2V Communication

the SPASS theorem prover [81] and the protocol verifier ProVerif [19] to check the generated Horn clauses. These automated tools do not always terminate, but when they do, this gives us either an attack or a proof of security. We give a brief introduction to the AIF language; a formal definition is found in [60].

State Transition Systems An AIF specification describes a state-transition system. A *state* is a set of *facts* that are true in that state, for instance the state

$$\{ik(crypt(k,m)), ik(inv(k)), ik(sign(inv(k'),m))\}$$

may intuitively say that in this state, the intruder knows a public-key encrypted message $crypt(k,m)$, the corresponding private key $inv(k)$ and a signature $sign(inv(k'),m)$. None of the symbols here has a predefined meaning. Their meaning is rather defined through the *transition rules* that we define on states. For instance we may define rules that reflect the ability of the intruder to encrypt and decrypt messages with known keys:

$$\begin{aligned}
&ik(K).ik(M) \Rightarrow ik(crypt(K,M));\\
&ik(crypt(K,M)).ik(inv(K)) \Rightarrow ik(M);\\
&ik(inv(K)).ik(M) \Rightarrow ik(sign(inv(K),M));\\
&ik(sign(inv(K),M)).ik(K) \Rightarrow ik(h(M));
\end{aligned}$$

Observe that in specifying such rules, we use *variables*, denoted by identifiers that start with an uppercase letter, in contrast to constants which start with a lowercase letter. Such a rule can be applied to any state that contains facts that match the left-hand side; this yields a new state that is obtained from the old one by adding the facts of the right-hand side under the given match.

We can specify transitions in which new values are freshly created, e.g., we can specify that at any time the intruder can generate himself a new key pair as follows:

$$=[K]\Rightarrow ik(K).ik(inv(K)); \tag{5.1}$$

Taking this transition, the variable K is bound to a new value (that did not occur so far), representing in this case a public key. Since the left-hand side is empty, the rule can be taken without any precondition.

Sets The rules as presented up to here represent what is standard in abstraction approaches, and in particular observe the states can only monotonically grow during such transitions. This does not allow for example the modeling of a transition where a key is revoked, because the fact that the key is valid would need to be somehow "retracted". Exactly for such cases, the AIF has a way to express transitions in which the state does not monotonically grow, namely using *sets*. An AIF specification can contain an arbitrary but fixed number of sets.

For instance, our SeVeCom model will include several sets of public-keys that the HSMs of the cars maintain, e.g. $db(hsm1, ltsig, uptodate)$ denotes the set of all up-to-date long-term signing keys stored in machine $hsm1$. We can conveniently describe an enumeration of such sets using variables that range over enumeration types, for instance in this case we have a family of 28 sets $db(HSM, KeyType, Updating)$ where

$$\begin{aligned}
HSM &: \{hsm1, hsm2\};\\
KeyType &: \{root1, root2, ltsig, ltdec, stsig, stdec, ppsig\};\\
Updating &: \{updating, uptodate\};
\end{aligned}$$

In detail the HSM maintains five types of keys:

- two *long-term root public keys* that are used to verify the authenticity of commands (e.g., revocation of the HSM) sent by the authorities to the HSM;
- a *long-term signature generation key* that is used to authenticate the real identity of the vehicle;
- a *long-term decryption key* that is used to decrypt encrypted messages intended to the vehicle itself;
- *short-term signature generation keys* that are typically used to authenticate the short-term pseudonyms used by applications running on the vehicle;
- *short-term decryption keys* that are used to decrypt encrypted messages intended to the applications running on the vehicle.

Thus, keys are organized by type: the above five types plus the signing and decryption keys of the *Pseudonym Provider* PP (details on these keys are given in Section 5.5.4). Moreover the keys could be "under update" or not: this is shown by the *Updating* status.

The database shows explicitly only the public-keys. For keys of type $\{lt, st\}sig$ and$\{lt, st\}dec$ the corresponding private keys are actually stored on the device, but this is implicit in our model.

To build a comprehensive model we also need to define some centralized databases of keys which are maintained by the Certification Authorities:

- $dbr(Root)$, the centralized database of root keys, where $Root : \{root1, root2\}$;
- $dbca(HSM, LongTerm)$, the centralized database of the long term keys, where $LongTerm : \{ltsig, ltdec\}$;
- $dbps(HSM, ShortTerm)$, the centralized database of pseudonyms, where $ShortTerm : \{stsig, stdec\}$;

Transitions Using Sets A *set-membership fact* is a fact of the form $m \in S$ where m is an element and S is a set. As an example, there are two public keys of the certification authority, called *root keys*, stored into every HSM at manufacturing time (more on their role in the system later). We can model this initialization of an HSM by a transition rule that simply creates new root keys:

$$\begin{array}{l} \lambda HSM. = [K_1, K_2] \Rightarrow ik(K_1).ik(K_2). \\ K_1 \in db(HSM, root1, uptodate). \\ K_2 \in db(HSM, root2, uptodate); \end{array} \tag{5.2}$$

Here, $\lambda HSM.$ says that this rule holds for any value in the domain of variable HSM ($\{hsm1, hsm2\}$ here) to avoid long enumerations. This rule is an over-approximation of reality, because it can be applied at any time and any number of times, while in the real system, there can only be one pair of root keys installed at manufacturing time. Formulating it in this way is necessary in the abstraction approaches as we explain below. We can now model that an HSM, when receiving a correctly formed revocation message, marks the respective key as being "under update" in its database:

$$\begin{array}{l}\lambda HSM, Root.ik(sign(inv(K), [K, T])).\\ K \in db(HSM, Root, uptodate)\\ \Rightarrow K \in db(HSM, Root, updating);\end{array} \tag{5.3}$$

Here, *Root* ranges over $\{root1, root2\}$. Moreover, $sign(inv(K), [K, T])$ is the format of a revocation command for a root key: it needs to be signed by the private key $inv(K)$ that belongs to the root key K. For simplicity, we will often say *private root key* for the private key that belongs to a public root key. The message also includes a timestamp T that we discuss later.

We model here an HSM that is directly under the control of an intruder who can send arbitrary commands to it. This is expressed by the *ik*-fact on the left-hand side of the rule: the HSM accepts any command of the revoke-key format that the intruder can craft (as long as the respective key K is indeed in $db(HSM, Root, uptodate)$).

Similarly, for other operations where there is an answer by the HSM, we will have this answer contained in an *ik* fact on the right-hand side of the rule to model that the intruder directly obtains this answer, can analyze it, and use it for further actions like crafting another command.

Also observe that the set-membership facts on the left-hand and right-hand side differ by their update-status. While for all other facts, the state monotonically grows over transitions, set-membership facts behave differently: left-hand side facts that do not appear on the right-hand side get *removed* by the transition. Thus, the matched key K in this example is moved from the *uptodate* to the *updating* set (of the respective machine and key kind). In fact, the other transitions ensure that only signatures with up-to-date keys are considered as valid. On the left-hand side of rules, we may also specify that a rule is only applicable to states in which a certain set-membership fact does *not* hold. For instance if we declare a family of sets $used(HSM)$, we can model a simple replay-prevention:

$$\begin{array}{l}\lambda HSM, Root.ik(sign(inv(K), [K, T])).\\ T \notin used(HSM).K \in db(HSM, Root, uptodate)\\ \Rightarrow T \in used(HSM).K \in db(HSM, Root, updating);\end{array} \tag{5.4}$$

This transition can only be taken for a timestamp T that the *HSM* has never seen before and that is afterwards stored as used. Here we do not model any properties of time (like freshness); we come back to timestamps later.

Goals and Reachability AIF has only one built-in fact symbol: attack. We use rules that have this fact on their right-hand side to specify *attack states*. For instance we can specify that it is an attack if the intruder finds out the private key of a valid root key:

$$\begin{array}{l}\lambda HSM, Root, Updating.K \in db(HSM, Root, Updating).\\ ik(inv(K)) \Rightarrow \mathsf{attack};\end{array}$$

The *initial state* is the empty set of facts. We say that an AIF specification is *secure*, if we can reach no attack state from the initial state by using the transition rules.

