POLITECNICO DI TORINO

SCUOLA DI DOTTORATO
PhD in Ingegneria Informatica e dei Sistemi – XXII cycle

PhD Dissertation

Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications

Alfredo Pironti

Supervisor
prof. Riccardo Sisto

PhD Coordinator
prof. Pietro Laface

2010

Contents

Summary IV

1 Introduction 1

I Formal Model Driven Development of Security Protocol Implementations 4

2 Methodology 5

2.1 The Spi Calculus . . . 6
2.2 A Reference Example . . . 8
2.3 Development Methodology and Tool Support . . . 13
2.3.1 Writing and Verifying the Formal Model . . . 13
2.3.2 Refining the Formal Model . . . 15
2.3.3 Implementing the Java Application . . . 21
2.4 Experimental Results . . . 26
2.5 Related Work . . . 27
2.6 Discussion . . . 30

3 Specific Issues 31

3.1 Translation . . . 32
3.1.1 Formalizing the Translation . . . 33
3.1.2 Soundness . . . 43
3.1.3 Discussion . . . 54
3.2 Encodings . . . 56
3.2.1 Abstract Protocol Models and Notation . . . 57
3.2.2 Handling the Channel Encoding/Decoding Layer . . . 61
3.2.3 Handling the Encoding of Data to be Ciphered and Key Material . . 70
3.2.4 Applications and Examples . . . 77
3.2.5 Discussion . . . 86

4 An SSH Transport Layer Protocol Client Example 90

4.1 The SSH-TLP Formal Model . . . 90
4.2 Formal Verification of the SSH-TLP Model . . . 97

4.2.1 Checking and Debugging the Model . . . 100
4.3 Client-Side Model Refinement and Implementation Generation . . . 101
4.4 Experimental Results . . . 103
4.4.1 Interoperability and Reliability . . . 103
4.4.2 Performances . . . 105
4.5 Discussion . . . 107

II Formally Verifiable Monitoring of Legacy Implementations 109

5 Methodology 110
5.1 Notation and Network Model . . . 112
5.2 The Monitor Generation Function . . . 115
5.3 Related Work . . . 122

6 An SSL Server Monitor Example 124
6.1 Monitor Specification . . . 124
6.2 Monitor Implementation . . . 131
6.3 Experimental Results . . . 132
6.3.1 The OpenSSL Security Flaw . . . 133

Conclusion 135

A Proofs for Section 3.1 138
A.1 Proof of Theorem 1 . . . 138
A.2 Proof of Theorem 2 . . . 141
A.3 Proof of Correctness of the Pair Class . . . 147
A.3.1 GetLeft() and GetRight() Methods . . . 147
A.3.2 Equals() Method . . . 148

B Proofs for Section 3.2 153
B.1 Proof of Theorem 3 . . . 153
B.2 Proof of Theorem 5 . . . 159

C Weaker Sufficient Conditions for Fault-Preserving Renaming Transformations for Security Protocols 167
C.1 Fault Preserving Renaming Transformations . . . 168
C.2 Weakening the Original Sufficient Conditions . . . 171
C.3 Using the Weakened Condition . . . 174

Bibliography 177

Summary

This work proposes a model-driven development (MDD) approach to semi-automatically generate sound implementations of security protocols from verified formal specifications. By following the proposed methodology, the generated implementation has the property of being a sound refinement of the original formal model, thus sharing all safety properties that can be proven on the model. Many classical security properties, such as secrecy and authentication, are safety properties indeed.

Thanks to the MDD approach, the developer first concentrates on designing the protocol logic, and then on lower-level implementation details. Application design and development is thus modular, and each implementation aspect is dealt with independently during a dedicated refinement step.

All refinement steps are proven sound, so that no security faults can be introduced during implementation refinement. A set of automated tools assists and guides the developer in the development workflow, so that developer interaction is minimized, and rapid prototyping is possible.

Interoperability details, such as data marshaling and run-time cryptographic algorithms negotiation, are taken into account within the development process. So, the final generated application is able to adhere to already existing standards, and to interoperate with third-party implementations of the same security protocol.

The proposed methodology is validated by implementing a fully featured client of the SSH Transport Layer Protocol. The generated client can fully handle per-session algorithms negotiation, and can cope with error conditions by sending appropriate error messages to the other party. Moreover, the client is able to interact with a key store in order to store and retrieve trusted keys of remote servers. The starting formal model from which the implementation is generated has been formally verified for secrecy and authentication against a Dolev-Yao attacker. This implies that the generated client implementation is resilient to Dolev-Yao secrecy and authentication attacks too.

Sometimes, an untrusted legacy implementation of a security protocol actor is in place, and due to high switching costs or corporate policies it cannot be substituted by a newly generated, formally verifiable one. In these cases, this work proposes a monitoring approach, where a formally verifiable monitor is developed and coupled with the legacy application, in order to check its behavior. The MDD methodology is leveraged in the development of the monitor, which is then responsible of enforcing the desired security properties on the legacy monitored implementation.

This approach has been validated too, by the development of a monitor for the SSL

protocol. The monitor correctly stops malicious sessions, while letting correct ones pass-through. The introduced overhead is usually lower than typical network times, and under 20% on the average session time when monitoring is not enabled. Moreover, the monitor has been coupled with a flawed version of an OpenSSL client; the monitor correctly stopped the protocol runs otherwise allowed by the faulty OpenSSL client. Furthermore, thanks to the hints given by the monitor, a new flaw has been discovered in JESSIE, an open source SSL client implementation.

Chapter 1

Introduction

Security protocols are communication protocols that make use of cryptography to reach some security goals over a untrusted network. Such security protocols are often concise and neat, yet they hide subtle behaviors that are difficult to analyze and handle. The interactions between different cryptographic primitives, and the presence of a malicious attacker are often hidden by the simplicity of abstract models.

Many times, security protocols that have been considered secure for years have been discovered flawed only after they were widely deployed [46]. Moreover, newly designed protocols still happen to be affected by the same flaws already exploited in previously designed protocols [26].

Best-practice engineering recommendations can help in improving overall quality of security protocol design [4, 8, 75], however they cannot rule out all possible flaws *a priori*, not even when only focusing on high level issues, where cryptographic primitives are assumed to behave ideally. Also testing techniques or code reviews can loose their effectiveness against security protocols, due to the usually unbounded number of scenarios and the presence of the attacker, that can be hardly completely simulated in test sessions.

For a motivated attacker, even a single flaw offered by a security protocol can be enough to exploit it, gaining access to those assets meant to be protected by the security protocol itself.

Rigorous proofs of correctness of security protocols with respect to their intended security properties can help tackling this problem, and formal methods represent a way to achieve this. Any formal proof starts from some assumptions, and based on a model of the problem, assesses a fact that holds in the model, under the given assumptions. Usually, in security protocols, some assumptions are made on the capabilities of cryptographic primitives, and the attacker is modeled as a network agent that can actively interact with other protocol participants. The more abstract are the assumptions on the cryptographic primitives and the model, the simpler are the proofs of correctness. However, low level details that are abstracted away could still be responsible for security flaws.

Dolev and Yao [27] proposed an algebraic view on cryptographic primitives. Essentially, a perfect cryptography model is assumed, where cryptographic primitives are algebraic operators that apply over abstract data, called terms. For example, given the terms M,k,

the $\{M\}_k$ term represents the encryption of M under the key k, while the $H(M)$ term represents the result of hashing M. The decryption is then represented by the algebraic property stating that a plaintext can be recovered from the ciphertext only if the encryption key (and the ciphertext) are available. All protocol participants and the attacker have access to the algebraic version of the cryptographic primitives. In particular, this gives a reasonable power to the attacker, which is able to decrypt exchanged messages (if and only if it has the required keys) and to create new cryptographic terms to possibly break the protocol. Moreover, for each message it reads, the attacker increases its knowledge, which is then used to forge new messages or to alter existing ones in order to break the protocol. No other assumptions are made on the attacker, so that any behavior (including the worst one) is considered during formal verification.

This approach proved useful, enabling security related proofs to be carried out, even automatically to some extent. So, some research effort was put into formal verification of Dolev-Yao abstract models [71, 22, 28, 48, 49]. Although effective, the Dolev-Yao approach is quite abstract, and in practice protocol attacks based on lower-level design issues are commonly found. For example, bad interactions between concrete cryptographic primitives have been exploited [67, 29, 7]; or aspects that are completely abstracted away in the model have been exploited, such as timing [73]. For this reason, a branch of research focused on refining Dolev-Yao models towards computationally sound models, where computational complexity of cryptographic primitives is taken into account [5]. Automatic tools for verification of computationally sound models have appeared only recently (e.g. [25]), due to the novelty of this research branch.

However, another issue stems from the fact that in practice executable implementations of security protocols are deployed, implemented in many different programming languages. Very often, such implementations are derived directly from some informal security protocol description, and they have no relationship with any abstract verified formal model. This means that such implementations can suffer both from the intrinsic anomalies of the used programming language (e.g. buffer or integer overflows for C-like languages), and from logical errors that are not present in the protocol design, but are introduced during coding (e.g. missing a prescribed check on a received nonce, thus enabling replay attacks).

The main goal of this work is to establish or enforce a soundness relation between verified formal abstract models of security protocols and their concrete implementations. When this soundness relation holds, the implementation behaves as specified by its linked abstract model. Being the model verified, it means that no logical attacks are possible on the protocol, and thus on the application implementing it. On such applications linked with their verified formal models, flaws due to the intrinsic low-level anomalies of programming languages will still be fully possible; at least, such applications are guaranteed error-free from a logical point of view, up to the detail level captured by the linked formal model.

Here, Dolev-Yao models are considered, meaning that any security protocol implementation linked with a verified model will be resilient to Dolev-Yao attacks. Other flaws such as for example cryptanalytic attacks, or timing attacks, which are not captured by Dolev-Yao models will still be possible, both at the design and implementation levels.

A model-driven-development (MDD) approach is proposed in this work. Briefly, the

developer starts from a formally verified model of a security protocol and, by iterative refinement steps, soundly adds all those low-level details that are not captured by the formal model, until a soundly refined executable implementation of the starting model is obtained. By always applying sound refinement iterations, the final application is guaranteed to behave like the original model, thus sharing all security properties that have been verified on the latter.

Note that this approach can only be used to generate new implementations linked with their abstract formal models; in principle, the MDD methodology cannot be applied to already existing, legacy implementations. In order to handle such legacy implementations too, the MDD approach is leveraged to generate formally sound monitors of security protocol actors. When the monitor is in place, it stops all incorrect protocol executions generated by the legacy application. In other words, the soundness relation between legacy applications and their abstract models is enforced by the monitor.

This work is divided in two parts. Part I presents the overall MDD workflow that can be used to generate sound implementations from verified formal models, addressing theoretical and practical aspects. Specifically, chapter 2 presents the methodology, while chapter 3 discusses how to ensure soundness of the generated implementations during critical refinement steps from the model to the implementation code. Finally chapter 4 presents a case study about the formally sound implementation of an SSH Transport Layer Protocol client, by using the MDD approach proposed in this work.

Part II focuses on the monitoring approach for legacy implementations, where sound monitors are generated from protocol agent models, in order to check agent's implementations behavior. In particular, chapter 5 shows how MDD monitor implementations can be obtained by fitting into the previously discussed MDD workflow for security protocol implementations. Then, chapter 6 validates the approach by showing a case study about monitoring several SSL legacy implementations.

Part I

Formal Model Driven Development of Security Protocol Implementations

Chapter 2

Methodology

With the ever-increasing security requirements of software applications, the use and importance of software components that offer security services is increasing too. Such components are based on cryptographic protocols, in order to offer general-purpose services, such as authentication or key agreement, but also domain-specific services, such as for example secure payment in electronic commerce applications.

Despite their simplicity, cryptographic protocols are difficult to get right, and in principle, the gap between an abstract model and a concrete implementation may be responsible for certain security faults, not found when the abstract model is analyzed. Here a model-based approach to cryptographic protocol implementation is described, which leverages the capabilities of high-level formal analysis tools and also gives support in order to ensure the correspondence between high level formal models and their implementations.

The procedure consists of refining abstract models into implementations, with the guide and support of specific tools, which provide automatic code generation facilities and code reuse. The architecture of the implementation code is not left to the programmer, but is defined in such a way that all the most critical code sections are either automatically generated or already included in a general-purpose library. The role left to the developer is mostly to write the high-level formal model, to provide some implementation choices, and to write specific parts of code. Hand-written code is statically analyzed, in order to ensure that it does not introduce security faults. Rapid prototyping is possible just after having produced and validated the first high-level formal model.

A prototype tool, called Spi2Java, which supports this approach, has been developed [55]. In describing the development method, the functions of the prototype tool will be specifically illustrated.

The formal high-level language that has been adopted is spi calculus [3], an extension of π-calculus [50], whereas the target language is Java. The spi calculus language has been chosen because, besides having good tool support, can describe cryptographic protocols and their security requirements in a quite simple and accurate way. Thus, formal specification is affordable for developers without strong background on formal methods and the gap between the formal model and its implementation in a programming language is smaller than it would be using other formalisms. For example, several other formal

languages proposed for cryptographic protocols cannot precisely specify how messages are processed, and assume that message processing always occurs in a predetermined general way.

On the implementation side, the Java programming language has been chosen for different reasons. First, being the Java code run within the Java Virtual Machine (JVM) environment, it is not affected by buffer overflow and similar anomalies (except those present within the JVM itself, that are outside the scope of this work) that heavily affect security related implementations developed in other programming languages, such as C or C++. Moreover, Java is object oriented, and offers a pluggable system with clearly defined interfaces to handle cryptography and cryptographic providers (Java Cryptography Architecture – JCA). These two features, combined together, allow the generated code to be modular and re-usable, minimizing user effort. Finally, as some parts of the implementation must be manually written, it is believable that the widely used Java language can easily be accepted by the application developers.

The rest of this chapter is organized as follows. Section 2.1 introduces the spi calculus, the domain specific language for security protocol specifications that will be used throughout the document. Section 2.2 presents a reference example that will be developed in section 2.3 to practically illustrate the proposed methodology. Section 2.4 briefly comments some metrics gathered on the generated applications for the reference example. Section 2.5 compares the methodology and tools presented here to other existing ones. Finally, section 2.6 concludes.

2.1 The Spi Calculus

The spi calculus [3] is a formal domain-specific language that can be used to abstractly specify security protocols. It extends the π-calculus [50], by adding a fixed set of cryptographic primitives to the language, namely symmetric and asymmetric encryptions, and hash functions, thus enabling the description of the main security protocols. Adding more custom cryptographic primitives to the language, so that more security protocols can be described, is rather straightforward.

A spi calculus specification is a system of concurrent processes that operates on untyped data, called terms. Terms can be exchanged between processes by means of input/output operations. Table 2.1 contains the terms defined by the spi calculus, while table 2.2 shows the processes.

A name n is an atomic value; a name that is not bound is free. A pair (M,N) is a compound term, composed of the terms M and N. The 0 and $suc(M)$ terms represent the value of zero and the logical successor of some term M, respectively. A variable x represents any term, and it can be bound once to another term. A variable that is not bound is free. The $M^{\sim}$ term represents a shared key built upon key material M, while the $\{M\}_N$ term represents the encryption of the plaintext M with the shared key N. The $H(M)$ term represents the result of hashing M. The M^+ and M^- terms represent the public and private part of the keypair M respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively.

$L,M,N ::=$	terms
n	name
(M,N)	pair
0	zero
$suc(M)$	successor
x	variable
$M^{\sim}$	shared-key
$\{M\}_N$	shared-key encryption
$H(M)$	hashing
M^+	public part
M^-	private part
$\{[M]\}_N$	public-key encryption
$[\{M\}]_N$	private-key signature

Table 2.1: Spi calculus terms.

$P,Q,R ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$P\vert Q$	composition
$!P$	replication
$(\nu n)P$	restriction
$[M\ is\ N]P$	match
$\mathbf{0}$	nil
$let\ (x,y) = M\ in\ P$	pair splitting
$case\ M\ of\ 0 : P\ suc(x) : Q$	integer case
$case\ L\ of\ \{x\}_N\ in\ P$	shared-key decryption
$case\ L\ of\ \{[x]\}_N\ in\ P$	decryption
$case\ L\ of\ [\{x\}]_N\ in\ P$	signature check

Table 2.2: Spi calculus processes

Informally, the $\overline{M}\langle N\rangle.P$ process sends message N on channel M, and then behaves like P, while the $M(x).P$ process receives a message from channel M, and then behaves like P, with x bound to the received term in P. The general forms $\overline{M}\langle N\rangle.P$ and $M(x).P$ allow for the channel to be an arbitrary term M. The only useful cases are for M to be a name, or a variable that gets instantiated to a name. A process P can perform an input or output operation iff there is a reacting process Q that is ready to perform the dual output or input operation. Note, however, that processes run within an environment (the Dolev-Yao attacker) that is always ready to perform input or output operations. Composition $P|Q$ means parallel execution of processes P and Q, while replication $!P$ means an unbounded number of instances of P that run in parallel. The restriction process $(\nu n)P$ indicates that n is a fresh name (i.e. not previously used, and unknown to the attacker) in P. The match

process executes like P, if M equals N, otherwise is stuck. The nil process does nothing. The pair splitting process binds the variables x and y to the components of the pair M, otherwise, if M is not a pair, the process is stuck. The integer case process executes like P if M is 0, else it executes like Q if M is $suc(N)$ and x is bound to N, otherwise the process is stuck. In the shared-key decryption process $case\ L\ of\ \{x\}_{N^\sim}\ in\ P$, if L is $\{M\}_{N^\sim}$, then the process executes like P, where x is bound to M. Similarly, in the decryption process $case\ L\ of\ \{[x]\}_{N^-}\ in\ P$, if L is $\{[M]\}_{N^+}$, then the process evolves like P, with x bound to M, and the same reasoning applies to the signature-check process. Note that the decryption processes are useful only when the correct kinds of keys are used.

The free names and free variables of a process P are denoted by the disjoint sets $fn(P)$ and $fv(P)$ respectively, and $fnv(P) = fn(P) \cup fv(P)$. Any arbitrary spi calculus process P can be ensured to have $fn(P) \cap fv(P) = \emptyset$ by applying α-renaming to it, so that every bound variable gets a unique identifier. A process is *closed* if it has no free variables.

The semantics of the spi calculus has been originally expressed by means of a reaction relation $P \rightarrow P'$, a reduction relation $P > P'$ and structural equivalence $P \equiv P'$ [3]. $P \rightarrow P'$ means that P can evolve into P' after a message exchange between two parallel components of P, $P > P'$ means that P can evolve into P' by performing some other (internal) operation, while $P \equiv P'$ allows processes to be rearranged so that the previous rules can be applied.

As the focus of this work is on open sequential processes interacting with the environment, an equivalent semantics for open processes based on a classical labeled transition system (LTS) is introduced. The given semantics is similar to the one presented in [1] for open applied π-calculus processes.

For any sequential process P, a τ transition $P \xrightarrow{\tau} P'$ means that P can evolve into P' without interaction with its environment. Formally, it means $P > P'$ or $P \equiv P'$. It is worth noting that the evolution $P \rightarrow P'$ is not possible if P is a sequential process. Instead, $P \xrightarrow{m!N} P'$ and $P \xrightarrow{m?N} P'$ mean that P can interact with its environment by respectively sending or receiving N on channel m. Formally,

$$\begin{array}{lll} P \xrightarrow{m!N} P' & \text{means} & \exists \overline{y} | \forall Q.\ (P|m(x).Q) \rightarrow (\nu \overline{y})(P'|Q[N/x]) \\ P \xrightarrow{m?N} P' & \text{means} & \exists \overline{y} | \forall Q.\ (P|(\nu \overline{y})\overline{m}\langle N \rangle.Q) \rightarrow (\nu \overline{y})(P'|Q) \end{array}$$

where $\overline{y}$ is a possibly empty list of names.

According to these definitions, it can be shown that the semantics of open spi calculus sequential processes can be expressed by the rules in figure 2.1.

2.2 A Reference Example

A simple, but significant example protocol is now introduced. Besides allowing the reader to get acquainted with the spi calculus, the example will be used throughout this chapter to show an application of the methodology proposed in this work. Here, an ASCII syntax of spi calculus is used: the 'ν' symbol is replaced by the '`@`' symbol, and the overline in the output process is omitted (input and output processes can still be distinguished by

$$
\begin{array}{rcl}
\overline{m}\,\langle N\rangle\,.P & \stackrel{m!N}{\rightarrow} & P \\
m(x).P & \stackrel{m?N}{\rightarrow} & P[^N/_x] \\
[M\ is\ M]\,P & \stackrel{\tau}{\rightarrow} & P \\
let\ (x,y) = (M,N)\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x][^N/_y] \\
case\ 0\ of\ 0 : P\ suc(x) : Q & \stackrel{\tau}{\rightarrow} & P \\
case\ suc(M)\ of\ 0 : P\ suc(x) : Q & \stackrel{\tau}{\rightarrow} & Q[^M/_x] \\
case\ \{M\}_{N^\sim}\ of\ \{x\}_{N^\sim}\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x] \\
case\ \{[M]\}_{N^+}\ of\ \{[x]\}_{N^-}\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x] \\
case\ [\{M\}]_{N^-}\ of\ [\{x\}]_{N^+}\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x]
\end{array}
$$

$$
\frac{P \stackrel{\tau}{\rightarrow} P'}{(\nu n)P \stackrel{\tau}{\rightarrow} (\nu n)P'} \qquad \frac{P \stackrel{m?N}{\rightarrow} P'}{(\nu n)P \stackrel{m?N}{\rightarrow} (\nu n)P'}\ ,\ n \neq m,\ n \notin fn(N)
$$

$$
\frac{P \stackrel{m!N}{\rightarrow} P'}{(\nu n)P \stackrel{m!N}{\rightarrow} (\nu n)P'}\ ,\ n \neq m,\ n \notin fn(N) \qquad \frac{P \stackrel{m!N}{\rightarrow} P'}{(\nu n)P \stackrel{m!N}{\rightarrow} P'}\ ,\ n \neq m,\ n \in fn(N)
$$

Figure 2.1: Formal semantics for open spi calculus sequential processes.

the different brackets); C-like comments are also allowed between `/*` and `*/`, or they start with `//` and extend until the end of the line. Moreover, by introducing a syntactic sugar, lists of n elements can be represented as `(A1,A2,...,An)`, and they are reduced into left associated nested pairs; so for example `(A,B,C)` becomes `((A,B),C)`. Note however, that where necessary to improve readability, some minor simplifications to the syntax have been made, with respect to the full syntax accepted by the real tools.

The proposed example is a client-server, two-messages challenge-response protocol, which authenticates the server to the client. If *Client* is the client, and *Server* the server, then an informal representation of the exchanged messages is

$Client \rightarrow Server : Na$
$Server \rightarrow Client : ID,M,\{H(M,Na)\}_{ShKey^\sim}$

where Na is a nonce, ID is the *Server* identification string, M is the data that *Client* wants to receive from *Server*, $H(\cdot)$ represents an hash function, $ShKey^\sim$ is a secret shared between *Client* and *Server*, and the notation $\{N\}_{K^\sim}$ denotes the ciphertext obtained from the encryption of N with the key $K^\sim$.

The goal of this protocol is to ensure that, for each session, if *Client* gets message M, then it can be sure that M was not altered and that it was really sent by *Server* in the same session. In order to reach this goal, when *Client* receives the *Server* message, first it must decrypt $\{H(M,Na)\}_{ShKey^\sim}$, obtaining $H(M,Na)$, then it must locally generate $H(M,Na)$, using the received M and the internally generated nonce Na. Server authentication is finally achieved if the two hash values are equal, because the intruder is not able to generate $\{H(M,Na)\}_{ShKey^\sim}$, since it does not know the secret $ShKey^\sim$.

Furthermore, as a secrecy requirement, $ShKey^{\sim}$ shall never be disclosed to the intruder, otherwise authentication would fail too, because the intruder would become able to generate the $\{H(M,Na)\}_{ShKey^{\sim}}$ message.

This is an informal high-level description of the protocol. In order to get a complete description, like the one provided by protocol standards, more information must be given. Specifically, the cryptographic algorithms to be used for each operation, the encoding of messages on channels and the underlying transport protocols must be specified, in order to enable interoperability of heterogeneous implementations. As all these "low-level" details are abstracted away by the spi calculus language, they can be ignored at this time but will be explicitly reported when necessary.

There is however something more that can be specified in spi calculus, which pertains to protocol implementation rather than to its specification. For example, the fact that the shared secret is accessed by the *Client* or by the *Server* through a key store, which is an associative array that pairs an identifier with some data, can be modeled using separate local processes for the key stores. Similarly, the fact that the contents of message M is obtained by the *Server* reading it from its local file system or from a DB can be modeled by introducing a process that represents the behavior of the data source.

```
 1: Client(keystoreC) :=
 2:   (@Na)
 3:   channel<Na>.
 4:   channel(resp).
 5:   let (ID,M,cryptedHash) = resp in
 6:   keystoreC<GET,ID>.
 7:   keystoreC(ShKey).
 8:   case cryptedHash of {Hash}ShKey in
 9:   [ H(M,Na) is Hash ]
10:   0
```

Figure 2.2: Fromal model of client protocol logic.

```
1: KeyStore(keystoreC,ID,ShKey) :=
2:   keystoreC(req).
3:   let (operand,IDrecv) = req in
4:   [ operand is GET ]
5:   [ IDrecv is ID ]
6:   keystoreC<ShKey>.
7:   0
```

Figure 2.3: Formal model of key store.

The client logic of a single protocol session can be modeled by the spi calculus process reported in figure 2.2. It must be recalled that the spi calculus is an untyped language. In the following explanation of the protocol, expressions like "the term x *is* a nonce" mean that the term x *is used as* a nonce in the current context, but no type is assigned to x.

At line 1, the *Client* process is declared. Its argument, *keystoreC*, is the channel that will be used to communicate with its key store. The reason why the process is parameterized in this way will be clear later on. At line 2, the nonce Na is declared as a restricted name, which in spi calculus models a data item initially not known to the intruder and not easily guessable. This is exactly the high level model of a nonce, freshly generated by a random number generator. At line 3 the generated nonce Na is sent on *channel*, which is the channel used by client and server in order to communicate with each other.

After the nonce is sent, at line 4 the client receives the server response *resp* on *channel*, and at line 5 the client parses it, getting the server identification string ID, the server message M, and the server ciphered hash *cryptedHash*. The `let` construct is used to decompose a message into its constituent parts.

At lines 6 and 7, the client retrieves the shared secret associated with the server identified by ID from its local key store, which is modeled by a send-receive interaction on the key store channel. More precisely, the client sends a pair containing an operand GET and an identifier ID. The GET operand asks the key store to retrieve the key associated with the alias ID, if any. If such a alias-key association is present in the key store, the client then reads back the secret key $ShKey$, otherwise the key store process gets stuck, also blocking the client on the input operation (note that the key store channel `keystoreC` is meant to be private, so neither the attacker nor other processes can have access to it). As the GET term is constant data accessible by any actor (attacker included), it is assumed to be implicitly declared in any process using it.

At line 8, the client deciphers the received $cryptedHash$ using the shared secret $ShKey$, and obtains the plaintext $Hash$, which is the value of the hash function computed by the server. Finally, at line 9, the client compares the received $Hash$ with the locally computed $H(M,Na)$. If the two hashes are equal, then the null process `0` at line 10 models that the protocol run terminates successfully, and the server is authenticated; otherwise, as soon as any of the previous operations does not succeed, the process becomes stuck, that is the protocol session is aborted.

In order to have a complete model where security properties can be formally checked, the model of the key store must be provided too. However, the generic model of a key store would be unnecessarily complex for our purposes. For this reason, only the behavior related to the execution of the protocol can be modeled. In particular, since the protocol has only read access to the key store, the key store process gives this kind of access only.

The key store model is given in the $KeyStore$ process reported in figure 2.3. At line 1, the $KeyStore$ process is declared, with three arguments: the channel *keystoreC*, used to communicate with its environment, an identifier ID, and its associated shared secret $ShKey$. At line 2 the key store receives a request, that is split at line 3 in its part: an operand and its related data. If the operand is GET (line 4), and if $IDrecv$ is equal to the identifier contained in the term ID (line 5), then, at line 6, the shared secret $ShKey$ is provided to the requesting process, and the $KeyStore$ terminates correctly at line 7. Of course, this process definition represents a single access to the key store. Multiple accesses can be represented by different parallel instances of this process.

Now that both the client logic and the key store have been modeled, the model of the whole client side of the protocol can be given:

```
1: ClientComplete(ID,ShKey) :=
2:   (@keystoreC)
3:   ( Client(keystoreC)
4:   | KeyStore(keystoreC,ID,ShKey) )
```

The $ClientComplete$ process has the server ID and the associated shared secret $ShKey$ as parameters. At line 2 the channel *keystoreC*, used by $Client$ and $KeyStore$ for internal communication, is created as a restricted data item, which models the fact that it is not directly accessible by the intruder. Indeed, communications among parts of the same system are not visible by a network intruder.

At lines 3–4 the $Client$ and $KeyStore$ processes are run in a *composition*, that means in a parallel interleaving. Since both processes are receiving the same *keystoreC* channel

as actual parameter, they will be able to synchronize and communicate with each other using that channel.

It is worth noting that different instances of the *ClientComplete* process will generate different instances of the *keystoreC* channel, so they will not interfere with each other on the private channel, thus modeling what actually happens with real different protocol sessions.

```
1: Server(keystoreC,DBc,ID) :=
2:   channel(Na).
3:   keystoreC<GET,ID>.
4:   keystoreC(ShKey).
5:   DBc(M).
6:   channel<ID,M,{H(M,Na)}ShKey>.
7:   0
```

Figure 2.4: Formal model of server protocol logic.

```
1: FileAccess(DBc,M) :=
2:   DBc<M>.
3:   0
```

Figure 2.5: Formal model of file access.

Now the server side is considered. The server logic for a single protocol session can be modeled as reported in figure 2.4. At line 1 the *Server* process is declared with three arguments: *keystoreC* is a channel used to communicate with the key store; DBc is a channel used to communicate with the source of information (e.g. the file system), in order to obtain the contents of message M; and ID is the server identification string that will be used for the current session. At line 2 the server receives the client nonce Na. At lines 3–4 the server asks the key store to retrieve the key associated with the alias ID, and reads the shared secret $ShKey$ back. At line 5 the server receives from the file system, on channel DBc, the contents of message M, which must be sent to the client. Finally, at line 6, the server sends its response to the client.

The key store can be modeled with the same $KeyStore$ process reported in figure 2.3. Similarly, the source of information can be modeled by a process that provides the contents of M once. The process $FileAccess$, reported in figure 2.5, takes two arguments: DBc is the channel used to communicate with the requesting process and M is the file contents. Since M is a parameter, it can change in different instances of this process, thus modeling different data being accessed at different times.

Finally, the complete server model can be written as:

```
1: ServerComplete(ID,ShKey,M) :=
2:   (@keystoreC)
3:   (@DBc)
4:   ( Server(keystoreC,DBc,ID)
5:   | KeyStore(keystoreC,ID,ShKey)
6:   | FileAccess(DBc,M) )
```

Like *ClientComplete*, the *ServerComplete* process creates the private channels, then runs the composition of the server components.

As a last step, the whole protocol model is defined.

```
1: Inst(ID,M) :=
2:   (@ShKey)
3:   ( !ClientComplete(ID,ShKey~)
4:   | !ServerComplete(ID,ShKey~,M) )
```

The `!P` construct means *replication* of P, that is, an unbounded number of instances of P that run in parallel, thus modeling the possibility to have multiple concurrent protocol sessions. In this particular model, the server ID and data M are user-provided. This behavior can be easily changed for example by hiding M inside the $ServerComplete$, bound by a restriction operator. The same shared secret material $ShKey$ and derived shared key $ShKey^{\sim}$ are used for all the instances of the protocol.

2.3 Development Methodology and Tool Support

In this section an approach to model-based, tool supported, application development is proposed. In order to show how this approach can be applied to real applications, an interoperable implementation of the reference example will be carried out while explaining the proposed methodology.

In order to enable the development of the applications and case-studies described in this work, the tools supporting the proposed MDD methodology have been re-implemented from scratch. Indeed, a preliminary version of the tools existed [61]. However, being an early prototype, such tools lacked some enabling features and theoretical foundation. An example of such enabling features is interoperability of the generated implementation, the possibility to extend the type hierarchy, an else-branch-enabled input language, the ability to perform type-casts and to interact with a key store. During the re-implementation of the tools, these enabling features have been taken into account since the early design stages. Moreover, the re-implemented tools are based on a firmer theoretical background, as they directly derive from their theoretical design and analysis, which is discussed here.