Abstraction Ideally, when writing a model, one does not need to think about the techniques that are used to analyze it, but unfortunately the complexity of the problems, and the side conditions that several techniques have, usually require a certain level of technical

knowledge. The AIF framework is based on abstraction techniques and the AIF language is designed so that all requirements for the *soundness* of the abstraction are satisfied by construction. Soundness here means: if the abstraction of an AIF specification is secure, then so is the concrete AIF specification. The other direction does not hold in general, because the abstraction over-approximates the behavior of the concrete system and can thereby introduce attacks that have no counter-part in the concrete model. We call such attacks *false positives*.

The main point of set-based abstraction is that we consider the equivalence classes induced by the set-membership of values; in our example, the abstract model identifies all public-keys that belong to the same subset of all the databases. For this to be sound, it is crucial that the specification cannot distinguish several values that have the same membership status, so we cannot write conditions like $X \neq Y$, not even indirectly. That implies that we cannot control the cardinality of sets (that they contain a particular number of elements). More generally, we can say that what is true for one value is also true for any number of values in some reachable state.

There are two main consequences for our model of $SeVeCom$. First, we need to allow that all sets of keys can hold any number of keys; this is of course sound in the sense that it over-approximates the real behavior where number is fixed. It turns out that this over-approximation does not introduce any false attacks. Second, we cannot directly talk about time (and timestamps) because the abstract model eliminates every notion of transitions and the timely order of events. We must therefore use sets to model crucial properties of time when they are needed. For instance in the example rule (5.4), we have only modeled the aspect of replay checking, but not recentness; we do the latter not before Section 5.5 where it becomes unavoidable.

5.4 Root Key Update

We first consider a model that focuses on the root keys and their update (and ignores all other kinds of keys and protocols). We use from the previous section the initialization rule (5.2) and the revocation rule (5.3) (without replay check). The next corresponding operation that we model is the update operation which is triggered by a command of the form $sign(inv(K), [K', T])$ where K is a root key in uptodate status, and a new root key K' that is to replace the other root key (which must be in updating status). The rationale behind the format of the update and revoke command is that, if one of the root private keys is compromised, it can only be used to revoke itself, but cannot be used to update either key (which requires the knowledge of both root private keys). The intention is that the system should be secure as long as at most one of the two keys is compromised. The update for key *root1* is formalized as follows (*root2* analogously):

$$\begin{aligned}
&\lambda HSM.ik(sign(inv(K_2), [K, T])).\\
&K_2 \in db(HSM, root2, uptodate).\\
&K_1 \in db(HSM, root1, updating)\\
&\Rightarrow K_2 \in db(HSM, root2, uptodate).\\
&K \in db(HSM, root1, uptodate).K_1 \in revoked(HSM);
\end{aligned}$$

Here, we again ignore the timestamp T at first. We also use a new family of sets here: $revoked(HSM)$. They contain all the public root keys that have ever been discarded from an HSM and are used later to formulate the goals.

5.4.1 Modeling the Authority

The revocation and update messages should be (at least in normal protocol runs) be generated by the certificate authority (CA), the (supposed) owner of the root keys. This will happen whenever a root key is suspected to be compromised, or maybe even on a regular basis. In a first model, we consider a CA that can at any point revoke either root key. Let us first consider a model where the authority can generate a pair of revoke and update commands at any time non-deterministically. Because we cannot be sure a priori that the update is correctly communicated, we must model the CAs database of root keys independent of the HSMs databases. To that end we use the two sets $dbr(Root)$ (recall $Root : \{root1, root2\}$). The revoke and update request is produced in one transition; we give the one for revoking and updating $root1$:

$$\begin{array}{l} K_1 \in dbr(root1).K_2 \in dbr(root2). \\ =[K]\Rightarrow K \in dbr(root1).K_2 \in dbr(root2). \\ ik(sign(inv(K_1), [K_1, T])).ik(sign(inv(K_2), [K, T])); \end{array}$$

5.4.2 Goals

We consider three goals:

Secrecy The intruder never knows the private key of a valid (or under update) public root key:

$$\begin{array}{l} \lambda HSM, Root, Updating. \\ K \in db(HSM, Root, Updating).ik(inv(K)); \\ \Rightarrow \mathsf{attack}; \end{array}$$

Authentication The intruder shall not be able to produce a confusion among the parties about who generated which keys and for which purpose. For the root key update, the only potential confusion is that the intruder manages to make the HSM accept an intruder-generated key as a root key. This would mean a violation of the secrecy goal already (because the intruder knows the private key of a self-generated public key pair). For the comprehensive model in Section 5.5, however, there are more interesting authentication properties.

Freshness The intruder shall not be able to introduce old keys into the HSM, even if they were once created by the correct party and the intruder does not know the private key. We want this property to prevent that older messages using these keys could be accepted again by anybody. To that end, we use the set $revoked(HSM)$ that we introduced before to hold all revoked keys:

$$\begin{array}{l} \lambda HSM, Root, Updating.K \in revoked(HSM). \\ K \in db(HSM, Root, Updating) \Rightarrow \mathsf{attack}; \end{array}$$

Observe that these goals have similarity with classic goals of secure communication, but adapted to the specific problem at hand.

5.4.3 An Attack

We first get a violation of the freshness goal and one that even works if the timestamps are checked for recentness. The attack uses the fact that the CA can generate revoke-update

commands arbitrarily for the existing keys and the intruder does not even need to know any private root keys for the attack to work. Suppose we initially have two root keys installed on the HSM, called k_0 and k_1. Suppose further, the authority revoke-updates the key k_1 to k_2 and then from k_2 to k_3 and so on. This will produce update messages of the form $sign(inv(k_0), [k_n, T])$ (for some timestamp T). At any such update, the intruder can block the current update message and replay an old message in order to install an older key k_i.

This attack is limited (but not prevented) by the use of timestamps: this works only if several updates are performed within a too short time, i.e. with overlapping validity periods of the timestamps. Since it is reasonable to assume that root-key updates are performed rather infrequently, this attack is not of such a practical relevance, but it suggests actually several additional security measures that are at least not explicitly mentioned in [75].

First, revoke-updates should be performed only with non-overlapping validity periods of the timestamps. Second, the HSMs should store all timestamped messages for the time they are valid and compare every further incoming message with them to prevent replay. Third, it may be a good idea to include into the update message also the public key that is supposed to be revoked and updated. Each of these suggestions can prevent the attack and each seems reasonable and good practice. An easy way to model the replay-prevention in the HSM is shown by rule (5.4), requiring a novel timestamp in every message. We can do the same for the update command of the HSM, but need to ensure that the revoke and update command are generated with different timestamps. This rule does not really model recentness of timestamps (so the intruder may arbitrarily delay the delivery of a message to the HSM in this model); we consider another time model in Section 5.5.

5.4.4 Revoking the wrong key

We now check what happens if the intruder is given one of the two root keys. The following rule gives the intruder all the private root keys stored as *root1* on the HSM.

$$\lambda HSM.K \in db(HSM, root1, uptodate).$$
$$\Rightarrow K \in db(HSM, root1, uptodate).ik(inv(K));$$

Consequently, we restrict the secrecy goal to private keys of *root2*. But also this has an attack namely if the CA happens to revoke the wrong key, i.e., if the CA wrongly thinks that *root2* is compromised, and issues a revoke-command for a *root2* key, i.e. $sign(inv(k_2), [k_2, T])$ (for some *root2*-key k_2 and timestamp T). Since we have given the intruder a *root1*-key k_1, he can now issue the update command for k_2, namely $sign(inv(k_1), [k_3, T])$ for some intruder-generated key k_3. It is indeed unclear, even assuming that the intruder can only know one of the root keys, how the CA can be sure which one it is.

While the protocol suggests a complete symmetry between the keys, one may think of using the keys in distinct ways. Suppose the *root1* is used for the daily business, while the private *root2* is kept reserved for emergencies, maybe under additional physical protection. Then it makes sense to assume that the intruder can only find out *root1* and if there is any suspicion of compromise (or also at regular time intervals), we use *root2* to update it, and we never update *root2* itself.

Restricting our model to only updates of *root1* keys, we have verified the secrecy goal.

5.5 Comprehensive Model

We now extend our previous model considering also the long-term and the short-term keys and the related update protocols. We first show how to integrate the notion of time into our model and discuss the modelling of the intruder.