A data flow diagram of the proposed approach is reported in figure 2.6.

2.3.1 Writing and Verifying the Formal Model

The programmer starts from an informal description of the protocol, and manually derives a formal spi calculus model, which includes all involved processes: protocol agents, auxiliary processes such as key stores and file access facilities, and “instance” processes simulating protocol sessions. For the reference example, the informal protocol description, without low-level details, and the complete formal spi calculus model have already been described in the previous section.

It is worth pointing out that the full spi calculus model describes the behavior of the whole system, yet it does not express anything about its intended security properties. In order to enable formal verification of this model, the developer must still add the description of the security properties that the protocol shall satisfy.

The formalization of security properties depends on the verification tool that is going to be used. Each tool defines templates or patterns for the usual security properties, such as for example secrecy and authenticity. The reference protocol security properties (secrecy of $ShKey^{\sim}$ and authenticity of M) have been specified and formally verified with the ProVerif tool [22].

Although the main focus of this work is not on formal verification, a few hints about verifying the reference example model with ProVerif are given. In principle, it is important

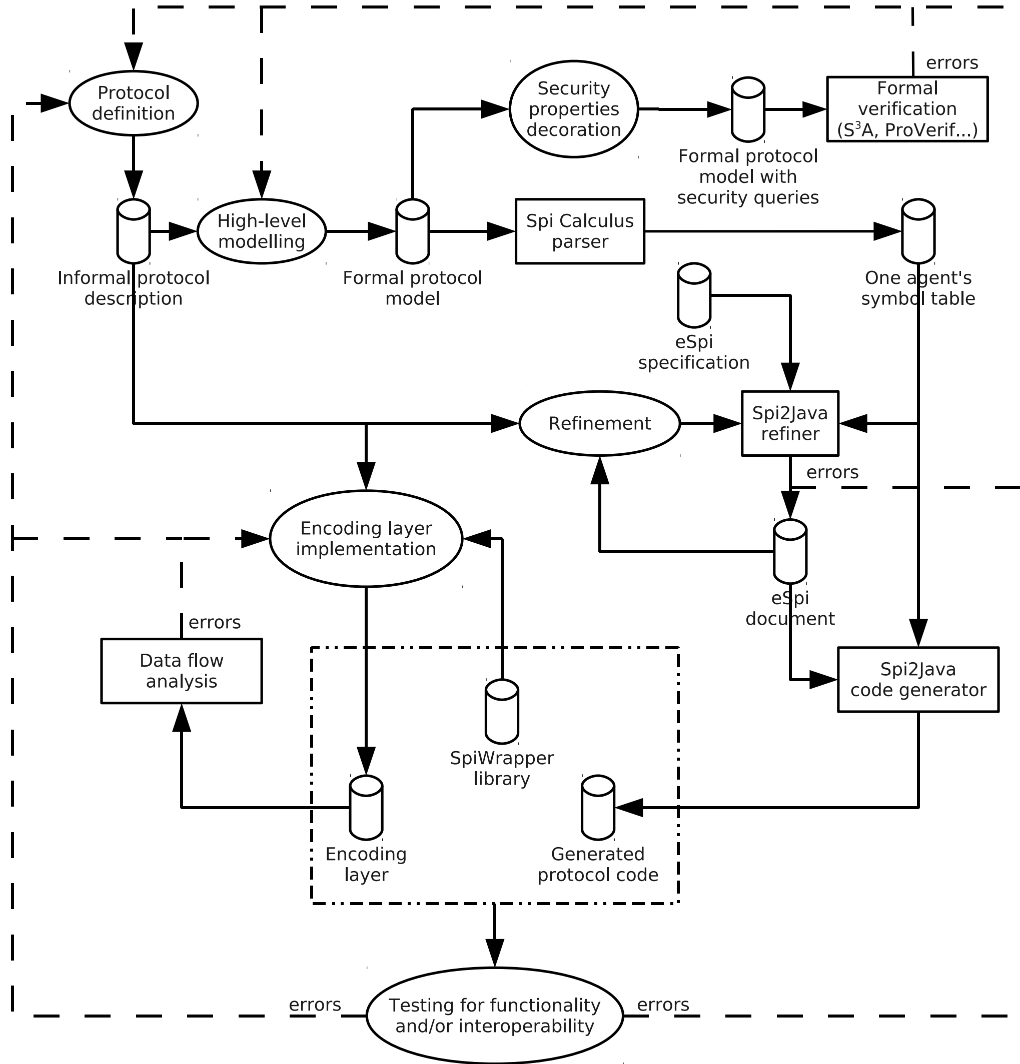

Figure 2.6: A data flow diagram of the proposed model-based development technique.

that the whole formal model is verified, before any protocol actor is implemented. If the formal verification step is not performed, then the derived implementation will still behave as the corresponding model, but no security claims could be made on it, because the original formal model is not proven to satisfy any security property.

ProVerif accepts a slightly different syntax than the one used by the Spi2Java framework. Fortunately, Spi2Java offers a tool, named Spi2Proverif, that translates from the Spi2Java syntax to the ProVerif one. Once this conversion is done, the developer can enrich the ProVerif specification with the queries about the intended protocol security properties. For example, secrecy of $ShKey^{\sim}$ is expressed by the query

```
query attacker:SK(ShKey).
```

where $SK(x)$ is a ProVerif constructor building a shared key out of material x.

Authentication is made by agreement over the session data M and Na. In ProVerif, for a simple client/server scenario like this reference example, agreement can be expressed by pairing *being*(x) and *end*(x) events, emitted by the server and the client respectively. Briefly, a *begin*(x) event signals that the server started a protocol session with the client, agreeing on some data x, while an *end*(x) event signals that the client ended a protocol session with the server, agreeing on some data x. Injective authentication is achieved if it is possible to prove that for any session that the client ended agreeing on same data x, the server started a session agreeing on the same data x.

In ProVerif, this can be expressed by properly placing the *begin*(x) and *end*(x) events in the specification, and by the following query:

```
query evinj:end(x) ==> evinj:begin(x).
```

The *begin*(M,Na) event is placed in the server specification before sending its message to the client; conversely, the client emits its *end*(M,Na) event after it receives that message.

2.3.2 Refining the Formal Model

When the formal protocol model is complete and verified, the description of one of the protocol roles (without security properties decorations) can be implemented. In general, it makes no sense implementing the whole protocol description as a single entity, but each role should be implemented as a separate program. In our example, the client and server parts of the model can easily be identified, and implemented separately. They correspond to the bare $Client$ and $Server$ processes, and not to the $ClientComplete$ and $ServerComplete$ processes, because the latter model both the protocol logic and some services offered by the platform where the programs run. The complete model is needed for verification but not for implementation.

Once the processes containing the protocol logic are identified, the programmer can start implementing any of them, supported by tools for refinement and code generation. Here, the use of the Spi2Java tools is presented, which supports implementation in Java, but other similar tools can be developed for other target languages.

In order to derive a Java application from the spi calculus source, the Spi2Java refiner can be used to fill the low-level implementation details that are abstracted away by the spi calculus language.

A tool like the Spi2Java refiner can automatically infer *some* information about the missing details that are not present in the formal high-level model. For example, the type of certain data can be automatically inferred by looking at how they are used. The implementation details that cannot be automatically inferred must be manually provided by the user, who can get them from the informal protocol description.

However, an interesting feature of the proposed methodology is the possibility to get early prototyping without any manual intervention, just after having written and validated the formal model. In order to get the early prototype, the tool can fill all the missing needed data with default values, which immediately gets to a completely refined specification. The user can later change the default values to accommodate needs; after

editing, the Spi2Java refiner checks the user-given values for correctness and coherence with the reference spi calculus specification.

The low-level implementation details can be grouped into two main categories:

1. Cryptographic and Configuration parameters
2. Encoding/decoding functions (or, simply, encoding functions)

The first group of details specifies parameters such as "what algorithm must be used for a particular encryption operation" or "what network interface must be used by a particular channel". In order to make the generated code compliant with the implemented protocol, it is necessary that these parameters can be set independently, at compile-time or at run-time, for each data item.

The second group of details deals with the transformation from the internal representation of messages into their external representation, and vice versa. The internal representation is the one used to perform all the operations prescribed by the protocol logic on the data; whereas the external representation is the stream of bytes that must be exchanged with the other parties. Decoupling internal and external representations is necessary in order to obtain interoperability, because the external representation must conform to the agreed binary formats defined for the protocol.

Another task that the Spi2Java refiner carries out is to statically assign a type to each spi calculus term. This is necessary because the spi calculus is an untyped language, while Java is statically typed. An extensible hierarchy of parametric types, reported in figure 2.7, has been defined for this purpose. In this chapter, only the essential details about the type system and related type inference algorithms are given, with the main goal of enabling the reader to understand the proposed methodology. Full formal details about the type system and type inference algorithms are given in section 3.1.

Some significant types are now described:

- *Message* is the most generic type: it represents an opaque message;
- *Name* represents an atomic term that can be freshly instantiated. A term that is not a name is composed, and cannot be freshly instantiated.
- *Channel* has some extensions that are worth noting:
  - *Tcp/Ip Channel* provides access to the Tcp/Ip communication layers;
  - *Key Store Channel* provides access to the system provided key store;
  - *File Channel* provides access to the local file system;
  - *Cast Channel* enables type casts to be performed.

It must be pointed out that *Private Key* and *Public Key* are not atomic names because they are always derived from a *Key Pair*; *Shared Key* $\langle A \rangle$ is not atomic too, because it is built upon some key material (of type A).

The extensible type hierarchy allows new types to be added when the need arises. For instance, new channel extensions could be defined, in order to provide access to other

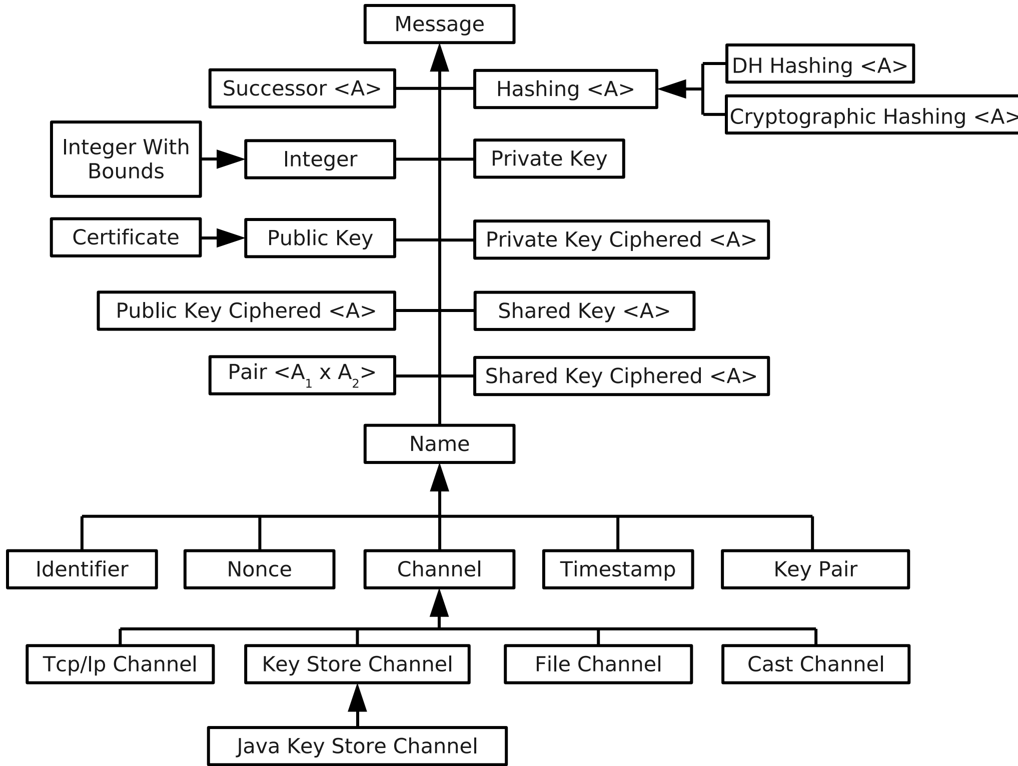

Figure 2.7: The currently defined eSpi type hierarchy.

communication layers, such as Udp/Ip, or to other system provided functions, such as databases.

Types are assigned by the Spi2Java refiner using a set of *inference rules*, that, based upon the use of a term in the spi calculus model, assign it the correct type. As stated above, it is possible that the type of a term needs to be manually refined into a more specialized type. However, the Spi2Java refiner checks that the type hierarchy is not infringed. For instance, if a term is automatically typed to *Channel*, it can be manually refined to the *Tcp/Ip Channel* or the the *Java Key Store Channel*, but it cannot be typed as *Message* or *Timestamp*.

Furthermore, there is a relationship between the type of a term and its associated low-level parameters. On one hand, each type has its own extensible set of cryptographic and configuration parameters. For instance, the *Shared Key* type has the *algorithm*, *strength* and *provider* parameters, which respectively specify the key cryptographic algorithm, the key strength and the Java Cryptography Architecture provider that will implement the required cryptographic functions. On the other hand, for each type an extensible set of Java classes can implement the encoding/decoding functions.

In order to store, for each spi calculus term, the assigned type and its low-level implementation details, the Spi2Java refiner uses an eSpi (extended Spi) XML document, which

is coupled with the original spi calculus source. The eSpi document is automatically generated by the Spi2Java refiner, and, when no information can be automatically inferred, default values are used.

In order to let the type hierarchy and the associated parameters be extensible, an XML document, called the eSpi specification, which contains these pieces of information, is parsed at run-time by the Spi2Java refiner; no information is hard coded into the tool. The default values of parameters are also stored into the eSpi specification, so that they can be modified at any moment too.

For further agile prototyping, a default encoding/decoding layer, which uses the Java serialization, is provided; however, in real environments, this default encoding/decoding layer has to be substituted with a user-given layer in order to implement the desired protocol encoding scheme.

Moreover, the Spi2Java refiner allows cryptographic and configuration parameters to be either specified statically at compile-time, or dynamically resolved at run-time. The latter behavior allows the implementation of protocols, such as for example the SSH Transport Layer Protocol, that prescribe cryptographic algorithm negotiation at run-time; the negotiated algorithm is stored into a spi calculus term whose value will be used as a parameter for a cryptographic operation.

With respect to the data flow diagram in figure 2.6, once the process to be implemented has been identified, the full formal model can be passed to the Spi2Java parser, which generates an intermediate symbol table containing only the process to be implemented. Then, the Spi2Java refiner takes this symbol table and the eSpi specification, and generates a first version of the eSpi document coupled with the given symbol table. In this eSpi document all information that could be automatically inferred is stored, and default values are used elsewhere. This first version of the eSpi document can be directly fed, along with the coupled symbol table, to the Spi2Java code generator. This will produce a working, but not interoperable implementation, useful for early prototyping. Otherwise, the eSpi document can be manually refined, adding information that could not be automatically inferred. The manually refined eSpi document, the coupled symbol table and the eSpi specification are then given back to the Spi2Java refiner, that checks consistency on the manually modified eSpi document, and outputs an updated eSpi document, taking into account user-provided refinement information.

In the reference example, both client and server will be implemented. Note though, that their implementation will happen independently from each other. That is, these refinement and implementation steps must performed once for the client side, and once for the server side. Nevertheless, they require very similar operations. For this reason, only the steps for the server side are reported here in detail. The client side will be referred to only when it significantly differs from the server side.

The eSpi document for the *Server* process can be obtained in two steps.

1. Only the spi calculus model and the eSpi specification are given as input to the Spi2Java refiner. The tool generates an eSpi document using all the information that can be automatically inferred from the given model. In particular, types are assigned to terms based on their use, while cryptographic and configuration parameters and

the encoding layer are set to their defaults.

2. The generated eSpi document is manually refined, adding the needed information that could not be automatically inferred. Then the Spi2Java refiner runs again, taking the modified eSpi document, the spi calculus model and the eSpi specification as input. The newly generated eSpi document contains all the information that could be automatically inferred from the model and the manually provided information.

It must be pointed out that the refinement step 2 can be repeated as many times as needed, until a satisfactory eSpi document is generated by the Spi2Java refiner. However, in most cases, like in this example, one run of step 2 is enough.

By now, in order to get an early prototype of the protocol implementation, the default encoding layer can be used. Later on, the custom encoding layer prescribed by the protocol description will be implemented for both client and server, and their eSpi documents will be updated to let the implementations make use of the custom encoding layer.

For brevity, only small relevant parts of the eSpi document will be presented here. As an example, after step 1, the eSpi document of the *Server* contains, among others, the lines reported in figure 2.8.

```
1 <term id="4" name="Na_1" type="Message">
2   <codify>MessageSR</codify>
3 </term>
```

Figure 2.8: Generated eSpi document – Na fragment.

```
1 <term id="4" name="Na_1" type="Nonce">
2   <codify>NonceSR</codify>
3   <parameters>
4     <param name="size" type="const">16</param>
5   </parameters>
6 </term>
```

Figure 2.9: Refined eSpi document – Na fragment.

In the eSpi document, a `term` element contains additional information about a spi calculus term declared in the formal model. Three attributes are present in this element: `id` is a unique identifier for the term, used internally by the Spi2Java tools; `name` is a human readable representation of the term, useful in order to uniquely identify it when manually modifying the eSpi document (a numeric suffix is added to the name that appears in the original spi calculus model); `type` contains the information about the type that has been statically assigned to the current term. The value of the `type` attribute must be present in the eSpi specification, and must be coherent with the use of the term in the model, as inferred by the Spi2Java refiner.

In the reported example, the term Na_1 has been automatically inferred by the Spi2Java refiner to have the type *Message*. Indeed, no particular use is made of Na in the server, such that it can be automatically inferred to belong to a more specialized type, and it can be correctly considered an opaque message. However, the informal protocol description specifies that Na is a nonce, so this information will be manually specified in the eSpi document.

The `codify` element contains the name of a Java class implementing the encoding layer for the current term. In the example, the Spi2Java refiner sets the default encoding class `MessageSR`, which uses the Java serialization. This default value is stored into the eSpi specification XML document.

Since the Na term must be manually refined to the *Nonce* type, the fragment of the eSpi document reported in figure 2.8 is modified to the one illustrated in figure 2.9, where the type of Na is now *Nonce*, and the default encoding layer has been modified accordingly.

Finally, since the eSpi specification states that the *Nonce* type requires the "size" parameter, the `parameters` element is added, which contains the set of parameters required by this type. Each `param` element has also an attribute called `type`, which is used in order to specify whether the parameter is assigned at compile-time (`type="const"`) or it must be resolved at run-time (`type="var"`). If the parameter must be resolved at run-time, then its value must be the identifier of another term that will contain the value of the parameter. In the reference example protocol, it is specified that the size of the nonce is statically fixed to 16 bytes, so "size" can be set as a `const` parameter in the eSpi document.

It may be argued that manual modification of the eSpi document is not user friendly. This is true; however the whole Spi2Java tool, which is currently a prototype, is designed to be an integrated development environment (IDE). In this context, a convenient user interface could accept user input, and then could transparently handle XML documents, automatically filling default values or adding required elements, as defined in the eSpi specification. With this design in mind, the use of the machine readable XML document format, and the definition of the eSpi specification get even more importance. For example, when the user refines the type of Na from *Message* to *Nonce*, the IDE, according to the eSpi specification, can automatically change the default encoding layer, and can add all the required parameters for the new type, filling them with default values, or asking custom values to the user.

Furthermore, the IDE could guide the user through the data flow which is detailed in this work: from protocol model definition, to final program implementation, passing through formal verification and semi-automatic refinement. The IDE could automatically run the already implemented core tools when needed, thus reducing the required user interaction, and increasing productivity.

After all custom editing for each term in the eSpi document is done, the Spi2Java refiner runs again validating the user modifications. So the final eSpi document, which uses the default encoding layer, is obtained.

Finally, it is worth noting that, in the final server eSpi document, there is exactly one element, namely the `term` element referring to the *channel* term, that has four attributes, the fourth attributes being added during the manual refinement step:

```
1 <term id="3" name="channel_0" server="TcpIpServer" type="Tcp/Ip Channel">
2   [...]
3 </term>
```

In particular, the `server` attribute is used to specify that the process owning this term will act as a responder on the *channel_0* channel (using the responder channel implementation provided in the $TcpIpServer$ Java class), so the Spi2Java code generator will take this into account. Since no term in the *Client* process has been manually given a `server` attribute, the client will be implemented as an initiator.

On the client side, a `return` attribute is manually added to the term M in the eSpi document. This attribute specifies that M is a protocol session return parameter. Indeed,

a security protocol can have some return parameters, such as the shared secrets generated during the protocol session, which shall be used after the end of the protocol session. In a session of the reference example protocol, the client obtains the message M, which could be used later. For this reason, M is set as a protocol session return parameter. In particular, as many `return` parameters as needed can be declared in an eSpi document, and the Spi2Java code generator will take care to make all protocol return parameters available to the Java code that is executed after a successful protocol session.

2.3.3 Implementing the Java Application

Once the final eSpi document of the prototype version is done, the Spi2Java code generator is used in order to obtain a Java implementation of the given spi calculus model, refined with the information contained in the coupled eSpi document.

The Spi2Java code generator engine navigates all the expressions listed in the spi calculus source and translates each of them into a sequence of semantically equivalent Java statements. The mapping between a spi calculus expression and its corresponding sequence of Java statements is called a translation rule. Again, this chapter only provides the reader with the essential information about the translation from spi calculus to Java, with the goal of explaining the underlying methodology. Formal analysis of the spi calculus to Java translation is detailed in section 3.1.

In order to keep the translation rules simple, the Spi2Java tools come with a Java library, *SpiWrapper* (previously called *secureClasses*), that allows, one to one, spi calculus statements to be mapped onto Java statements. This is achieved by letting each class in the SpiWrapper library implement the semantics of one of the types declared in the eSpi specification, thus hiding its complexity. Moreover the Java *extends* feature is leveraged in order to comply with the type hierarchy stated in the eSpi specification.

The following example is a significant excerpt of the Java code belonging to the client implementation. This code is automatically generated by the Spi2Java code generator, and implements the refined spi calculus model of the client in its prototype version, using the default encoding layer. It must be pointed out that the generated Java code implements the plain spi calculus language, without any syntactic sugar for lists. Note how the Spi2Java code generator automatically adds comments showing the mapping between each spi calculus statement and its Java implementation, thus improving code readability.

```
  /* [...] */
1 /* let (ID_5,M_5) = _w0_4 in */
2 Identifier ID_5 = (Identifier) _w0_4.getLeft();
3 Message M_5 = (Message) _w0_4.getRight();
4 _return.put("M_5", M_5);
  /* [...] */
5 /* case cryptedHash_4 of {Hash_8}ShKey_7 in */
6 Hashing Hash_8 = (Hashing) cryptedHash_4.decrypt(new CryptoHashingSR(),
7   ShKey_7, "DES", "12345678", "CBC", "PKCS5Padding", "SunJCE");
  /* [...] */
```

The SpiWrapper library allows each spi calculus statement to be mapped on a few corresponding Java statements, and all operations on a spi calculus term are handled by

methods of the SpiWrapper class implementing the type assigned to the term. For example, at lines 5–7, the decryption construct in the spi calculus model can be implemented in only one Java statement by using the `decrypt()` method, which belongs to the `SharedKeyCiphered` SpiWrapper class. The latter implements the *Shared Key Ciphered* eSpi type, which has been automatically assigned to the $cryptedHash$ term by the Spi2Java refiner in the eSpi document. Moreover, since the term M (`M_5` in the Java code) has been set as a protocol return parameter in the eSpi document, at line 4 it is added to the `_return` map, which is an associative array mapping all return parameters to their names, so that the caller will be able to retrieve them after the protocol session, if it ends successfully.

The current version of the SpiWrapper implementation does not make use of the Java generic types, available from Java version 1.5. Although desirable to further improve the link between the parametric eSpi types and their SpiWrapper implementation, use of generic types is not strictly necessary: as better explained later on, generic types are implemented by erasure, and their same purpose is taken into account into the parameterized eSpi type hierarchy.

The Spi2Java code generator does not only generate the security critical Java code implementing the spi calculus model; instead, it always generates complete application templates that can be compiled and executed without any manual modification. This strongly reduces user interaction, enabling agile prototyping.

The Spi2Java code generator is able to implement applications using two different architectures, whose flowchart is reported in figure 2.10. If an application must act as an initiator, that is, no `term` element has the `server` attribute in the eSpi document, then the Spi2Java code generator automatically uses the iterative client architecture. Otherwise the application must act a responder on the channel having the `server` attribute, and the Spi2Java code generator automatically uses the concurrent server architecture. The Spi2Java code generator is designed such that it is easy to add new implementation architectures, such as concurrent crew servers (also known as "prefork") and the like.

It is worth noting that, while the `performHandshake()` method implements the logic of a protocol session, the `act()` method is executed only if the current protocol session ends successfully (that is, `performHandshake()` returns and does not throw an exception), and is initially empty. This method can be implemented by the user in order to perform any action that must be done after a successful end of the protocol session.

Although the Spi2Java code generator always generates code that can be compiled and executed without any modification, the input parameters of the `performHandshake()` method must be manually initialized before the program can be correctly executed, because no information can be automatically inferred on their contents. The method input parameters are the input parameters of the spi calculus process. Input parameters typed as *Channel* (or a subtype of it) do not need to be initialized, because the eSpi document already contains enough information for their initialization. For this reason, the *Client* process does not require any input parameter initialization, while the server needs ID, its identification string, to be initialized. It is worth noting that input parameters can be initialized at compile-time, or at run-time, for example based upon user input.

In order to preserve security, if any input parameter is not initialized, the generated

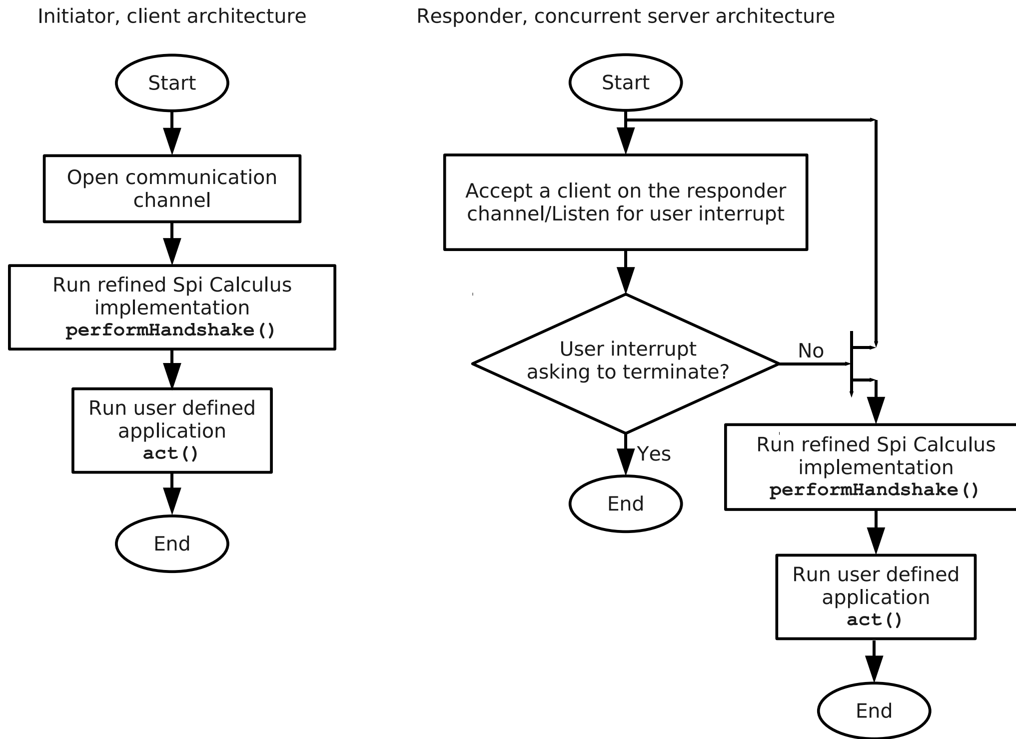

Figure 2.10: Flowchart of the initiator and responder architectures.

code will stop execution, even before the code actually implementing the spi calculus model is run.

Although the used encoding scheme is still not compliant with the protocol description, the prototype programs are fully implementing the protocol logic, so they can be used to test the protocol behavior and functionality. This test step is important, because a protocol could be verified safe, even if it is not functional, and thus useless. For example, suppose a protocol where all sessions abort at the very beginning, because of a wrong design. Then this protocol is probably satisfying secrecy (and possibly authentication) requirements, because the intruder cannot get the secret, since no message is ever exchanged. However, such a protocol is useless.

Once protocol functionality is tested with the prototype applications, the encoding layer can be implemented, in order to obtain the fully functional and interoperable applications.

In order to create the encoding layer, four abstract methods declared in the SpiWrapper classes must be implemented by the programmer for each type of encoding that is required by the specification. More precisely, the four methods can be implemented by extending the SpiWrapper class representing the type for which the encoding scheme is going to be written. It is worth noting that this approach isolates hand-written code with respect to automatically generated code.

The four methods that must be implemented are: `_encodePayload()`; `_serialize()`; `decodePayload()`; `deSerialize()`. (The first two methods begin with an underscore '`_`' for legacy reasons.)

The first method is responsible for translating the internal representation of a term into the payload, encoded as requested by the informal protocol specification. The second method is used to add the necessary headers and trailers to the payload. This approach gives high flexibility by allowing different and independent encodings for cryptographic and network operations.

The third and fourth methods are dual with respect to the first and second ones. `deSerialize()` extracts the payload from the whole data sent by the other parties, and `decodePayload()` transforms the encoded payload into the internal representation of the term.

With respect to the reference example, before its custom encoding layer can be implemented, the low-level details of the protocol must be explicitly defined in its informal description.

For brevity, only a portion of the low-level details that have been specified is reported here. For example, it is specified that all hashes shall be computed with the SHA-256 algorithm, and all nonces shall be 16 bytes long. A binary packet format has also been defined. For example, the binary packet that is sent from client to server is composed of

Encoding scheme	Content
`byte[16]`	The Na nonce.

where `byte[n]` is an array of n bytes. The binary packet from server to client has also been defined.

As an example of encoding layer implementation, let us consider the encoding scheme for the first protocol message. In this message, only the nonce Na is sent by the client to the server.

In the client spi calculus model, this operation is represented with the `channel<Na>` statement. Moreover, the final eSpi document assigns the *Nonce* type to the Na term. From this it follows that a term typed *Nonce* is being sent as the whole message, so an encoding scheme for the *Nonce* type is required.

First, the `NonceCH` Java class, implementing the four required methods, is written. This class extends the `Nonce` class defined in the SpiWrapper library, which implements the *Nonce* type declared in the eSpi specification.

Since in our example the protocol encoding scheme prescribes that the nonce payload is the whole packet payload, the `_encodePayload()` method writes the nonce payload on the given output stream, and the `_serialize()` method, which shall encapsulate the result of `_encodePayload()` into a binary packet, simply relays the payload on the output stream.

In the same way, the `deSerialize()` method decodes all binary packet information, and then delegates the `decodePayload()` method, which, following the protocol description, must read 16 bytes from the stream, and save them into the internal representation.

Once this class has been written, the client eSpi document must be updated to let the Spi2Java code generator use the right class for the encoding scheme. With respect to the

eSpi document fragment reported in figure 2.9, the contents of the `codify` element are changed from `NonceSR` to `NonceCH`. That is, the class that will implement the encoding scheme for the Na_1 term is changed from `NonceSR`, the default encoding layer that uses Java serialization, to `NonceCH`, the custom encoding layer that complies with the protocol description.

When the other encoding classes are written, and the client eSpi document is properly updated to make use of them, the Spi2Java refiner is executed, just to check that manual modifications did not break the eSpi document. Finally, the Spi2Java code generator is executed, so that the interoperable version of the client gets finally implemented in Java and is ready to be used.

The same procedure applies to the server side.

It is worth noting that the `NonceCH` class happens to be usable for the server encoding layer too. Indeed, the server receives the first message with the `channel(Na)` construct. Moreover the server eSpi document assigns the *Nonce* type to the Na term, so the same `NonceCH` class can be used. In general, different terms may require different encoding schemes.