5.5.1 A Timed Model

Time plays an important role in the SeVeCom protocols, namely for the validity of key certificates and timestamps to prevent replay. Each HSM (and the CA) has its own clock. These clocks may differ by a certain margin δ; as long as δ is much smaller than the validity period of long-term keys, this may in the worst case disrupt communication for time δ at the end of a key's validity period, but not endanger any security properties. We therefore simply assume synchronized clocks.

The abstractions of the AIF framework do not provide any notion of time and so we cannot directly talk about the order in which events occur. Despite this fact, we can model some properties of time using the sets. The idea is to divide the timeline into several epochs. For SeVeCom, we find a split into three epochs suitable: $old, expsoon$ and new. We want that the abstraction of keys depends (besides the databases that we already have) also on the epoch that they belong to. We thus define a family of sets $timer(Time)$ where $Time$ ranges over $\{old, expsoon, new\}$.

These epochs are used to model the life-cycle of the key as follows. A key is first freshly created by an HSM and is in epoch new; the key cannot be used yet, the HSM first needs to run a key registration protocol with the CA. After registration, the key becomes valid and moves from status new to status $expsoon$. This means that the HSM can now use the key for encryption or signing and a process to generate and register a new key can be started. After some time, the key finally moves from $expsoon$ to old and can no longer be used; the HSM will delete the old key (but an intruder can still try to use/re-introduce it). Let us look at these steps of the key life cycle in more detail. When the HSM has a key K that is currently in use, i.e., in epoch $expsoon$, it can generate a new key NK which is initially in epoch new. This rule has the form:

$$\begin{array}{l} K \in timer(expsoon).K \in db(\cdot)\ldots \\ =[NK]\Rightarrow K \in timer(expsoon).NK \in timer(new). \\ K \in db(\cdot).ik(\cdot); \end{array}$$

where $ik(\cdot)$ abbreviates an outgoing certificate request message for the new key NK (to the CA) and $\ldots$ represents some other facts.

The second step models the actual progress of time:

$$K \in timer(expsoon) \Rightarrow K \in timer(old);$$

i.e. a key K can change its status from $expsoon$ to old—and this can happen "at any time" so to speak: the "world" that we model here can just choose to do such a transition for any $expsoon$ key. Observe that this progress of time is independent of what the parties are doing at this time.

The third step is now expressing that, if an HSM has key K currently in use and a fresh key NK has successfully been registered with the CA, then as soon as K has turned old (with

the previous rule), we can discard K and start using NK as the current key:

$$K \in timer(old).NK \in timer(new).K \in db(\cdot).NK \in db(\cdot)$$
$$\Rightarrow K \in timer(old).NK \in timer(expsoon).NK \in db(\cdot).$$
$$K \in revoked(HSM) \ldots;$$

Thus NK's transition from *new* to *expsoon* happens exactly when the HSM starts using it.

5.5.2 Modelling the intruder and the API

The API of the HSM offers the interface through which applications running on the car can invoke the functionality provided by the HSM. In particular, besides keys and device management, decryption and digital signing are the two main functions offered by the API. They can be employed with the long-term and short-term keys. Recall that the corresponding private keys are stored in the HSM memory and never released outside. As an example, for long-term decryption keys (that have the attribute *ltdec*), we model the decrypt function as follows in AIF:

$$\lambda HSM, Updating.ik(crypt(K, M)).$$
$$K \in db(HSM, ltdec, Updating)$$
$$\Rightarrow ik(M).K \in db(HSM, ltdec, Updating);$$

Note that *Updating* is a variable in order to model the ability to employ the keys regardless their status. Similarly we have a rule for signing:

$$\lambda HSM, Updating.ik(M).K \in db(HSM, ltsig, Updating)$$
$$\Rightarrow ik(sign(inv(K), M)).K \in db(HSM, ltsig, Updating);$$

Observe that this allows the intruder to get a signature with any valid signing key in the HSM on any message he can construct (and similarly he can decrypt any message that is encrypted with an HSM-stored decryption key). To put it another way: if the intruder has direct access to an HSM, then there is not much difference from the intruder knowing the respective private keys himself—only he cannot get self-generated keys *into* the HSM. With this, of course, the intruder can trivially break several goals of the key update protocols; the only exception is the root-key update which is secure (under the given assumptions) even for an intruder with direct access to HSMs.

It is reasonable to assume that the intruder has only direct access to the HSM(s) in his own car(s). For other cars, he can only observe the communication with their environment (inter-car, road signs, and CA).

In the following we thus consider an attack model where the intruder has limited access to the HSMs; experimenting with different settings we can verify all our goals if the intruder cannot access the signature function for long-term keys, while we may still give him access to all other functions without breaking the security.

5.5.3 Long-Term Key Update Protocol

The long-term signature generation key of the HSM is used to authenticate messages with the real identity of the HSM. The long-term decryption key of the HSM is used to decrypt encrypted messages that are intended for the vehicle. These keys are generated by

the HSM typically at manufacturing time, and the CA creates the certificates for such keys ($LongTerm : \{ltsig, ltdec\}$):

$$\begin{aligned}&\lambda HSM, Root, LongTerm.RK \in dbr(Root)\\ &=[K]\Rightarrow K \in db(HSM, LongTerm, uptodate).\\ &K \in dbca(HSM, LongTerm).RK \in dbr(Root).ik(K).\\ &ik(cert[HSM, K, RK]).K \in timer(expsoon);\end{aligned}$$

Here, $cert[HSM, K, RK]$ denotes a certificate by the CA that K is a key of HSM (signed with the private root key $inv(RK)$). $K \in timer(expsoon)$ ensures that K is directly usable (as the certification is already finished) and at any time we can start the update for a new key. As explained before, the key update happens in several phases. First, the HSM generates new long-term key pairs and produces a certificate request message for the CA and waits for receiving a corresponding certificate. The second phase begins when the current keys expires, and only now the HSM starts to use the newly generated key. The goals for the long-term keys are similar to those for the root keys, only here we have a relevant authentication goal. We formalize that whenever the CA has recorded the key K as a valid long-term key (for signing or decryption) of a particular HSM, then this machine also has K stored in the database (either in status *uptodate* or *updating*). It would thus count as an attack, if the intruder manages to confuse the CA about the long-term keys of an HSM. This is formulated as follows in AIF:

$$\begin{aligned}&\lambda HSM, LongTerm.K \in dbca(HSM, LongTerm).\\ &K \notin db(HSM, LongTerm, updating).\\ &K \notin db(HSM, LongTerm, uptodate) \Rightarrow \mathsf{attack};\end{aligned}$$

This goal is particularly important for accountability, because it ensures that the CA does never attribute keys to a wrong HSM. We verified that our model is safe under the assumption that the intruder does not have full access to the long-term signature function.

5.5.4 Short-Term Key Update Protocol

The short-term keys are used to sign or decrypt the periodic beacon messages broadcasted or received by the vehicle. For privacy reasons, the public keys that correspond to these short-term private keys may be certified in an anonymous manner by a trusted third party, called the pseudonym provider (PP), which is a particular instance of CAs within the PKI. An anonymous certificate contains only the public key, the validity period of the certificate, the identifier of the issuer, and the digital signature of the issuer. In particular, it does not contain the identifier of the vehicle to which it has been issued.

However the HSM does not store the certificates but only supports the pseudonym management by generating short-term key pairs, storing the private keys, and computing signatures. The HSM can be instructed (through its API) to generate a new short-term signature key pair. When the HSM generates a new key pair, it creates a new entry in the internal key database and it stores the private key together with the corresponding context information and outputs the public key. It is the responsibility of applications running on the car to obtain a certificate for it. The certificate request is passed back to the HSM for being signed with the long-term signature key of the HSM.

Also for short-term keys we check the above mentioned goals of secrecy, authentication and freshness. The SeVeCom specification [75, §4.2.4] assumes an authentic and confidential

communication channel between the HSM and the PP. We made experiments both with realizing the channels with the long-term keys of the HSM, and by using an ideal channel model that abstracts from the implementation [63]. In both cases, we can verify that the system is safe, even allowing the intruder to have access to the short-term signature function.

Conclusions

We presented *AnBx*, the currently most expressive Alice and Bob style language. The distinguishing key-features of the language is a powerful concept of channels that includes forwarding. We analyzed both the formal details related to the definition of the language and the practical issues about a realistic cryptographic implementation of the introduced high-level security abstractions. We showed the amenability of the language for the analysis of real-life protocols from the e-payment area, namely *iKP* and *SET*, and we argue that the abstraction from low-level security mechanisms turns out to be helpful also for protocols designers. Our compiler from *AnBx* to IF is available online[3] along with the related documentation and the source code of both our case studies.

In chapters 3 and 4 we described a tool for the automatic generation of the Java code of security protocols specified in *AnBx*. Extending the work of Briais and Nestmann [25], we generate an optimized executable narration, which includes the checks on reception derivable from the static information. Our optimization, as the experimental results has confirmed, improves the protocol execution speed, avoiding repeating the same cryptographic operations on the same data. The generation of the source code keeps apart the protocol logic from the application logic, making possible to extend this work, with a reasonable effort, to other object-oriented or procedural languages. Our experiments showed that our framework can be applied effectively to real-world industrial protocols, like the e-payments applications.

Although, due to the complexity of the target language, we cannot prove the formal correctness of the last phase of the translation (from the executable narration to Java), we think that this experimental work offers some interesting insights in the topic of protocol design and automatic application generation. First of all, we showed the effectiveness of *AnBx* as a language not just for abstract protocol prototyping but also for the generation of concrete implementations. With respect to [58, 73], we take benefit of check generation algorithm presented in [25], to produce Java code which includes checks on reception, which are missing on the other tools. Moreover the same tools do not include a type system to handle complex messages as the one we implemented.

Moreover in contrast with [72,73,78] which are using a process calculus as input language, we propose a higher-level and more intuitive language, making our tool suitable for a larger audience. Additionally we designed a Java library for security which can be used not only in conjunction with *AnBx*, but also, in a broader context, even by programmers without a deep knowledge of the security foundations. Last but not least, having an high degree of automation, the tool is suitable for agile prototyping and rapid development of security protocols.

In the second part of the thesis (Chapter 5) we formally analyzed the Secure Vehicle Communication system developed by the EU-project SeVeCom, using the AIF framework [60] which is based on a novel set-abstraction technique. Our analysis of the SeVeCom root key update protocol has revealed two potential weaknesses. Under reasonable assumptions though, we can exclude the attacks and verify the root key update. The detection of the

[3] `http://www.dais.unive.it/~modesti/anbx/`

attacks and the verification of the fixed system take a few minutes each. We have also considered a comprehensive model of the system including all communication protocols and have verified all goals under the assumption that the intruder does not have access to the signing functionality of all HSMs. The verification of this more complex task takes less than 2 hours. The specifications are available at [60].

Our work was inspired by a similar work of Steel [77], who modeled parts of the SeVeCom-system using SATMC [8]. Here, the number of steps had to be bounded. Since he did not model the certification authority, he could not find the problems in root key update.

Our work shows that even complex systems (that require features like the revocation of keys) can be efficiently verified without bounding the number of steps that agents can perform. More generally, it shows that even the relevant aspects of time can be integrated into a model that abstracts from the traces and transitions (and thereby any notion of time): using sets for different time periods, we can integrate the time information into the abstraction.

A

Appendix

A.1 *AnBx* Case Studies Source Code

```
Protocol: Orig_3KP

Types:
  Agent C,M,a;
  Certified C,M,a;
  Number TID,Auth,empty,Desc,Price,ID,SALTM,V,VC,NONCE;
  Symmetric_key RC, SALTC;
  Function pk,sk,hash,hmac,can

Definitions:
  IDC: hmac(can(C),RC);
  Common: Price,ID,TID,NONCE,hmac(can(C),RC),hmac(Desc,SALTC),hash(V),hash(VC);
  Clear: ID,TID,NONCE,hash(Common),hash(V),hash(VC);
  Slip: Price,hash(Common),can(C),RC,SALTM;
  EncSlip:{Slip}pk(a);
  SigM: {hash(Common)}inv(sk(M));
  # Modification in "Formal Analysis of iKP" - Ogata-Futatsugi
  SigM2: {hash(Common),EncSlip}inv(sk(M));
  SigC: {hash(EncSlip,hash(Common))}inv(pk(C));
  # SigA: {hash(Auth,hash(Common))}inv(sk(a))
  # Proposed modification of iKP
  SigA: {C,M,hash(Auth,hash(Common))}inv(sk(a))

Knowledge:
  C: C,M,a,can(C);
  M: C,M,a;
  a: C,M,a;
  C,M share Price,Desc

Actions:

# 1. Initiate
  C -> M: SALTC,IDC

# 2. Invoice
  M -> C: Clear,SigM

# 3. Payment
  C -> M: EncSlip,SigC

# 4. Auth-Request
  M -> a: Clear,hmac(SALTC,Desc),EncSlip,SigM2,SigC

# 5. Auth-Response
  a -> M: Auth, SigA

# 6. Confirm
  M -> C: Auth,SigA,V,VC

Goals:
    [...]
```

Table A.1: Portion of the *AnBx* specification of the original *3KP*

```
Protocol: Revised_3KP

Types:
  Agent C,M,a;
  Certified C,M,a;
  Number TID,Desc,Price;
  SeqNumber Auth;
  Function can

Definitions:
  Contract: Price,TID,[can(C):a],[Desc:M]

Knowledge:
  C: C,M,a,can(C);
  M: C,M,a;
  a: C,M,a
  C,M share Price,Desc

Actions:

  C -> M,@(C|M|M): [can(C):a],[Desc:M]
  M -> C,@(M|C|C): TID,[Contract]
  C -> M,(C|a|a): Price,TID,can(C),[can(C):a],[Contract]
  M -> a,(C|a|a): Price,TID,can(C),[can(C):a],[Contract]
  a -> M: empty
  M -> a,@(M|a|a): Price,TID,[Desc:M],[Contract]
  a  -> M,@(a|M,C|M): Auth,TID,[Contract]
  M -> C,(a|M,C|C): Auth,TID,[Contract]

Goals:

  # credit card secrecy and authorization
  can(C) secret between C,a
  C confidentially sends can(C) to a
  a weakly authenticates C on can(C)

  # included in a stronger goal [Contract],Auth
  M authenticates a on Auth
  C authenticates a on Auth

  # global agreement
  C authenticates M on [Contract]
  a authenticates M on [Contract]
  M authenticates a on [Contract],Auth
  C authenticates a on [Contract],Auth
  a authenticates C on [Contract]
  M authenticates C on [Contract]

  # secrets
  Desc secret between C,M
  Auth secret between C,M,a
  TID secret between C,M,a
  Price secret between C,M,a
  # contract
  [Contract] secret between C,M,a
```

Table A.2: *AnBx* specification of the revised *3KP*

```
Protocol: SET_Original

# Signed Purchase Request
# G.Bella, F.Massacci and L.C.Paulson "Verifying the SET Purchase Protocols"

Types:
  Agent C,M,a;
  Certified C,M,a;
  Number PurchAmt,XID,OrderDesc,LIDM,ChallC,ChallM,AuthCode;
  Function pan
  # pan(C) abstract CardSecret,PAN,PANSecret  see AVISPA SM (AI)
Definitions:
  HOD: hash(OrderDesc,PurchAmt);
  PIHead: LIDM,XID,HOD,PurchAmt,M,hash(XID,pan(C));
  OIData: LIDM,XID,ChallC,HOD,ChallM;
  PANData: pan(C);
  PIData: PIHead,PANData;
  PIDualSign: {hash(PIData),hash(OIData)}inv(sk(C)),{PIHead,hash(OIData),PANData}pk(a);
  OIDualSign: OIData,hash(PIData)
# CompCode: PurchAmt,AuthCode,Status
Knowledge:
  C: C,M,a,pan(C);
  M: C,M,a;
  a: C,M,a;
  C,M share PurchAmt,OrderDesc
Actions:

# 1. Purchase Initialization Request
# The Cardholder sends the Merchant a freshness challenge (ChallC)
# and a local transaction identifier (LIDM).
  C -> M : LIDM,ChallC

# 2. Purchase Initialization Response
# The Merchant replies with a signed message that includes a freshness
# challenge (ChallM) and generates a nonce that serves as the globally
# unique transaction identifier XID
  M -> C : {LIDM,XID,ChallC,ChallM}inv(sk(M))

# 3. Purchase Request
# Payment Instructions PIData and the Order Information OIData
  C -> M : PIDualSign,OIDualSign

# 4. Authorization Request
  M -> a : {{LIDM,XID,hash(OIData),HOD,PIDualSign}inv(sk(M))}pk(a)

# 5. Authorization Response
  a -> M : {{M,LIDM,XID,PurchAmt,AuthCode}inv(sk(a))}pk(M)

# 6. Purchase Response
  M -> C : {LIDM,XID,ChallC,hash(PurchAmt),AuthCode}inv(sk(M))

Goals:
  [..]
```