Furthermore, all terms that are not sent or received through a channel and that are not involved in any cryptographic operation do not need an encoding layer, because only the internal representation is used. For instance, in the reference example all channels do not need an encoding layer, so the default one can be reused in the interoperable version too, and no more code needs to be written.

Once implemented, the client and server programs can be tested for interoperability and correctness. If errors are found, then the broken parts can be fixed, and, following the data flow shown in figure 2.6, the fixed applications can be generated again, until all the desired tests are passed.

As a means to further improve the security of the developed implementation, a static analysis can be performed on the hand-written part of the code, i.e. the implementation of the encoding and decoding functions. The aim of this analysis is to statically verify that each encoding or decoding function is a mere transformation from the internal representation of a data object to a corresponding external representation, or vice versa, without any other information flow to or from other data objects. More precisely, the analysis must ascertain, for example, that the output produced on the `out` OutputStream by any invocation of the `_serialize(OutputStream out)` method only depends on the value of the data object on which it is invoked, and on the encoding parameters used for encoding. A similar condition can be defined for the calls to the `_deserialize(InputStream in)` method.

It is worth noting that verifying these conditions is enough in order to assess the logical correctness of the implementation, if it is assumed that the semantics of the original spi calculus specification is indeed preserved by the translation rules from spi calculus to Java. This can be intuitively explained in the following way. A safe assumption is that the intruder knows the parameters used for encoding or decoding (otherwise we were assuming some kind of security through obscurity). With this assumption, if the encoding and decoding methods can be abstractly represented as mathematical functions that map a data object onto a byte stream and vice versa, an erroneous implementation of an encoding

method cannot make more harm than an intruder can. In fact, an intruder can execute any transformation on the data sent or received by the protocol agents, including the ones actually implemented by the real encoding and decoding methods. Therefore, the formal analysis of the abstract model, by considering any possible behavior of the intruder, already covers any possible implementation of the encoding and decoding methods, provided that the encoding and decoding methods are checked to really implement pure transformations, i.e. to be semantically equivalent to mathematical functions.

A formalization of this intuitive reasoning has been done with the CSP formalism, and can be found in section 3.2.

This kind of static analysis does not exclude some possible lower-level coding bugs, such as for example those related to integer overflows or other potential sources of vulnerabilities. However, specific static analyses are available for this purpose too [45], and they are fast and precise enough to be usable after the last refinement phase, in order to intercept and fix potential causes of vulnerabilities in the hand-written code.

2.4 Experimental Results

In order to evaluate the effects of the proposed methodology on the generated applications, the client and server reference example implementations presented in the previous sections have been measured. Although the reference example is not a standard protocol, it is still significant, because it prescribes in detail all agent behaviors. A full case study considering the SSH Transport Layer Protocol is illustrated in chapter 4.

For brevity, only client metrics are reported in table 2.3. Server metrics have also been measured, and they share the same magnitude, so the same comments apply.

Package	TLOC[a]	MLOC[b]	Ca[c]
spiWrapper	3768	2180	102
spiWrapperSR	1554	738	11
spiWrapperCH	374	159	3
challengeClient	117	69	0

Table 2.3: Measures of the reference example client code.

[a]Total Lines of Code: non-blank and non-comment lines in a class.
[b]Method Lines of Code: non-blank and non-comment lines inside method bodies of a class.
[c]Afferent Coupling: number of classes outside a package that depend on classes inside the package.

The spiWrapper package contains the *SpiWrapper* library, which is used to implement the operations prescribed by the protocol logic. The code inside this package is shipped with the Spi2Java tools. The spiWrapperSR package contains the default encoding layer, which is part of the reusable code library shipped with Spi2Java, while the spiWrapperCH package contains the manually written custom encoding layer, which allows interoperability. The challengeClient package contains the protocol logic automatically generated by the Spi2Java code generator.

It can be argued that *TLOC* and *MLOC* metrics highly depend on coding style. This is true; however, all packages have been written with the same coding style and rules. Because of this, it can be assumed that, in this particular context, both *TLOC* and *MLOC* are significant.

By looking at the *TLOC* and *MLOC* metrics it is possible to conclude that, at least for the reference example, the library reusable code (spiWrapper and spiWrapperSR packages) composes the greatest part of the whole application, whereas the manually written code (spiWrapperCH package) amounts to few hundreds lines. The automatically generated code (challengeClient package) is short too, being only responsible for the protocol logic, that is the critical code coordinating cryptographic and input/output operations. This shows that the manual effort in deriving the application is limited, mainly because the reference example has a relatively simple encoding scheme, and because the default encoding layer is reused where possible. In general, when dealing with larger protocols requiring more complex marshaling layers, the manual effort is expected to increase, although the reusable library code and the generated code still being a significant part of the application.

The *Ca* metric shows code dependencies instead. As expected, the "top-level" package challengeClient is not required by any other package, because it only contains the protocol logic, coordinating the functions offered by the underlying libraries. The marshaling layer packages (spiWrapperSR and spiWrapperCH) are used by the classes within the challengeClient package, and the spiWrapper package is required by all other packages, since the protocol logic requires it to perform the cryptographic operations, and the marshaling layer requires it to get the internal representation of data. The analysis of this metric brings to the conclusion that the proposed methodology also helps in creating structured applications, improving application maintainability and code reuse.

2.5 Related Work

Formally linking security protocol models and their implementations is a relatively recent research field. During the last years, several approaches have been proposed.

The approach proposed in [20] is the dual of the approach presented here. Instead of generating an interoperable implementation from a verified security protocol formal model, it extracts a verifiable formal model from an interoperable implementation of a security protocol, written in *F*, a subset of *F#* [69]. In [20], the model extraction algorithm is proven sound so that any property verified on the formal model also holds on the implementation. In [20], like in the work being presented here, correctness of some low level cryptographic libraries is assumed, that is, the concrete low level libraries are assumed to behave like the abstract symbolic counterparts. The model extraction approach has the advantage of allowing existing implementations to be verified without changing the way applications are currently written. However, although promising, this approach currently uses a functional programming language that is not very common in practice, as opposed to the imperative or object-oriented languages such as C or Java often used to develop implementations of security protocols. Moreover many constraints are put on the *F* source,

so that currently only *ad hoc* written applications could be verified.

Another available technique is to add refinement types to an existing implementation source code, so that some security properties can be proven on the implementation [17]. This approach, like the previous one, is promising, although the supported source code is usually a restricted version of a functional language. Moreover, this approach currently only allows to verify correspondence properties, such as authentication, but cannot deal with the secrecy property, which is very often required in security protocols indeed.

Another dual approach that derives and verifies the formal model from C source code implementations is presented in [40]. It extracts a simplified control flow graph from the C source code, and then translates this graph into a set of first-order logic axioms. The obtained axioms, together with a logical formalization of the requested security properties, are finally checked with an automated theorem prover.

A framework to directly evaluate security properties on an annotated subset of C source code is proposed in [31]. The user must annotate the source code in order to create a link between C functions and data, and symbolic Dolev-Yao cryptographic primitives and data. Then a symbolic execution environment must be defined, representing all protocol roles not implemented in the C application. Finally, static analysis can be run on the application. Currently, only secrecy properties are supported. It is worth pointing out that this and the previous approaches expose the programmer, all at once, to the whole protocol logic and implementation complexity, because a complete protocol implementation must be provided, before it can be verified. In contrast, the model-driven approach presented here lets the programmer focus on high-level issues first, and then on specific implementation aspects.

An approach very close to the one presented in earlier versions of Spi2Java is described in [70]. However, the translation algorithm is not proven sound, and neither the generated code is interoperable, nor algorithms and parameters can be selected at run-time or for a specific cryptographic operation.

In [14], an approach to automatic Java implementations of Zero-Knowledge Proof of Knowledge (ZK-PoK) protocols from their specifications is presented. The work only focuses on ZK-PoK protocols, and does not deal with classical security related protocols, such as SSH or SSL.

An on-the-fly, fast C source code security protocol generator is presented in [42]. The main goal of the work is to provide a fast protocol generator, that can be used in portable terminals to replace/update current protocol implementations. Formal soundness of the generated implementations is not addressed in that work.

AGVI [74] is a framework for the design and implementation of custom security protocols starting from their security requirements. Given some security requirements, AVGI generates a family of custom security protocols that fulfill those security requirements. Then the user can generate implementations of a chosen custom protocol. Although useful, this approach does not allow to deal with standard protocols, nor to generate interoperable implementations.

ACG-C# [38] is a tool that automatically generates C# code from a verified Casper script [48]. This tool does not deal with the interoperability of the generated code. Moreover, ACG-C# accepts scripts that are slightly different from the verified Casper scripts.

This requires manual modification of the verified Casper script in order to let ACG-C# generate the code. These manual changes are error prone, and the generated code starts from a model that is not the verified one. Instead, the Spi2Java tools use the verified spi calculus model for code generation, and all implementation details are stored in a separate (but coupled) eSpi document, which does not alter the spi calculus source. It is also worth noting that the Casper scripting language is not as expressive as the spi calculus language. Because of this, with ACG-C# it is more complex to exactly model the behavior of actors, as prescribed by informal protocol descriptions. Finally, since in AGC-C# a formal translation function is not provided, it is not possible to reason about soundness proofs.

The work presented in [32] illustrates another approach useful to translate formal models into Java code. In particular, the problem of mapping abstract data types into implemented Java classes is addressed in [33]. It is proposed to represent all abstract data types as byte arrays. Since allowed data contained in abstract types can always be mapped into a subset of all possible byte arrays, a middleware is provided to check that the byte arrays stored in Java classes belong to the allowed subset. However, this work does not take into account interoperability, because it does not deal with the different encodings (different byte array representations) that can be assumed by the same abstract data item, when it is sent to, or received from other actors.

In [35], a formal model of a security protocol used by smart cards is derived and refined from its informal description, then a Java implementation of the refined model is manually derived. Finally, JML [44] properties are manually added to the Java source in order to verify that the implementation adheres to the refined specification. No automatic tools are used to refine the model, nor to generate the Java implementation and the JML properties. Because of this, the approach is error prone, requiring manual work in all development stages. Since no formal translation rules from CSP processes to JML constraints are provided, there is no formal evidence that the specified JML annotations correctly express the properties of the refined model. This solution is interesting nevertheless, because it leads to Java code that can be directly verified.

In web services, security properties are expressed at a higher level, as policy assertions [11, 12]. Rather than specifying *how* security is achieved, through the coordination of cryptographic primitives inside the protocol specification, a policy assertion specifies a property that must hold for a specific set of SOAP messages. If the policy assertions implementation is assumed to be correct, then, if the properties expressed by the policy assertions are verified to be the desired security properties, then the web service implementation can be considered secure.

The tool described in [19], checks user given policy assertions, finding common flaws. It does not give any formal proof about the correctness of the user given policy assertions, however this tool can increase the confidence about them. Still, there is the need to verify that the policy assertion implementations are correct. The work presented in [18] gives a verified reference implementation of the WS-Security [51] protocol, written in *F#*. This implementation can be used in future complete protocol implementations as a starting point. However, no tool that verifies an user given implementation of policies, nor that generates a correct implementation from user given policies, is, to the best of our

knowledge, currently available for web services policy assertions.

2.6 Discussion

Formal methods have the potential to significantly improve the trustworthiness of critical software artifacts, especially when development turns out to be intrinsically error prone, and bugs difficult to discover, as it happens with cryptographic protocols. However, one of the problems that still limits the widespread use of formal methods is the high level of expertise that they normally require and the high cost of development that their use implies. A way to partially overcome the above problem and to improve the acceptance and productivity of formal methods is to provide methodologies and tools that simplify the use of formal methods, introducing automation and hiding underlying complexity.

In this work, a model-based methodology for cryptographic protocol implementation that goes in this direction is proposed. The combined use of code generation and of a flexible library limits the amount of code that the programmer must write by hand. Indeed, the programmer's task is limited to formalize the high-level protocol behavior, which is formally checked by a tool, to provide implementation choices, and to write well delimited sections of code. Moreover, thanks to the modular software architecture and to the features of the SpiWrapper library, the hand written sections of code can be checked for correctness by efficient static analysis procedures.

A first important effect of this approach is a reduction of the possibility of introducing security bugs, because they are proportional to the amount of hand-written code. For example, security bugs caused by missing high level checks on input messages are avoided, because the formal model explicitly includes checks at the abstract level, and if one of such checks is needed for security, its absence is detected by formal analysis. Similarly, security bugs caused by lack of low-level checks can be detected by static analysis on the hand-written code that performs message decoding.

A second important effect of the proposed approach is an improvement in development simplicity and productivity, which is achieved not only by limiting the amount of hand-written code, but also by enabling early prototyping and incremental development. Specifically, the methodology leads the programmer to handle the various aspects (protocol logic, cryptographic algorithms, message encodings) one at a time, thus avoiding the complexity of simultaneously keeping all of them in mind.

By the examples reported in this work, the presented methodology has also been shown to be sufficiently flexible to enable the implementation of different kinds of protocols, including those that interact with a local key store or with other data storage facilities, and to get interoperable implementations of standard protocols.

An interesting topic left for further study is the possibility to automate the generation of the encoding and decoding functions, starting from high level descriptions such as ASN.1 or NetPDL [62]. On one hand, this could further increase the security trustworthiness of the final application; on the other hand, it could improve productivity during application development.

Chapter 3

Specific Issues

This chapter analyzes in full details some specific issues that have been particularly challenging when designing and developing the MDD approach proposed in this work. All the considered topics are related with the final goal of ensuring soundness of the generated application, with respect to the starting formally verified Dolev-Yao model.

A first critical aspect is related to the function that translates single spi calculus statements into sequences of Java statements. In order to be useful, this function must be formally proven to generate Java code that behaves accordingly with the original spi calculus specification. In order to achieve this goal, one pre-requisite is that there is a way to link a Java implementation with a spi calculus model. Unfortunately, there was no such work available in the literature. So, first this link has been formally defined, and then it has been shown that the spi calculus to Java translation function actually generates Java programs that are correctly linked with the original spi calculus specification. Furthermore, it has been proven that the link between spi calculus and Java has been defined in such a way that safety properties that hold on the spi calculus model are indeed preserved in the linked Java implementations. Since many classical security properties, such as secrecy and authentication, are safety properties, this leads to the main formal result stating that any Java program generated by the Spi2Java framework shares the same safety (security) properties that can be proven on the original spi calculus model.

The generated code relies on the custom SpiWrapper library for cryptographic and input/output operations, and it also employs user-provided method implementations for encoding functions. In principle, these two pieces of code may introduce Dolev-Yao security related flaws that cannot be captured when analyzing the formal model. For example, the SpiWrapper library could enable reply attacks by incorrectly generating non pseudo-random nonces; or a wrongly implemented encoding function could leak a secret key by just writing it on the communication channel, when it is not supposed to.

The SpiWrapper section of code has been dealt with by formally expressing the intended semantics of all the methods exposed by the library. Then, under the assumption that the implementation of the SpiWrapper library behaves as formally specified, it has been shown that the Java code generated by the Spi2Java framework correctly uses this library, thus preserving the link between the generated code and the original spi calculus

model. An orthogonal challenge is still to show that there exists a SpiWrapper implementation actually complying with its formal intended semantics. As a proof of concept, some methods of the SpiWrapper implementation shipped with Spi2Java have been formally proven to adhere to the intended semantics.

The user-provided implementation of the encoding functions has been treated differently instead, because it is under the application developer control, and is not part of the Spi2Java framework. By creating a formal model of such encoding functions, sufficient conditions have been found, so that when they are met, any implementation of the marshaling function cannot interfere with the Dolev-Yao security properties. In the simplest scenario, when the encoding functions are used over communication channels to operate data marshaling, it turns out that the sufficient conditions are simple data flow properties (such as: the method implementation does never read secret data) that can be easily verified in isolation, on sequential code. More constraining conditions are in place when such encoding functions are used to transform data that are going to be encrypted or hashed, because in this case injectivity becomes necessary, which is not trivial to be proven on custom Java code.

In this chapter, section 3.1 focuses on the spi calculus to Java translation function, and on the usage of the SpiWrapper library by the generated code, while section 3.2 focuses on finding the sufficient conditions on the encoding functions.

An abridged version of the content of section 3.1 appeared in [57, 60]. Part of the content of section 3.2 appeared in [59, 58].

3.1 Translation

This section aims at improving the correctness assurance that can be achieved by using an automatic code generation approach, by formally defining what was informally described in chapter 2, that is a translation function from the spi calculus specification language, to the Java implementation language. The translation function relies on the SpiWrapper Java library which essentially wraps the Java Cryptography Architecture library calls that implement cryptographic primitives in the Java environment, and low-level socket and stream primitives. This section formally shows that, if it is assumed that the implementation of the SpiWrapper library satisfies some conditions that are formally expressed, then the generated Java code correctly refines the abstract model. Moreover, a verified implementation of part of the library is presented.

The following section 3.1.1 presents a type inference system for the spi calculus and a formal translation from the spi calculus to Java, while section 3.1.2 presents the main correctness property of the translation, i.e. that the generated Java implementation simulates the spi calculus specification it has been generated from, and shows a possible verified implementation of part of the custom library used by the generated code. Finally, in section 3.1.3 the results are discussed and some hints for future work are given.

Note that both the type system and the translation function rely on the assumption that spi calculus processes are sequential and non-recursive. That is, composition, replication and recursive processes are not handled. In fact, this is not an important limitation,

and it does not prevent the results obtained here to apply to most security protocols. Indeed, each session of a security protocol is very often composed of two or more sequential agents acting concurrently in a distributed environment. Multiple protocol sessions can run concurrently. Accordingly, a security protocol is generally described in spi calculus by an expression of the form $!P_1|!P_2|\cdots|!P_n$, where each protocol agent P_i is a sequential process. Replication is used to express the possibility of spawning multiple instances of each agent, acting in different protocol sessions, and composition is used to express that the different agents run concurrently. The composition and replication processes are rarely used within each agent's specification.

As explained along with the methodology, during formal verification the whole protocol specification (including replication of agents and their composition) is considered, so that all the possible scenarios are verified. When deriving the implementation, however, each sequential agent is translated into Java separately. Replication and composition are implicitly implemented by running several instances of the sequential actors in different Java threads, or in different machines.

Allowing replication and composition inside each actor would be practically not so useful. At the same time, it would make the formalism and the proofs presented here much more complex. This would happen because, as it will be clear in the rest of the work, more semantic rules would have to be considered for both spi calculus and Java, and each proof would have to be extended to consider the possible interleavings of concurrent Java threads. Recursion could be useful to describe protocol runs of unbounded length, but this feature is normally not allowed by verification tools anyway.

Finally, note that the Spi2Java tools offer preliminary support for composition handling, by generating code that uses Java threads. However, no formal proofs of correctness are provided when using such features.

3.1.1 Formalizing the Translation

The Type System

Spi calculus terms are untyped, which enables, during formal verification, to find type flaw attacks, i.e. attacks that are based on type confusion. However, since Java is statically typed, in order to enable the former language to be translated to the latter, it is necessary to assign a static type to every term used in a spi calculus specification. Types can be inferred automatically to some extent, so that the user work is minimized. However, unfortunately, it is not possible to reuse existing type systems for the spi calculus, because they have different purposes. For example, in [37], a generic type system is developed in order to describe some process behavior properties, such as deadlock-freedom or race-freedom (which, by the way, are not so related to some common security properties, such as secrecy or authentication). It turns out that, for our purposes, the type system in [37] is even too much expressive about program behavior, but it does not assign static types to terms, thus being useless for our purposes.

The type system and associated type inference algorithm developed in this work recall some standard type systems for the λ-calculus, such as the one in [52]. Essentially, the

type system allows the type of a term to be inferred by looking at the context where that term is used. The type system relies on a set of known, hierarchically related (by the subtype relation <:) types, depicted in figure 2.7. $Message$ is the top type, representing any message, so that every term has type $Message$. The types that directly descend from $Message$ correspond to different forms of spi calculus terms. Then, each of these types can be specialized based on particular usages; for example the $Name$ type is extended, among others, by the *Channel* type, to handle those circumstances where a spi calculus name is in fact used as a channel, or by the $Nonce$ type when a fresh name is generated to be used as a use-once number. The user is allowed to extend the given type hierarchy, by adding more specialized types.

In order to formalize the type system, a typing context Γ is defined as a set containing type assignments of the form $x : A$, where x ranges over names and variables and A over types. The judgment $\Gamma \vdash M : A$ means that, within the typing context Γ, which must contain type assignments for all the free variables and free names of M, M is well formed and must have type A (it can still have subtypes of A). The judgment $\Gamma \vdash P$ means that process P is well formed in the typing context Γ. Note that the proposed type system does not assign types to processes, but only to terms. Indeed, in order to enable translation into Java, it is sufficient to assign a static type to each term, because terms are translated into Java typed data, while this is not needed for processes, which are translated into sequences of Java statements. For P to be well formed within Γ, it is necessary (but not sufficient) that all of its free names and free variables appear in Γ. As it will be clear later on, given a generic spi calculus process P, it may not be possible to find Γ such that P is well formed. Since our translation function only translates well formed processes, it turns out that only the set *Spi* of well formed processes, which is a subset of all spi calculus processes, is translated by our function. Note that this constraint does not alter the Dolev-Yao attacker model. Indeed, during formal verification of a (well formed) process, the attacker is still modeled as the (possibly non well formed) environment.

The typing rules are reported in figure 3.1. For brevity, only some significant rules are commented.

The (T-Pair) rule states that if M_1 has type A_1 and M_2 has type A_2, then it is possible to state that the pair (M_1,M_2) has type $A_1 \times A_2$. Note that this formalization keeps, for each pair, information on the types of the contained items. A possible Java implementation of this feature can be obtained by using Java generic types, introduced in Java 5. Otherwise, a simple non-generic *Pair* type could be used. The two implementations, with and without generics, can be shown to be equivalent in this context, as discussed later.

The (P-Match) rule states that if M and N are two well formed messages, and P is a well formed process, then the process that matches M and N, and then behaves like P, is well formed. Note that the (P-Match) rule does not require M and N to have the same type: we consider well formed a process where M and N are matched, even if they have different, incompatible types (e.g. M is typed as $Name$ and N as $Hash\langle Message\rangle$). This is not an issue, because when the desired security properties are checked against the untyped spi calculus, type flaw attacks are taken into account, so even an erroneous Java implementation that would consider equal two terms with incompatible types, would have been considered in the formal verification step. By the way, adding to (P-Match) the

$$\frac{\Gamma \vdash M : A_1 \qquad A_1 <: A_2}{\Gamma \vdash M : A_2} \text{ (Subs)} \qquad \frac{}{\Gamma \vdash \mathbf{0}} \text{ (P-Nil)}$$

$$\frac{\Gamma \vdash M : Channel \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\langle N \rangle .P} \text{ (P-Out)}$$

$$\frac{\Gamma \vdash M : Channel \qquad \Gamma,x : A \vdash P}{\Gamma \vdash M(x).P} \text{ (P-In)}$$

$$\frac{\Gamma \vdash M : A_1 \times A_2 \qquad \Gamma,x : A_1,y : A_2 \vdash P}{\Gamma \vdash let\ (x,y) = M\ in\ P} \text{ (P-PSplit)}$$

$$\frac{\Gamma \vdash M : Message \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash [M\ is\ N]\ P} \text{ (P-Match)}$$

$$\frac{\Gamma,n : A \vdash P \qquad A <: Name\ \wedge\ A \not<: Channel}{\Gamma \vdash (\nu n)P} \text{ (P-Restr)}$$

$$\frac{\Gamma \vdash M : ShKeyC\langle A_1 \rangle \qquad \Gamma \vdash K : ShKey\langle A_2 \rangle \qquad \Gamma,x : A_1 \vdash P}{\Gamma \vdash case\ M\ of\ \{x\}_K\ in\ P} \text{ (P-ShKD)}$$

$$\frac{\Gamma \vdash L : PubKC\,\langle A \rangle \qquad \Gamma \vdash N : PriK \qquad \Gamma,x : A \vdash P}{\Gamma \vdash case\ L\ of\ \{[x]\}_N\ in\ P} \text{ (P-PubKD)}$$

$$\frac{\Gamma \vdash L : PriKC\,\langle A \rangle \qquad \Gamma \vdash N : PubK \qquad \Gamma,x : A \vdash P}{\Gamma \vdash case\ L\ of\ [\{x\}]_N\ in\ P} \text{ (P-PriKD)}$$

$$\frac{\Gamma \vdash M : Integer \qquad \Gamma \vdash P \qquad \Gamma,x : A \vdash Q \quad A <: Integer}{\Gamma \vdash case\ M\ of\ 0 : P\ suc(x) : Q} \text{ (P-IntCase)}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (T-NameVar)} \qquad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1,M_2) : A_1 \times A_2} \text{ (T-Pair)}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash H(M) : Hash\,\langle A \rangle} \text{ (T-Hash)}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M^{\sim} : ShKey\langle A \rangle} \text{ (T-ShKey)}$$

$$\frac{\Gamma \vdash M : A_1 \qquad \Gamma \vdash K : ShKey\langle A_2 \rangle}{\Gamma \vdash \{M\}_K : ShKeyC\langle A_1 \rangle} \text{ (T-ShKC)}$$

$$\frac{\Gamma \vdash M : KeyPair}{\Gamma \vdash M^+ : PubK} \text{ (T-PubK)} \qquad \frac{\Gamma \vdash M : KeyPair}{\Gamma \vdash M^- : PriK} \text{ (T-PriK)}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : PubK}{\Gamma \vdash \{[M]\}_K : PubKC\,\langle A \rangle} \text{ (T-PubKC)} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : PriK}{\Gamma \vdash [\{M\}]_K : PriKC\,\langle A \rangle} \text{ (T-PriKC)}$$

$$\frac{}{\Gamma \vdash 0 : Integer} \text{ (T-Zero)} \qquad \frac{\Gamma \vdash M : Integer}{\Gamma \vdash suc(M) : Integer} \text{ (T-Succ)}$$

Figure 3.1: Typing rules.

constraint stating that the type of M must equal the type of N would also be possible, though probably too rigid.

In the (P-ShKD) rule, note that the third premise requires the variable x to be added to the typing context, because x appears free in P, thus satisfying the necessary condition on Γ. The same reasoning holds for the first premise of (P-Restr). In the latter rule, however, it is also required that $A <: Name\ \wedge\ A \not<: Channel$. This is motivated because the restriction process generates a fresh *name*, and not a fresh term. The additional requirement avoids fresh channels to be created because they are essentially useless under our

assumptions of sequential agent behavior. Indeed, a channel is useful for communication only if it is known by more than one sequential agent.

A minor difference between the typing rules given in figure 3.1 and the types depicted in figure 2.7 concerns $Integer$ handling. In plain spi calculus, 0 is the only “integer number”, and the $suc(M)$ is the logical successor of any term. In order to faithfully describe this matter, figure 2.7 uses two different types: $Integer$, whose only value is the 0 term; and $Successor\langle A\rangle$, whose values are all terms of the form $suc(M)$. However, considering the goal of deriving an implementation from a spi calculus model, it is not trivial to find a general way to implement the logical successor of any arbitrary term. For example, there is no standard way of creating the successor of a channel, or the successor of a pair. In fact, real protocols almost always use successors only to express integer numbers. Based on these considerations, the typing rules in figure 3.1 force both 0 and $suc(M)$ to be $Integer$, and the $Successor\langle A\rangle$ type is never used (or alternatively, this may be expressed by using the $Successor\langle A\rangle$ type, and forcing $A <: Integer$). In practice, this means that the values of the $Integer$ type are all the natural integers, and that any $suc(M)$ term can only be a strictly positive integer. As a positive side effect, this also enables an efficient representation of integer numbers in Java: without these constraints on the type system, number n should be represented in memory by storing the whole stack composed of n successor terms, in order to be able to retrieve any of its arbitrary predecessors in the integer-case process; instead, with the given typing rules, number n can be stored in memory by using Java native types or objects, and any predecessor can be simply obtained by subtraction.

Note that the parametric typing rules of figure 3.1 ensure type safety in the same way Java 5 generic types do. In automatically generated code that starts from well-formed spi calculus processes, this means that using Java 5 generic types is redundant. If Java 5 generic types are not used, then some downcasts will be syntactically required, that could be avoided by instead using generic types. However, this kind of downcasts can never fail, because the parametric typing rules of figure 3.1 actually provide type safety.

This rather standard type system shares some common properties with other well known type systems [52]. In particular, the canonical form lemma can be proven, stating that if a term M has a particular type A, then it can only assume the particular values of A. Moreover, it can be shown that a type inference algorithm terminates finding the principal type for every term, if it is possible to find one.

As hinted above, not every spi calculus process is well-formed (that is, has a complete type derivation tree, where each leaf is an axiom) in the proposed typing system. For example, the following process

$$\overline{c}\langle\{M\}_{H(N)}\rangle.0$$

is not well formed. Reconstructing its derivation tree is impossible, because first the rule (T-ShKD) forces $H(N)$ to have type $ShKey\langle B\rangle$ (for some $B <: Message$), so then the (T-Hash) rule cannot be applied, making the derivation stuck.

Finally, the user can manually refine the type of a particular term. A user-provided type refinement for a given term c can be represented by adding a custom downcast rule, only valid for term c, to the type system. This is needed, because some types (e.g. the

subtypes of *Channel*) cannot be inferred automatically only on the basis of their usage. For example, if the user wants to specify that c is a *Tcp/Ip Channel*, then the following rule is added.

$$\frac{\Gamma \vdash c : Channel \qquad Tcp/Ip\ Channel <: Channel}{\Gamma \vdash c : Tcp/Ip\ Channel}$$

Note that the properties of the type system are still preserved even when custom downcasts are added, thanks to the premises of the downcast rules. Indeed, the first premise ensures that the downcast is performed only if the type inference algorithm can infer that c must have type *Channel*; the second rule ensures that the downcast required by the user is coherent with the type hierarchy.

The Translation Function

The translation from spi calculus to Java is formalized by a set of functions, each dealing with a particular aspect of the translation. All of these functions operate on well formed spi calculus processes and terms, that is processes and terms for which a type derivation tree can be found.

Each sequential spi calculus process, typically representing one of the protocol roles, is translated into a sequence of Java statements implementing the spi calculus process. These statements are embedded into a `try` block, followed by `catch` and `finally` blocks, which are in turn embedded into a method, that is invoked when a protocol run is requested. All the Java code surrounding the generated statements is called here the "context". The generated method will have one input parameter for each free variable and free name of the spi calculus process. For example, the $\overline{c}\langle M\rangle .c(x).\mathbf{0}$ process has a free name c and a free variable M: the generated Java method will have two input parameters, because it is assumed that the user will provide sensible values for these two terms.

Without proper encapsulation, translating a spi calculus process into Java would generate complex and non modular code. This complexity would make it difficult to show that the generated code correctly refines the spi calculus process. Indeed, all the implementation details that are abstracted away in spi calculus must be explicitly handled in Java. For example, every spi calculus encryption implies the creation and initialization of a Java `Cipher` object from the Java Cryptography Architecture (JCA). This object must be fed with data to be encrypted, and finally the resulting encryption can be obtained. Analogously, sending data over a channel requires direct handling of sockets or of other data structures, depending on the underlying medium for that channel.

In this work, the aforecited SpiWrapper Java library is leveraged to provide the required encapsulation. In this chapter its properties and interfaces are formally defined, so that a sound translation from spi calculus to Java can be obtained. The SpiWrapper library encapsulates some implementation complexity, allowing the generated code to be modular and easy to be mapped back to the original spi calculus specification. By defining a formal semantics for the intended behavior of this library, formal verification of the generated code is modularized too. One goal is to check that the generated code correctly refines the spi calculus specification, by assuming that the used SpiWrapper library behaves as

specified. Another independent goal is to provide a formally verified implementation of such a library.

The SpiWrapper library offers an abstract Java class for each type depicted in figure 2.7. Each abstract class implements the operations that can be performed on the corresponding type. For instance, the abstract class `Pair<A,B>` offers the methods `A getLeft();` and `B getRight();` that retrieve the first and second items of the pair, allowing the pair splitting process to be implemented.

Each SpiWrapper class is abstract, because only the internal data representation is handled, while the marshaling functions, encoding the internal representation into the external one that is sent over the network and vice versa, are declared, but not implemented. This enables the user to define her own marshaling layer, by extending the abstract class, and implementing the marshaling and unmarshaling functions. It could be argued that letting the user implement the marshaling functions could introduce security flaws that were not present in the abstract model. This problem is specifically addressed in section 3.2, where it is shown that if some static checks on the user-written code are performed, such possible flaws are avoided.