Table A.3: Portion of the *AnBx* specification of the original *SET*

```
Protocol: SET_Revised
# Signed Purchase Request

Types:
  Agent C,M,a;
  Certified C,M,a;
  Number LIDM,XID,PurchAmt,OrderDesc;
  SeqNumber AuthCode;
  Function pan

Definitions:
  TID: LIDM,XID;
  OIdata: OrderDesc;
  PIdata: pan(C);
  HOD: [OIdata:M],[PIdata:a]

Knowledge:
  C: C,M,a,pan(C);
  M: C,M,a;
  a: C,M,a;
  C,M share PurchAmt,OrderDesc

Actions:

# 1. Purchase Initialization Request
  C -> M,@(C|M|M): LIDM
# 2. Purchase Initialization Response
  M -> C,@(M|C|C): XID
# 3. Purchase Request
  C -> M,@(C|M|M): TID,HOD
  M -> C: empty
  C -> M,(C|a|a): TID,PurchAmt,PIdata,HOD
# 4. Authorization Request
  M -> a,(C|a|a): TID,PurchAmt,PIdata,HOD
  a -> M: empty
  M-> a,@(M|a|a): TID,PurchAmt,HOD
# 5. Authorization Response
  a -> M,@(a|M,C|M): TID,HOD,AuthCode
# 6. Purchase Response
  M -> C,@(a|M,C|C): TID,HOD,AuthCode

Goals:

  # credit card secrecy and authorization
  pan(C) secret between C,a
  C confidentially sends pan(C) to a
  a weakly authenticates C on pan(C)

  # included in a stronger goal TID,HOD,AuthCode
  M authenticates a on AuthCode
  C authenticates a on AuthCode
  # extra goal forwarded auth
  C authenticates M on AuthCode

  # global agreement
  a authenticates M on TID,HOD
  a authenticates C on TID,HOD
  C authenticates a on TID,HOD,AuthCode
  C authenticates M on TID,HOD
  M authenticates C on TID,HOD
  M authenticates a on TID,HOD,AuthCode

  # secrets
  TID secret between C,M,a
  HOD secret between C,M,a
  OrderDesc secret between C,M
  PurchAmt secret between C,M,a
  AuthCode secret between C,M,a
```

Table A.4: *AnBx* specification of the revised *SET*

A.2 SeVeCom Case Studies Source Code

Listing A.1: SeVeCom RootKey Update Protocol

```
Problem: SEVECOM_RootKey;

%% Version of Sevecom where the intruder knows no root key,
%% Either key can be revoked non-deterministically.
%% ATTACK: re-introduce old keys
%% Proverif times out

Types:
A    : {hsm1,hsm2,auth,pp,i};  % agents
HSM    : {hsm1,hsm2};     % Honest HSMs
CA    : {auth};      % Certification Authority
PP    : {pp};       % Pseudonymous providers // same as CA
KeyType  : {root1,root2};
Root  : {root1,root2};
Updating : {updating,uptodate};
RootSts   : {one_root_key,two_root_keys}; % status of the HSM
Dummy  : {i};

K,K1,K2,RK,NK,PK,N,INFO : value;    % RK=RootKey  NK=NewKey  PK=PP Key N=Nonce
M,M1,M2,M3,M4,M5,M6  : untyped;

Sets:
db(HSM,KeyType,Updating), % HSM's own db
       % the flag "is_removable" is omitted because can be inferred by the KeyType
       % is_removable = shortterm keys, otherwise not is_removable
dbr(Root),     % centralized db for root keys
dishonest(Dummy),   % intruder generated keys
revoked(HSM);    % all the revoked keys

Functions:
public  sign/2, crypt/2, pair/2, h/1;
private inv/1;

Facts:
iknows/1, attack/0, candidate/2, secureCh/3;

Rules:
% Intruder deduction:
\A. => iknows(A);

iknows(sign(M1,M2)) => iknows(h(M2));
iknows(M1).iknows(M2) => iknows(sign(M1,M2));

iknows(crypt(M1,M2)).iknows(inv(M1)) => iknows(M2);
iknows(M1).iknows(M2) => iknows(crypt(M1,M2));

% pair
iknows(pair(M1,M2)) => iknows(M1).iknows(M2);
iknows(M1).iknows(M2) => iknows(pair(M1,M2));
% hash
iknows(M) => iknows(h(M));

=[K]=> iknows(K).iknows(inv(K)).K in dishonest(i);

% ---------------------------- ROOT KEYS ----------------------------
% hsm root keys UPDATE

% API: initDevice
% root keys

\HSM. =[K1,K2]=> iknows(K1).iknows(K2).
         K1 in dbr(root1).K2 in dbr(root2).
```

```
         K1 in db(HSM,root1,uptodate).K2 in db(HSM,root2,uptodate);

%% Pseudonymous provider key generation
%% we assume that the Cars know the PP public key
%\HSM. =[K]=> K in db(HSM,ppsig,uptodate).iknows(K);

% Revoke root keys
% API: revokeRootKey
\HSM,Root.iknows(sign(inv(K),K)).K in db(HSM,Root,uptodate)
=>
iknows(K).K in db(HSM,Root,updating);

% Update a root key1
% API: setRootKey
\HSM.iknows(sign(inv(K2),K)).
K2 in db(HSM,root2,uptodate).
K1 in db(HSM,root1,updating).
=>
K2 in db(HSM,root2,uptodate).
K  in db(HSM,root1,uptodate).
K1 in revoked(HSM);
% K is now valid, K1 is no longer in db

% Update a root key2
\HSM.iknows(sign(inv(K1),K)).
K1 in db(HSM,root1,uptodate).
K2 in db(HSM,root2,updating).
=>
K1 in db(HSM,root1,uptodate).
K  in db(HSM,root2,uptodate).
K2 in revoked(HSM);
% K is now valid, K2 is no longer in db

%%% CA revokes root1-key

\HSM.K1 in dbr(root1).
K2 in dbr(root2).
iknows(K1).iknows(K2)
=[K]=>
K1 in dbr(root1).
K in dbr(root2).
iknows(sign(inv(K1),K1)).
iknows(sign(inv(K2),K));

%%% CA revokes root2-key

\HSM.K1 in dbr(root1).
K2 in dbr(root2).
iknows(K1).iknows(K2)
=[K]=>
K in dbr(root1).
K2 in dbr(root2).
iknows(sign(inv(K2),K2)).
iknows(sign(inv(K1),K));

% ------------------------ Attacks on Root keys ------------------------


%%%% Attacks on Root keys

% The intruder gets hold of a private key
\HSM,Root,Updating.
K in db(HSM,Root,Updating).
iknows(inv(K))
=>
attack;
```

```
% The intruder has inserted one of his keys:
\HSM,Root,Updating.
K in db(HSM,Root,Updating).
K in dishonest(i).
=>
attack;

% The intruder can make the TRD accept an old key:
\HSM,Root,Updating.
K in revoked(HSM).
K in db(HSM,Root,Updating).
=>
attack;

% ------------------------------------------------------------------------------
```

Listing A.2: SeVeCom Comprehensive Timed Model

```
Problem: SEVECOM;

% SeVeCom comprehensive model - with Timing and abstract channel for short-term key
    update
% RESULT goal unreachable: attack: (SAFE)

% This variant models the root, long and short term keys with several vehicles

Types:
A    : {hsm1,hsm2,auth,pp,i};       % agents
HSM    : {hsm1,hsm2};          % Honest HSMs
CA    : {auth};           % Certification Authority
PP    : {pp};            % Pseudonymous providers
                % can be the same as CA if PP:{auth}
KeyType   : {root1,root2,ltsig,ltdec,stsig,stdec,ppsig};  % root,longterm,shortterm,ps.
    prov
Root   : {root1,root2};
LongTerm : {ltsig,ltdec};
ShortTerm : {stsig,stdec};
Updating : {updating,uptodate};
RootSts    : {one_root_key,two_root_keys};      % status of the HSM
NoncesSts   : {challenged,responded};
Dummy   : {i};
Time   : {old,expsoon,new};

K,K1,K2,RK,NK,PK,N,INFO : value;        % RK=RootKey  NK=NewKey  PK=PP Key N=Nonce
M,M1,M2,M3,M4,M5,M6   : untyped;

Sets:
db(HSM,KeyType,Updating),  % HSM's own db
        % the flag "is_removable" is omitted because can be inferred by the KeyType
        % is_removable = shortterm keys, otherwise not is_removable
dbr(Root),      % centralized db for root keys
dbca(HSM,LongTerm),    % centralized db for all CAs,
dbps(HSM,ShortTerm),   % centralized db for pseudos
dishonest(Dummy),    % intruder generated keys
revoked(HSM),     % all the revoked keys
nonces(PP,HSM,NoncesSts),  % nonces exchanged during psudonymous update
inonces(CA,HSM),    % INFO nonces exchanged during longterm update
timer(Time);     % timer for time progress

Functions:
public  sign/2, crypt/2, pair/2, h/1;
private inv/1;

Facts:
iknows/1, attack/0, candidate/2, secureCh/3;
```

```
Definitions:

% Used for LongTerm key update and certificates
trip[x1,x2,x3] : pair([x1],pair([x2],[x3]));
quad[x1,x2,x3,x4] : pair([x1],trip([x2],[x3],[x4]));
sig[sk,x,pkx,t]: sign(inv([sk]),trip([x],[pkx],[t]));       % pkx public key of the
    HSM, x its identity, inv[sk] the private key of the TTP, t is the TTP identity
cert[x,pkx,sk,t]: sig([sk],[x],[pkx],[t]);          % certificate created by TTP t
crdata[x,npkx,t,info]: quad([x],[t],[npkx],[info]);       % npkx is the new vehicle
    public key
cr[x,npkx,skx,t,info]: sign(inv([skx]),crdata([x],[npkx],[t],[info]));  % inv[skx] old
     private key of the vehicle
cack[x,skx,t,info]: sign(inv([skx]),trip([t],[x],[info]))      % acknoledge (longterm)

Rules:
% Intruder deduction:
\A. => iknows(A);

iknows(sign(M1,M2)) => iknows(h(M2));
iknows(M1).iknows(M2) => iknows(sign(M1,M2));

iknows(crypt(M1,M2)).iknows(inv(M1)) => iknows(M2);
iknows(M1).iknows(M2) => iknows(crypt(M1,M2));

% pair
iknows(pair(M1,M2)) => iknows(M1).iknows(M2);
iknows(M1).iknows(M2) => iknows(pair(M1,M2));

% hash
iknows(M) => iknows(h(M));

% the intruder can generate arbitrary key-pairs (that are not
% part of the database --- unless he manages to insert them).
=[K]=> iknows(K).iknows(inv(K)).K in dishonest(i);

% Decrypt with API
\HSM,Updating. iknows(crypt(K,M)).K in db(HSM,ltdec,Updating) => iknows(M).K in db(HSM
    ,ltdec,Updating);
\HSM,Updating. iknows(crypt(K,M)).K in db(HSM,stdec,Updating) => iknows(M).K in db(HSM
    ,stdec,Updating);

% Sign with API
% long-term signature function is disable for the intruder
%\HSM,Updating.iknows(M).K in db(HSM,ltsig,Updating) => iknows(sign(inv(K),M)).K in db
    (HSM,ltsig,Updating);
\HSM,Updating.iknows(M).K in db(HSM,stsig,Updating) => iknows(sign(inv(K),M)).K in db(
    HSM,stsig,Updating);

% Initialization: initial keys; our model allows an arbitrary number
% of them, and obviously an implementation that limits this number is
% a special case.

% ---------------------------- ROOT KEYS --------------------------
% hsm root keys UPDATE

% API: initDevice
% root keys

\HSM. =[K1,K2]=> iknows(K1).iknows(K2).
         K1 in dbr(root1).K2 in dbr(root2).
         K1 in db(HSM,root1,uptodate).K2 in db(HSM,root2,uptodate);

% Revoke root keys
% API: revokeRootKey
\HSM,Root.iknows(sign(inv(K),K)).K in db(HSM,Root,uptodate)
=>
```

```
iknows(K).K in db(HSM,Root,updating);

% Update a root key1
% API: setRootKey
\HSM.iknows(sign(inv(K2),K)).
K2 in db(HSM,root2,uptodate).
K1 in db(HSM,root1,updating).
=>
K2 in db(HSM,root2,uptodate).
K  in db(HSM,root1,uptodate).
K1 in revoked(HSM);
% K is now valid, K1 is no longer in db

% Update a root key2
\HSM.iknows(sign(inv(K1),K)).
K1 in db(HSM,root1,uptodate).
K2 in db(HSM,root2,updating).
=>
K1 in db(HSM,root1,uptodate).
K  in db(HSM,root2,uptodate).
K2 in revoked(HSM);
% K is now valid, K2 is no longer in db

\HSM.K1 in dbr(root1).K2 in dbr(root2).
iknows(K1).iknows(K2)
=[K]=>
K1 in dbr(root1).
K in dbr(root2).
iknows(sign(inv(K1),K1)).
iknows(sign(inv(K2),K));

\HSM.K1 in dbr(root1).K2 in dbr(root2).
iknows(K1).iknows(K2)
=[K]=>
K in dbr(root1).
K2 in dbr(root2).
iknows(sign(inv(K1),K1)).
iknows(sign(inv(K2),K));

% ------------------------- Attacks on Root keys -------------------------

% The intruder gets hold of a private key
\HSM,Root,Updating.
K in db(HSM,Root,Updating).
iknows(inv(K))
=>
attack;

% The intruder has inserted one of his keys:
\HSM,Root,Updating.
K in db(HSM,Root,Updating).
K in dishonest(i).
=>
attack;

% The intruder can make the TRD accept an old key:
\HSM,Root,Updating.
K in revoked(HSM).
K in db(HSM,Root,Updating).
=>
attack;

% ------------------------------------------------------------------------

% --------------------------- LONG TERM KEYS ---------------------------
```

```
% time progress

=[K]=> K in timer(new);
K in timer(expsoon) => K in timer(old);

% Initialization:
% The TTP CA creates the certificates
% we assume that the rootkeys (RK) are known to the car's HSM and that they can be
    used by CA to sign messages
% this is also the way longterm keys are initialized
% we assume a trustworthy communication link

\HSM,CA,Root,LongTerm. RK in dbr(Root)
=[K]=>
K in db(HSM,LongTerm,uptodate).K in dbca(HSM,LongTerm).RK in dbr(Root).iknows(K).
    iknows(cert[HSM,K,RK,CA]).K in timer(expsoon);

% update of the longterm keys
% see 3.3.4
% V(x) -> T
% the vehicle requests a new certificate and provide a new key NK in a self signed
    message
% API: initLongTermKeyUpdate

\HSM,CA,LongTerm. K in timer(expsoon). K in db(HSM,LongTerm,uptodate)
=[ NK,INFO ]=>
K in timer(expsoon).NK in timer(new).iknows(cr[HSM,NK,K,CA,INFO]).K in db(HSM,LongTerm
    ,updating).iknows(NK).NK in db(HSM,LongTerm,updating);

% CA returns the new certificate
% we assume that the new certificate is valid instantaneously
% in reality it has a validity period (as the old one)
% "The presence of the HSM ensures that the vehicle will not utilize the newly
    acquired
% certificate and the corresponding private key during the validity period E of its
    current
% key and certificate"

% CA receives a request
\CA,HSM,Root,LongTerm. iknows(cr[HSM,NK,K,CA,INFO]).K in dbca(HSM,LongTerm).RK in dbr(
    Root).INFO notin inonces(CA,HSM).K in timer(expsoon).NK in timer(new)
=>
% and issue the certificate
% API: finalizeLongTermKeyUpdate
iknows(cert[HSM,NK,RK,CA]).K in dbca(HSM,LongTerm).RK in dbr(Root).INFO in inonces(CA,
    HSM).K in timer(expsoon).NK in timer(new);

% HSM receives the certificate set new LongTermKey NK and revokes K
% and Notification
% V(x)->T

\CA,HSM,Root,LongTerm. iknows(cert[HSM,NK,RK,CA]).iknows(cr[HSM,NK,K,CA,INFO]).RK in
    db(HSM,Root,uptodate).K in db(HSM,LongTerm,updating).NK in db(HSM,LongTerm,
    updating)
=>
NK in db(HSM,LongTerm,uptodate).RK in db(HSM,Root,uptodate).iknows(cack[HSM,NK,CA,INFO
    ]).K in db(HSM,LongTerm,updating);

% CA receives the notification and puts NK in dbca
\HSM,CA,LongTerm. iknows(cack[HSM,NK,CA,INFO]). iknows(cr[HSM,NK,K,CA,INFO]).iknows(
    cert[HSM,NK,RK,CA]).K in dbca(HSM,LongTerm).K in timer(expsoon).NK in timer(new).
    INFO in inonces(CA,HSM)
=>
K in dbca(HSM,LongTerm).NK in dbca(HSM,LongTerm).K in timer(expsoon).NK in timer(new).
    INFO in inonces(CA,HSM);
```

```
% revocation of K
\CA,HSM,Root,LongTerm.RK in db(HSM,Root,uptodate).
K in dbca(HSM,LongTerm).K in db(HSM,LongTerm,updating).
K in timer(old).NK in timer(new).INFO in inonces(CA,HSM).
NK in dbca(HSM,LongTerm). NK in db(HSM,LongTerm,uptodate). iknows(cack[HSM,NK,CA,INFO
    ]). iknows(cr[HSM,NK,K,CA,INFO]). iknows(cert[HSM,NK,RK,CA])
=>
% K is then REVOKED
K in revoked(HSM).K in timer(old).NK in timer(expsoon).NK in dbca(HSM,LongTerm). NK in
     db(HSM,LongTerm,uptodate).RK in db(HSM,Root,uptodate).INFO in inonces(CA,HSM);

% ----------------- Attacks on Long Term Keys -----------------------------

% The intruder has inserted one of his "longterm" keys:
\HSM,LongTerm,Updating.K in db(HSM,LongTerm,Updating).K in dishonest(i) => attack;
\HSM,LongTerm,Updating.K in dbca(HSM,LongTerm).K in dishonest(i) => attack;

% The intruder can make the TRD accept an old longterm key:
% in order to avoid attacks the constrain on time must be set on the rules
% .K in timer(expsoon).NK in timer(new) in LS and RS of the rule
\HSM,LongTerm.K in revoked(HSM).K in dbca(HSM,LongTerm) => attack;
\HSM,LongTerm,Updating.K in revoked(HSM).K in db(HSM,LongTerm,Updating) => attack;
\HSM,LongTerm,Updating.K in timer(old).K in revoked(HSM).K in dbca(HSM,LongTerm).K in
    db(HSM,LongTerm,Updating) => attack;

% The intruder knows any private longterm K
\HSM,LongTerm,Updating.K in db(HSM,LongTerm,Updating).iknows(inv(K)) => attack;
\HSM,LongTerm.K in dbca(HSM,LongTerm).iknows(inv(K)) => attack;


% Authentication Goal - the CA does not consider valid a key which is not in the HSM
    db
\HSM,LongTerm.K notin dbca(HSM,LongTerm).K in db(HSM,LongTerm,uptodate).K in timer(
    expsoon) => attack;
\HSM,LongTerm.K in dbca(HSM,LongTerm). forall Updating. K notin db(HSM,LongTerm,
    Updating) => attack;

% ------------------------------------------------------------------------------

% ---------------------------- SHORT TERM KEYS - PSEUDONYMS -------------------------

% see 4.2.4
% short term keys init and generation
% API: initDevice
% API: generateKeyPair

% ShortTerm keys are generated and processed in a batch,
% here we generate and process one by one

% The communication with the PP has to be done via an authenticated and confidential
    communication link
% secureCh/3;

% Pseudonym provider key generation
% we assume that the Cars know the PP public key
\HSM. =[K]=> K in db(HSM,ppsig,uptodate).iknows(K);

% X-> PP: PSNYM-PKX
\HSM,PP,ShortTerm. =[K]=> K in db(HSM,ShortTerm,updating).secureCh(HSM,PP,K);

% PP->X: N
\PP,HSM.secureCh(HSM,PP,K) =[N]=> N in nonces(PP,HSM,challenged).secureCh(PP,HSM,N);

\HSM,PP,ShortTerm. K in db(HSM,ShortTerm,updating).secureCh(PP,HSM,N)
=>
secureCh(HSM,PP,sign(inv(K),N)).K in db(HSM,ShortTerm,updating);
```

```
% PP->X: cert(X)
\HSM,PP,ShortTerm.secureCh(HSM,PP,sign(inv(K),N)).N in nonces(PP,HSM,challenged).PK in
     db(HSM,ppsig,uptodate)
=>
secureCh(PP,HSM,cert[HSM,K,PK,PP]).K in dbps(HSM,ShortTerm).N in nonces(PP,HSM,
    responded).PK in db(HSM,ppsig,uptodate);

\HSM,PP,ShortTerm.secureCh(PP,HSM,cert[HSM,K,PK,PP]).K in db(HSM,ShortTerm,updating)
=>
% at the end of the subprotocol the intruder knows the certificate)
K in db(HSM,ShortTerm,uptodate).iknows(cert[HSM,K,PK,PP]);

% ------------------- ShortTerm Key Attacks ---------------------------------

% The intruder knows any private shortterm K
\HSM,ShortTerm,Updating.K in db(HSM,ShortTerm,Updating).iknows(inv(K)) => attack;
\HSM,ShortTerm.K in dbps(HSM,ShortTerm).iknows(inv(K)) => attack;

% The intruder has inserted one of his "pseudonyms":
\HSM,ShortTerm,Updating.K in db(HSM,ShortTerm,Updating).K in dishonest(i) => attack;

% Safe
\HSM,ShortTerm,PP.K in dbps(HSM,ShortTerm).iknows(cert[HSM,K,PK,PP]).iknows(inv(K)) =>
     attack;
\HSM,ShortTerm,Updating,PP.K in db(HSM,ShortTerm,Updating).iknows(cert[HSM,K,PK,PP]).
    iknows(inv(K)) => attack;
```