Let *Spi* be the aforecited set of well formed spi calculus processes, *SpiTerm* the set of well formed terms, and $Java$ the set of strings representing sequences of Java statements. Then, the function $tr_p : Spi, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ generates the Java statements for the spi calculus process given as its first parameter. In Java, all variables must be declared and initialized before they can be used. The second parameter of tr_p, let us call it $built \in 2^{SpiTerm}$, traces the well formed terms that have already been declared and initialized in the Java code. Moreover, some value should be returned after a successful protocol run (e.g. a negotiated shared secret, or a secure session id). The third parameter of tr_p, let us call it $return \in 2^{SpiTerm}$, contains the well formed terms that must be returned if a protocol run ends successfully. In other words, the *return* set is populated by those terms having the `return` attribute set to `true` in the eSpi document. In order to return the desired values to the user, a Java object declared as `Map<String,Message> _return` is maintained, that maps the Java name of every term to be returned onto its value (i.e. the Java object that implements it). The map is filled as long as the values to be returned become available.

Before showing the definition of tr_p, some auxiliary functions are introduced. $ub : SpiTerm, 2^{SpiTerm} \rightarrow 2^{SpiTerm}$ updates the *built* set taken as second parameter, by adding to it a term M, taken as first parameter, and its subterms. Formally, *ub* is defined as

$$ub(M, built) = built \cup subterms(M)$$

where $subterms(M)$ is the set containing M and all its subterms.

$ret : SpiTerm, 2^{SpiTerm} \rightarrow Java$ generates the Java code that fills the `_return` map. It puts the reference to the Java object that represents the term that is passed as its first parameter into the map, if it is in the set of terms that must be returned when the protocol run ends. This set is the second parameter. For every function that returns Java code, the following typographical conventions are used: the returned text is quoted by double quotes; inside the quoted text, *italic* is used for functions that return text, while `courier`

is used for verbatim returned text. The *ret* function is formally defined as

$$ret(M,return) = \text{“”, if } M \notin return$$
$$\text{“}\texttt{_return.put("}J(M)\texttt{",}J(M)\texttt{);}\text{”, otherwise}$$

where $J(M)$ is a bijection that gives the name of the Java variable for term M, by mangling it.

$tr_t : SpiTerm, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ takes a term M, the *built* set and the *return* set, and generates the Java code that "builds", that is declares and initializes, the Java variable $J(M)$ for the given spi calculus term M, if it has not yet been built.

The tr_t function is formally defined in figure 3.2. Some interesting cases are explained in details. All declared Java variables are actually also marked as `final`, which however is omitted here for brevity. For every term M, if M is already in the *built* set, then no code is generated, because the Java variable has already been declared and initialized. The $T(M)$ function returns the inferred type for the term M, which corresponds to one of the SpiWrapper abstract classes. The $Ts(M)$ function returns instead the name of the concrete user-provided Java class implementing the marshaling functions and extending the class returned by $T(M)$. Finally the $Param(M)$ function returns some user-defined parameters needed to make the protocol interoperable. The type and number of such parameters depend on the type of M, and they are specified in the eSpi specification XML document; the value of such parameters is specified into the eSpi document, specifically in the `term` element for M. For example, if M takes the form $N^{\sim}$ and is thus typed as $ShKey\langle A \rangle$ (a shared key wrapping some key material of type A), then the eSpi specification XML document states that the parameters are the key length, the key type and the desired JCA provider; while the eSpi document contains the actual values for term $N^{\sim}$ of such parameters. Nevertheless, the reader should not be distracted by interoperability details now, whereas the main focus is formally linking spi calculus processes to sequences of Java statements.

In the name n case, the code emitted by the *ret* auxiliary function is appended to the generated code, so that, if n is to be returned, it is added to the `_return` map.

In the pair (M,N) case, first tr_t is invoked on M and N, to ensure they are built. Note that, by invoking $tr_t(N,ub(M,built),return)$, N is built by taking into account that M and all its subterms have already been built, so they are not built twice. For example, if $M = (a,b)$ and $N = (b,c)$, then b is built when M is built, and it must not be built again when N is built. Once M and N are built, tr_t appends the code that actually builds the pair, and the (possibly empty) code needed to return the pair. Note that it is only needed to explicitly call *ret* on the pair, and not on its components, because the recursive invocations of tr_t on the components already ensure they are added, if needed, to the `_return` map as soon as they are built.

In the shared key ciphered $\{M\}_N$ case, first M and N are built, then the shared key ciphered object is instantiated and assigned to the variable named $J(\{M\}_N)$, and the *ret* function is invoked.

Finally, in the public key K^+ case, the `T getPublic(Class<T>,...)` method is invoked, that encapsulates the public part of the keypair K, in the given class. Same reasoning applies to the private key K^- case.

$tr_t(M,built,return)$ = "", if $M \in built$
$tr_t(n,built,return)$ = "$T(n)$ $J(n)$ = new $Ts(n)$($Param(n)$); $ret(n,return)$"
$tr_t((M,N),built,return)$ = "$tr_t(M,built,return)$ $tr_t(N,ub(M,built),return)$
  $T((M,N))$ $J((M,N))$ = new $Ts((M,N))$($J(M)$,$J(N)$, $Param((M,N))$);
  $ret((M,N),return)$"
$tr_t(K^{\sim},built,return)$ = "$tr_t(K,built,return)$
  $T(K^{\sim})$ $J(K^{\sim})$ = new $Ts(K^{\sim})$($J(K)$,$Param(K^{\sim})$); $ret(K^{\sim},return)$"
$tr_t(\{M\}_N,built,return)$ = "$tr_t(M,built,return)$ $tr_t(N,ub(M,built),return)$
  $T(\{M\}_N)$ $J(\{M\}_N)$ = new $Ts(\{M\}_N)$($J(M)$, $J(N)$,$Param(\{M\}_N)$);
  $ret(\{M\}_N,return)$"
$tr_t(H(M),built,return)$ = "$tr_t(M,built,return)$
  $T(H(M))$ $J(H(M))$ = new $Ts(H(M))$($J(M)$,$Param(H(M))$);
  $ret(H(M),return)$"
$tr_t(K^+,built,return)$ = "$tr_t(K,built,return)$
  $T(K^+)$ $J(K^+)$ = $J(K)$.getPublic($Ts(K^+)$.class,$Param(K^+)$);
  $ret(K^+,return)$"
$tr_t(K^-,built,return)$ = "$tr_t(K,built,return)$
  $T(K^-)$ $J(K^-)$ = $J(K)$.getPrivate($Ts(K^-)$.class,$Param(K^-)$);
  $ret(K^-,return)$"
$tr_t(\{[M]\}_N,built,return)$ = "$tr_t(M,built,return)$ $tr_t(N,ub(M,built),return)$
  $T(\{[M]\}_N)$ $J(\{[M]\}_N)$ = new $Ts(\{[M]\}_N)$($J(M)$,$J(N)$,$Param(\{[M]\}_N)$);
  $ret(\{[M]\}_N,return)$"
$tr_t([\{M\}]_N,built,return)$ = "$tr_t(M,built,return)$ $tr_t(N,ub(M,built),return)$
  $T([\{M\}]_N)$ $J([\{M\}]_N)$ = new $Ts([\{M\}]_N)$($J(M)$,$J(N)$,$Param([\{M\}]_N)$);
  $ret([\{M\}]_N,return)$"
$tr_t(0,built,return)$ = "$T(0)$ $J(0)$ = new $Ts(0)$(0,$Param(0)$); $ret(0,return)$"
$tr_t(suc(M),built,return)$ = "$tr_t(M,built,return)$
  $T(suc(M))$ $J(suc(M))$ = new $Ts(suc(M))$($J(M)$,$Param(suc(M))$);
  $ret(suc(M),return)$"

Figure 3.2: Definition of the tr_t function.

Note that no case is available for variables. Indeed, variables cannot be "built", rather, they are declared and assigned by the code implementing spi calculus processes that bind variables.

Now that all the auxiliary functions have been defined, the formal definition of tr_p is given in figure 3.3. Some interesting cases are explained in details. Like for tr_t, all declared variables are also marked as `final`, though not shown here. Moreover, the translated spi calculus process is also printed as a Java comment to improve readability of the generated code (not shown here).

When translating the output process, first N is built, then the `send` method is invoked on the channel referenced by $J(M)$, passing $J(N)$ as its argument, so that it is sent over

$tr_p(\mathbf{0},built,return) =$ “”
$tr_p(\overline{M}\,\langle N\rangle\,.P,built,return) =$ “$tr_t(N,built,return)$
  $J(M)$`.send(`$J(N)$`);`
  $tr_p(P,ub(N,built),return)$”
$tr_p(M(x).P,built,return) =$
  “$T(x)$ $J(x)$ `=` $J(M)$`.receive(`$Ts(x)$`.class,`$Param(x)$`);` $ret(x,return)$
  $tr_p(P,ub(x,built),return)$”
$tr_p((\nu n)P,built,return) =$ “$tr_t(n,built,return)$ $tr_p(P,ub(n,built),return)$”
$tr_p([M\ is\ N]P,built,return) =$ “$tr_t(M,built,return)$ $tr_t(N,ub(M,built),return)$
  `if (!`$J(M)$`.equals(`$J(N)$`)) { throw new MatchException(); }`
  $tr_p(P,ub(M,built) \cup ub(N,built),return)$”
$tr_p(let\ (x,y) = M\ in\ P,built,return) =$ “$tr_t(M,built,return)$
  $T(x)$ $J(x)$ `=` $J(M)$`.getLeft();` $ret(x,return)$
  $T(y)$ $J(y)$ `=` $J(M)$`.getRight();` $ret(y,return)$
  $tr_p(P,ub(M,built) \cup ub(x,built) \cup ub(y,built),return)$”
$tr_p(case\ L\ of\ \{x\}_N\ in\ P,built,return) =$ “$tr_t(L,built,return)$ $tr_t(N,ub(L,built),return)$
  $T(x)$ $J(x)$ `=` $J(L)$`.decrypt(`$J(N)$`,`$Param(L)$`);` $ret(x,return)$
  $tr_p(P,ub(L,built) \cup ub(N,built) \cup ub(x,built),return)$
$tr_p(case\ L\ of\ \{[x]\}_N\ in\ P,built,return) =$ “$tr_t(L,built,return)$ $tr_t(N,ub(L,built),return)$
  $T(x)$ $J(x)$ `=` $J(L)$`.decrypt(`$J(N)$`,`$Param(L)$`);` $ret(x,return)$
  $tr_p(P,ub(L,built) \cup ub(N,built) \cup ub(x,built),return)$”
$tr_p(case\ L\ of\ [\{x\}]_N\ in\ P,built,return) =$ “$tr_t(L,built,return)$ $tr_t(N,ub(L,built),return)$
  $T(x)$ $J(x)$ `=` $J(L)$`.decrypt(`$J(N)$`,`$Param(L)$`);` $ret(x,return)$
  $tr_p(P,ub(L,built) \cup ub(N,built) \cup ub(x,built),return)$”
$tr_p(case\ M\ of\ 0 : P\ suc(x) : Q,built,return) =$ “$tr_t(M,built,return)$
  `if (`$J(M)$`.isSpiZero()) {`
    $tr_p(P,ub(M,built),return)$
  `} else {`
    $T(x)$ $J(x)$ `=` $J(M)$`.getPrevious(`$Ts(x)$`.class,`$Param(x)$`);` $ret(x,return)$
    $tr_p(Q,ub(M,built) \cup ub(x,built),return)$
  `}`”

Figure 3.3: Definition of the tr_p function.

the channel. As M is enforced to be a free name by the type system, there is no need to build it.

For the input process, the `T receive(Class<T>,...);` method is invoked. The arguments of this method allow it to create an instance of $Ts(x)$, fill it with the received data, and return its reference, that is assigned to the Java variable $J(x)$. Again, being a channel, term M is not built, because the type system enforces it to be a free name.

In the decryption process the decryption key is passed as argument. If `ShKC<T>` is the type of $J(L)$, the `decrypt` method returns a newly created object of type `T` containing the

decrypted data.

In the restriction process, the name n is built, then the rest of the process is translated; when n is built, a constructor is called, which generates the Java implementation of a new fresh name of the expected type.

With the pair splitting process, first M is built, then the x and y variables are declared and assigned, which corresponds to the spi calculus binding of a variable. Note that when the following P process will then be translated, M, x and y will all have already been built; indeed, x and y are not built by the `new` operator, rather, being variables, they are assigned the value (Java reference) of another term.

When translating the match case, after having built M and N, if they are not equal, execution is stopped by throwing an exception, which is handled by the context, else the match is successful, and execution can continue with the translation of the P process, where both M and N are marked as built.

All processes presented here can easily be extended to the case where an `else` branch is taken if the process fails. In fact, the implementation of the tr_p function in the Spi2Java tools supports `else` branches.

The context handles the exception by setting the `_return` map to `null`, thus simulating a stuck spi calculus process (a successful run of a protocol that does not need to return any value, still returns an empty map, and not `null`). Note that when a message is received from a channel, or a plaintext is reconstructed from a ciphertext, a new SpiWrapper object holding the obtained data must be created, even if the corresponding spi calculus term is already instantiated in another Java object. For example, the Java code implementing the spi calculus process $\overline{c}\langle M\rangle.c(x).\mathbf{0}$, will store one object for the M term, and one different object for the received x term. It may happen, however, that x is assigned the same value of M (simulating that the spi calculus process receives exactly the M term back), although they are two different objects. For this reason, equality of objects cannot be checked by means of reference equality, but the `equals` method must check if the value of the two objects is the same. Using singleton instances to represent spi calculus terms, and thus letting the match case check for reference equivalence, would also be possible, but it would not be better. Indeed, in the `receive` (`decrypt`) method, it would be necessary to check the content of received (decrypted) data, to decide if their representing singleton is already instantiated or not.

In the integer case, three cases are possible: if M is 0 then P is executed, if M is $suc(N)$, then $Q[^N/_x]$ is executed, else the process is stuck. However, note that the type system enforces M to be typed as $Integer$, so, by the canonical form lemma, M must be either 0 or $suc(N)$, where N is typed as $Integer$ too. For this reason, the third case, where the process is stuck, cannot happen at run-time, and only the first two cases are handled by the `if-else` Java statement. In fact, this is also the way the Spi2Java code generator tool handles `else` branches for other spi calculus processes.

Handling channels (e.g. TCP/IP channels), and in general all Java resources that need to be allocated before usage and disposed after usage, requires a little more effort. Thanks to the type system, all channels are forced to be free names, which means they are input parameters of the method implementing the spi calculus code. In turn, this means that channels are already built (declared and initialized) and opened in the context. For scope

reasons, other disposable resources that are not input parameters shall be declared before the `try` block, and initialized and opened when necessary within the generated code. All disposable resources that have been opened and that must not be returned to the caller shall be closed before the method returns. This is done in the `finally` block of the method, ensuring disposable resources are closed anyway (and this is the reason why they have to be declared outside the `try` block). Of course, the formal translation functions are modified so that disposable resources are just assigned, and not declared, when they are built, which simply amounts to omit their type when assigning them. So, the skeleton of the generated method looks like

```
Map<String,Message> generatedSpi(@InputParams@) {
  Map<String,Message> _return = new TreeMap<String,Message>();
  @DisposableResourcesDeclaration@
  try { @GeneratedSpiImpl@ }
  catch (SpiWrapperException e) { _return = null; }
  finally { @CloseDisposableResources@ }
  return _return;
}
```

The `@InputParams@` placeholder gets substituted by the free variables and free names of the translated spi calculus process. `@DisposableResourcesDeclaration@` gets substituted by the Java declaration of the variables holding references to the disposable objects (not already declared as input parameters, like it happens instead for channels). `@GeneratedSpiImpl@` gets substituted by the generated Java code implementing the core spi calculus process being implemented, and the `@CloseDisposableResources@` placeholder gets substituted by the code closing disposable resources that shall not be returned by the caller, that is whose reference is not stored within the `_return` map. The `SpiWrapperException` class extends the standard Java `Exception` class, and captures all subtypes of exceptions that can be thrown by the SpiWrapper library.

3.1.2 Soundness

The formal definitions of the translation functions tr_p, tr_t, ub and ret allow some properties about the generated code to be stated. In order to enable the formal proofs of these properties, some reasonable assumptions are made explicit below.

- For any term M, it is assumed that the relation $Ts(M) <: T(M)$ holds, which can be enforced by checking that the user-provided implementation of the marshaling functions is done by extending the appropriate SpiWrapper abstract classes.