Bibliography

[1] M. Abadi. Security protocols and their properties. *NATO ASI Series F Computer and Systems Sciences*, 175:39–60, 2000.

[2] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–315. ACM New York, NY, USA, 2000.

[3] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47. ACM, 1997.

[4] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, 1994. Proceedings.*, pages 122–136, 1994.

[5] P. Adao and C. Fournet. Cryptographically sound implementations for communicating processes. *Lecture Notes in Computer Science*, 4052:83–94, 2006.

[6] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Drielsma, P. Héam, O. Kouchnarenko, J. Mantovani, et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV*, volume 5, pages 281–285. Springer, 2005.

[7] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. In *20th IEEE Computer Security Foundations Symposium (CSF07)*, pages 385–396, 2007.

[8] A. Armando and L. Compagna. SATMC: A SAT-based model checker for security protocols. *Lecture Notes in Computer Science*, pages 730–733, 2004.

[9] M. Avalle, A. Pironti, R. Sisto, and D. Pozza. The JavaSPI framework for security protocol implementation. In *Availability, Reliability and Security (ARES 11)*, page 746–751. IEEE Computer Society, IEEE Computer Society, 2011.

[10] AVISPA. Deliverable 2.3: The Intermediate Format. Available at `www.avispa-project.org`, 2003.

[11] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[12] G. Bella, F. Massacci, and L. Paulson. Verifying the SET registration protocols. *IEEE Journal on Selected Areas in Communications*, 21(1):77–87, 2003.

[13] G. Bella, F. Massacci, and L. Paulson. An overview of the verification of SET. *International Journal of Information Security*, 4(1):17–28, 2005.

[14] G. Bella, F. Massacci, and L. Paulson. Verifying the SET purchase protocols. *Journal of Automated Reasoning*, 36(1):5–37, 2006.

[15] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. Van Herreweghen, and M. Waidner. Design, implementation, and deployment of the iKP secure electronic payment system. *IEEE Journal on selected areas in communications*, 18(4):611–627, 2000.

[16] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. iKP a family of secure electronic payment protocols. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, 1995.

[17] K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF 2009*, 2009.

[18] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *FMCO*, pages 197–222, 2003.

[19] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01*, pages 82–96. IEEE Computer Society Press, 2001.

[20] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 126–140. IEEE, 2003.

[21] Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *AVIS'04*, pages 1–11, 2004.

[22] J. Borgström, S. Briais, and U. Nestmann. Symbolic bisimulation in the spi calculus. *CONCUR 2004-Concurrency Theory*, pages 161–176, 2004.

[23] L. Bozga, Y. Lakhnech, and M. Perin. Pattern-based abstraction for verifying secrecy in protocols. *Int. J. on Software Tools for Technology Transfer*, 8(1):57–76, 2006.

[24] S. Briais. *Theory and tool support for the formal verification of cryptographic protocols.* PhD thesis, EPFL, Switzerland, 2008.

[25] S. Briais and U. Nestmann. A formal semantics for protocol narrations. *Theor. Comput. Sci.*, 389:484–511, December 2007.

[26] S. Brlek, S. Hamadou, and J. Mullins. A flaw in the electronic commerce protocol SET. *Information Processing Letters*, 97(3):104–108, 2006.

[27] M. Bugliesi, S. Calzavara, S. Mödersheim, and P. Modesti. AnBx-Enhanced Alice and Bob notation for security protocols design and analysis. *In preparation*, 2012.

[28] M. Bugliesi and R. Focardi. Language based secure communication. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 3–16, 2008.

[29] M. Bugliesi and P. Modesti. AnBx-Security protocols design and verification. In *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security: Joint Workshop, ARSPA-WITS 2010, Paphos, Cyprus, 2010, Revised Selected Papers*, pages 164–184. Springer-Verlag, 2010.

[30] U. Carlsen. Generating formal cryptographic protocol specifications. *IEEE Symposium on Research in Security and Privacy*, pages 137–146, 1994.

[31] S. Ciobâcă and V. Cortier. Protocol composition for arbitrary primitives. In *Proceedings of CSF'10*, 2010.

[32] S. Clover. HStringTemplate. `www.haskell.org/haskellwiki/HStringTemplate`.

[33] R. Corin, P.-M. Dénielou, C. Fournet, K. Bhargavan, and J. J. Leifer. Secure implementations of typed session abstractions. In *CSF 2007*, pages 170–186. IEEE, 2007.

[34] C. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer, 2008.

[35] CWE/SANS. CWE/SANS Top 25 most dangerous software errors. `http://www.sans.org/top25-software-errors/`, 2011.

[36] G. Denker and J. Millen. CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory, 1999.

[37] G. Denker, J. Millen, and H. Rueß. The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, 2000.

[38] T. Dierks and C. Allen. RFC2246: The TLS protocol version 1.0. *Internet RFCs*, 1999.

[39] C. Dilloway and G. Lowe. On the specification of secure channels. In *WITS '07*, 2007.

[40] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Inform. Theory*, 2(29), 1983.

[41] N. Ferguson and B. Schneier. *Practical cryptography*, volume 141. Wiley New York, 2003.

[42] E. Foundation. Eclipse IDE. `http://www.eclipse.org`.

[43] H. Gao, F. Nielson, and H. R. Nielson. Protocol stacks for services. In *Proc. of the Workshop on Foundations of Computer Security (FCS)*, July 2009.

[44] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*. Addison-Wesley, 2003.

[45] J. Guttman. Security protocol design via authentication tests. In *In Proceedings of 15th IEEE Computer Security Foundations Workshop. IEEE Computer*, 2002.

[46] IBM. IBM X-Force 2009 mid-year report. `www.ibm.com`, 2009.

[47] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *LPAR 2000*, LNCS 1955, pages 131–160, 2000.

[48] M. Jakobsson, K. Sako, and R. Impagliazzo. Designated verifier proofs and their applications. In *Lecture Notes in Computer Science 1070*, pages 143–158, 1996.

[49] P. Jarupunphol and C. Mitchell. Measuring 3-D Secure and 3D SET against e-commerce end-user requirements. In *Proceedings of the 8th Collaborative Electronic Commerce Technology and Research Conference*, pages 51–64, 2007.

[50] A. Kamil and G. Lowe. Specifying and modelling secure channels in strand spaces. In *Proceedings of the Worskshop on Formal Aspects of Security and Trust (FAST 2009)*, 2009.

[51] G. Lowe. A hierarchy of authentication specifications. In *CSFW'97*, pages 31–43. IEEE Computer Society Press, 1997.

[52] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[53] LSV. Security protocols open repository (spore). `www.lsv.ens-cachan.fr/spore/`.

[54] S. Lu and S. Smolka. Model checking the secure electronic transaction (set) protocol. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1999. Proceedings. 7th International Symposium on*, pages 358–364, 1999.

[55] U. M. Maurer and P. E. Schmid. A calculus for secure channel establishment in open networks. In *ESORICS*, pages 175–192, 1994.

[56] C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Financial Cryptography*, page 122. Springer, 1998.

[57] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC, 1997.

[58] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[59] S. Mödersheim. Algebraic properties in Alice and Bob notation. In *International Conference on Availability, Reliability and Security (ARES 2009)*, pages 433–440, 2009.

[60] S. Mödersheim. Abstraction by Set-Membership—verifying security protocols and web services with databases. In *CCS 2010*. ACM, 2010. Implementation and examples, including SeVeCom specifications, available on `www.imm.dtu.dk/~samo`.

[61] S. Mödersheim and P. Modesti. Verifying sevecom using set-based abstraction. In *7th International Wireless Communications and Mobile Computing Conference (IWCMC), Istanbul, Turkey, 2011*, pages 1164–1169. IEEE, 2011.

[62] S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In *Foundations of Security Analysis and Design V*, page 194. Springer-Verlag, 2009.

[63] S. Mödersheim and L. Viganò. Secure pseudonymous channels. In M. Backes and P. Ning, editors, *Computer Security – ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 337–354. Springer Berlin / Heidelberg, 2009.

[64] K. Ogata and K. Futatsugi. Formal analysis of the iKP electronic payment protocols. *Lecture Notes in Computer Science*, pages 441–460, 2003.

[65] D. O'Mahony, M. Peirce, and H. Tewari. *Electronic payment systems for e-commerce*. Artech House Publishers, 2001.

[66] P. Papadimitratos, L. Buttyan, T. Holczer, E. Schoch, J. Freudiger, M. Raya, Z. Ma, F. Kargl, A. Kung, and J. Hubaux. Secure vehicular communication systems: design and architecture. *Communications Magazine, IEEE*, 46(11):100–109, 2008.

[67] T. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. ACM, 2004.

[68] T. Parr. A functional language for generating structured text. 2006. `www.cs.usfca.edu/~parrt/papers/ST.pdf`.

[69] T. Parr. Web application internationalization and localization in action. In *Proceedings of the 6th international conference on Web engineering*, pages 64–70. ACM, 2006.

[70] B. Pierce. *Types and programming languages*. The MIT Press, 2002.

[71] M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java 2 Security: Building Secure and Robust J2EE Applications*. Addison Wesley, 2004.

[72] D. Pozza, R. Sisto, and L. Durante. Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications-Volume 2*, page 400. IEEE Computer Society, 2004.

[73] J. N. Quaresma and C. W. Probst. Protocol implementation generator. In *Proceedings of The 15th Nordic Conference on Secure IT Systems*, 2010.

[74] B. Schneier and P. Sutherland. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.

[75] SEVECOM. Deliverable 2.1-App.A: Baseline security specifications. `www.sevecom.org`, 2009.

[76] C. Sprenger and D. Basin. Developing security protocols by refinement. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM Press, 2010.

[77] G. Steel. Towards a formal security analysis of the Sevecom API. ESCAR, 2009.

[78] B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. *Certification and Security in Inter-Organizational E-Service*, pages 33–54, 2005.

[79] E. Van Herreweghen. Non-repudiation in SET: Open issues. *Lecture Notes in Computer Science*, pages 140–156, 2001.

[80] Visa. Visa 3-D Secure specifications. Technical report, 2002.

[81] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spass version 3.0. In *CADE*, pages 514–520, 2007.