- For every constructor c offered by the SpiWrapper class $T(M)$, it is assumed that a constructor c' exists in the user-provided class $Ts(M)$ that extends $T(M)$. The c' constructor is assumed to have the same parameters of c and to be implemented only by a call to the `super` method. Unfortunately, Java does not support constructor inheritance, so the $Ts(M)$ class is not syntactically forced to have such constructor, and the existence of this constructor must be checked by other means.

- It is assumed that $Param(M)$ returns the correct number and type of user-provided interoperability parameters, according to the content of the eSpi specification XML document and the eSpi document.

Given a well formed spi calculus process P, let $t(P) \in 2^{SpiTerm}$ be the set of all the terms in P. Under the assumptions made, the following theorem can be proven.

Theorem 1. *If* $\Gamma \vdash P$ *and* $return \subseteq t(P)$*, then* $tr_p(P,fnv(P),return)$ *is well formed.*

By "well formed" we mean a sequence of Java statements that, put in the context, forms a Java program that compiles, i.e. a Java program that is correct from a syntactic and static semantic point of view. Note that the terms in $fnv(P)$ are the protocol input parameters, so their corresponding Java variables are already declared in the context, and passed as method input parameters.

The full proof of theorem 1 is given in appendix A. Now, as an informal proof hint, consider for example the case where an output process $\overline{M}\langle N\rangle.Q$ is translated into Java. First, it is shown that the tr_t function declares and assigns, if needed, the N term (while M must be a free name because of the type system, so it is already declared and initialized in the context). Then, it is shown by inspection that the code implementing the output process is actually well formed. Finally, it is shown by induction that the code implementing the Q process is well formed, letting the whole generated code be well formed.

Now the semantic properties of the generated Java code are discussed. The main goal is to show that, under some assumptions on the behavior of the SpiWrapper classes, the security properties verified on the spi calculus abstract specification are preserved by the generated Java implementation. Since a Dolev-Yao attacker is considered, we focus on safety security properties that can be defined by means of a predicate that must hold for all traces of a protocol. For example, secrecy and authentication are safety properties. In a Dolev-Yao context, liveness properties cannot be proven, because the attacker is always able to drop messages.

In order to prove security properties preservation from spi calculus to Java, it is shown that the generated Java code simulates the corresponding spi calculus process. That is, for each trace that can be executed in the Java domain, the same trace exists in the spi calculus domain. As a corollary, the Java traces are a subset of the spi calculus traces. Finally, if a spi calculus specification is proven secure against a safety property, it means that all of its traces are safe, and so the subset of Java traces is. Technically, a weak simulation relation between the generated Java code and the corresponding spi calculus is shown. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a concrete process, but each process is still allowed to perform any internal step in between two external states. More details about the weak refinement used here can be found, for example, in [65].

In order to state and prove the simulation relation, the semantics of the spi calculus must be written in a slightly different way. According to the notation introduced in section 2.1, a transition can be in general written as

$$P \xrightarrow{\mathcal{L}} P'\sigma'$$

where $\mathcal{L}$ is the transition label, and σ' is a (possibly empty) substitution that binds variables. If we do not apply substitutions, but we leave them explicitly indicated as postfix operators, a generic state P of a system run will be written as $P_1\sigma$, where P_1 includes all the free variables of P, as well as all the bound variables that have been substituted in the past evolution that has led to P, while σ incorporates such substitutions. Using this representation for processes, a generic transition can be written as

$$P_1\sigma \xrightarrow{\mathcal{L}} P_1'\sigma\sigma'$$

and a state can be divided into two components: a process expression followed by a variable substitution.

In the LTS for the spi calculus, all states are defined as external.

An LTS for a Java sequential program obtained by our transformation function is now defined. In order to relate the Java behavior with the spi calculus behavior, the Java LTS uses the same abstract labels used for the spi calculus LTS. Let j be the Java code that is going to be executed, $JavaVar$ the set of identifiers that can be used as variables in Java programs, and $JavaObj$ the set of object identifiers. Then a generic state (j,Val,Res) is defined by the code j that is going to be executed, plus a partial function $Val : JavaObj \rightarrow SpiTerm$, mapping each Java object that has been created by previously executed code to the spi calculus term the object is implementing, and a partial function $Res : JavaVar \rightarrow JavaObj$ mapping each Java variable in the scope of j to the referenced Java object. For example, $Val(o) = \{M\}_N$ means that the Java object o implements the $\{M\}_N$ spi calculus term; $Res(\texttt{var}) = o$ means that the Java variable `var` references the object o. The intended invariant that should hold is $Val(Res(J(M))) = M\sigma$, where σ is the variable substitution in the corresponding spi calculus process. That is, the object referenced by the Java variable $J(M)$ must implement the $M\sigma$ term, which is the run-time value of the M spi calculus term. A Java state (j,Val,Res) is defined as external iff $j = tr_p(P,dom(Val \circ Res \circ J),return)$ for some spi calculus process P that does not begin with a restriction and for some return set $return$. Note that, since $Val \circ Res \circ J$ is a composition of partial functions, the domain of J is properly restricted such that its codomain matches the domain of Res; in turn this is restricted so that its codomain matches the domain of Val. The transitions of the form

$$j,Val,Res \xrightarrow{\mathcal{L}} j',Val',Res'$$

take from one generic state to another, following an abstract operational semantics for the Java language. In this work, we formally define an operational semantics that, if implemented by the SpiWrapper classes, makes it possible to have a weak simulation relation between the spi calculus process and the generated Java code. The formal semantics for the SpiWrapper classes presented in this work is reported in tables 3.1, 3.2 and 3.3.

A `void` method returns the *unit* value, while `true` and `false` are the boolean values; variable assignment evolves into the *unit* value, but its side effect is to map the variable to the assigned object, formally

$$T(x)\ J(x) = o,Val,Res \xrightarrow{\tau} unit,Val,\{(J(x),o)\} \cup Res$$

Name	Semantic Rule
P-Out	$\mathcal{c}.\mathtt{send(}\mathcal{M}\mathtt{)},\{(\mathcal{c},c),(\mathcal{M},M)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}\xrightarrow{c!M}\xrightarrow{\tau}{}^{*}$ $unit,\{(\mathcal{c},c),(\mathcal{M},M)\} \cup Val,Res$
P-In	$\mathcal{c}.\mathtt{receive(Ts.class,params)},\{(\mathcal{c},c)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}\xrightarrow{c?N}\xrightarrow{\tau}{}^{*}$ $\mathcal{N},\{(\mathcal{c},c),(\mathcal{N},N)\} \cup Val,Res$
T-Pair	$\mathtt{new\ PairMarsh(}\mathcal{A}\mathtt{,}\mathcal{B}\mathtt{,params)},\{(\mathcal{A},A),(\mathcal{B},B)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathcal{o},\{(\mathcal{A},A),(\mathcal{B},B),(\mathcal{o},(A,B))\} \cup Val,Res$
P-PSplit – left	$\mathcal{o}.\mathtt{getLeft()},\{(\mathcal{o},(M,N)),(\mathcal{M},M)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathcal{M},\{(\mathcal{o},(M,N)),(\mathcal{M},M)\} \cup Val,Res$
P-PSplit – right	$\mathcal{o}.\mathtt{getRight()},\{(\mathcal{o},(M,N)),(\mathcal{N},N)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathcal{N},\{(\mathcal{o},(M,N)),(\mathcal{N},N)\} \cup Val,Res$
P-Match – true	$M = N \Rightarrow \mathcal{a}.\mathtt{equals(}\mathcal{b}\mathtt{)},\{(\mathcal{a},M),(\mathcal{b},N)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathtt{true},\{(\mathcal{a},M),(\mathcal{b},N)\} \cup Val,Res$
P-Match – false	$M \neq N \Rightarrow \mathcal{a}.\mathtt{equals(}\mathcal{b}\mathtt{)},\{(\mathcal{a},M),(\mathcal{b},N)\} \cup Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathtt{false},\{(\mathcal{a},M),(\mathcal{b},N)\} \cup Val,Res$
P-Restr	$\mathtt{new\ TMarsh(params)},Val,Res$ $\xrightarrow{\tau}{}^{*}$ $\mathcal{o},\{(\mathcal{o},n)\} \cup Val,Res \ \wedge\ n \notin codom(Val)$

Table 3.1: Formal semantics of the SpiWrapper library – Part I.

Name	Semantic Rule
T-Hash	$\texttt{new HashMarsh}(\mathcal{M}\texttt{,params}),\{(\mathcal{M},M)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{N},\{(\mathcal{M},M),(\mathcal{N},H(M))\} \cup Val,Res$
T-ShKey	$\texttt{new ShKMarsh}(\mathcal{M}\texttt{,params}),\{(\mathcal{M},M)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{N},\{(\mathcal{M},M),(\mathcal{N},M^{\sim})\} \cup Val,Res$
T-ShKC	$\texttt{new ShKCMarsh}(\mathcal{M}\texttt{,}\mathcal{K}\texttt{,params}),\{(\mathcal{M},M),(\mathcal{K},K)\} \cup Val,Res \xrightarrow{\tau}{}^{*} o,\{(\mathcal{M},M),(\mathcal{K},K),(o,\{M\}_K)\} \cup Val,Res$
P-ShKD	$o\texttt{.decrypt}(\mathcal{K}\texttt{,params}),\{(\mathcal{K},K^{\sim}),(o,\{M\}_{K^{\sim}})\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{M},\{(\mathcal{M},M),(\mathcal{K},K^{\sim}),(o,\{M\}_{K^{\sim}})\} \cup Val,Res$
T-PubK	$\mathcal{K}\texttt{.getPublic(Ts.class,params)},\{(\mathcal{K},K)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{L},\{(\mathcal{K},K),(\mathcal{L},K^{+})\} \cup Val,Res$
T-PubKC	$\texttt{new PubKCMarsh}(\mathcal{M}\texttt{,}\mathcal{K}\texttt{,params}),\{(\mathcal{M},M),(\mathcal{K},K)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{N},\{(\mathcal{M},M),(\mathcal{K},K),(\mathcal{N},\{[M]\}_K)\} \cup Val,Res$
P-PubKD	$\mathcal{N}\texttt{.decrypt}(\mathcal{K}\texttt{,params}),\{(\mathcal{N},[\{M\}]_{K^{-}}),(\mathcal{K},K^{+})\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{M},\{(\mathcal{N},[\{M\}]_{K^{-}}),(\mathcal{K},K^{+}),(\mathcal{M},M)\} \cup Val,Res$

Table 3.2: Formal semantics of the SpiWrapper library – Part II.

Name	Semantic Rule
T-PriK	$\mathcal{K}$`.getPrivate(Ts.class,params)`$,\{(\mathcal{K},K)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{L},\{(\mathcal{K},K),(\mathcal{L},K^-)\} \cup Val,Res$
T-PriKC	`new PriKCMarsh(`$\mathcal{M}$`,`$\mathcal{K}$`,params)`$,\{(\mathcal{M},M),(\mathcal{K},K)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{N},\{(\mathcal{M},M),(\mathcal{K},K),(\mathcal{N},[\{M\}]_K)\} \cup Val,Res$
P-PriKD	$\mathcal{N}$`.decrypt(`$\mathcal{K}$`,params)`$,\{(\mathcal{N},\{[M]\}_{K+}),(\mathcal{K},K^-)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{M},\{(\mathcal{N},\{[M]\}_{K+}),(\mathcal{K},K^-),(\mathcal{M},M)\} \cup Val,Res$
T-Zero	`new IntMarsh(0,params)`$,Val,Res \xrightarrow{\tau}{}^{*} \mathcal{Z},\{(\mathcal{Z},0)\} \cup Val,Res$
T-Succ	`new IntMarsh(`$\mathcal{M}$`,params)`$,\{(\mathcal{M},M)\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{N},\{(\mathcal{M},M),(\mathcal{N},suc(M))\} \cup Val,Res$
P-IntCase – zero	$\mathcal{Z}$`.isSpiZero()`$,\{(\mathcal{Z},0)\} \cup Val,Res \xrightarrow{\tau}{}^{*}$ `true`$,\{(\mathcal{Z},0)\} \cup Val,Res$
P-IntCase – succ	$\mathcal{N}$`.isSpiZero()`$,\{(\mathcal{N},suc(M))\} \cup Val,Res \xrightarrow{\tau}{}^{*}$ `false`$,\{(\mathcal{N},suc(M))\} \cup Val,Res$
T-Previous	$\mathcal{N}$`.getPrevious(Ts.class,params)`$,\{(\mathcal{N},suc(M))\} \cup Val,Res \xrightarrow{\tau}{}^{*} \mathcal{M},\{(\mathcal{N},suc(M)),(\mathcal{M},M)\} \cup Val,Res$

Table 3.3: Formal semantics of the SpiWrapper library – Part III.

Generic types are not taken into account into the definition of the dynamic semantics, because in Java they are implemented by erasure, that is they are checked at compile-time, and then discarded in the compiled bytecode. For this reason the dynamic semantics can be defined without them. It is assumed that if a method cannot succeed (for example, a wrong key is passed to the `decrypt` method), it throws an exception, that simulates the stuck process. Formally, when an exception is thrown from any state (j,Val,Res), the following semantics is assumed

$$j,Val,Res \xrightarrow{\tau} \bot$$

where $\bot$ is a special non-external state that represents all the states where the control has passed to the context, due to a thrown exception. Standard congruence and computation semantic rules are assumed for the sequential concatenation of statements, and for the other Java statements.

The simulation relation S, that relates external spi calculus states to external Java states, is formally defined as

$$S(((\nu\overline{n})P)\sigma,(j,Val,Res)) \Leftrightarrow$$
$$j = tr_p(P,dom(Val{\circ}Res{\circ}J),return) \wedge \sigma_{|fnv(P)} = Val{\circ}Res{\circ}J_{|fnv(P)} \wedge \sigma \supseteq Val{\circ}Res{\circ}J$$

for any spi calculus process P that does not begin with a restriction, and any Val,Res such that $dom(Val \circ Res \circ J)$ is closed under the *subterms* function. Informally, a spi calculus state $((\nu\overline{n})P)\sigma$ and a Java state (j,Val,Res) are S-related, iff the Java state is external, and the invariant $M\sigma = Val(Res(J(M)))$ holds. Note that it is required that the domain of $Val \circ Res \circ J$ contains all the free names and variables in P; however some compound terms may not (yet) be stored in Java memory (because they will be built by the code generated by the $tr_t(\cdot)$ function): it is enough to require that the invariant holds for the already built terms, which are stored in Java memory.

Theorem 2. *If the SpiWrapper library behaves as specified in tables 3.1, 3.2 and 3.3, then, for any external state* (j',Val',Res')

$$S((\nu\overline{n})P\sigma,(j,Val,Res)) \;\wedge\; j,Val,Res \xrightarrow{\tau}{}^{*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau}{}^{*} j',Val',Res' \Rightarrow$$
$$P\sigma \xrightarrow{\mathcal{L}} (\nu\overline{m})P'\sigma' \;\wedge\; S((\nu\overline{n})(\nu\overline{m})P'\sigma',(j',Val',Res'))$$

Theorem 2 formally expresses the simulation relation between a spi calculus process and its corresponding generated Java program. If the simulation relation holds between a state of a spi calculus process and a state of its corresponding Java program, and if the Java program can evolve into a new external state, then the spi calculus process can evolve into a new external state too, and the new external states are still related by the simulation relation S.

The full proof of theorem 2 is given in appendix A. Now, as an informal proof hint, consider for example the case where a decryption process *case* L *of* $\{x\}_{N^{\sim}}$ *in* Q is implemented. On the Java side, it is first shown that the code generated by the $tr_t(\cdot)$ function, invoked first on L and then on $N^{\sim}$, executes by properly setting up the memory, namely

by creating, if needed, the Java objects such that $Val(Res(J(L))) = \{M\}_{N^\sim}\sigma$ (if code execution is going to succeed, which is the interesting case requiring the simulation relation to hold in the final state) and $Val(Res(J(N^\sim))) = N^\sim\sigma$. Then, it is shown that the rest of the code executes as

$$T(x)\ J(x) = J(L)\texttt{.decrypt(}J(N^\sim)\texttt{,}Param(L)\texttt{)},Val,Res \xrightarrow{\tau}{}^{*} \\ unit,\{(\mathcal{M},M)\} \cup Val,\{(J(x),\mathcal{M})\} \cup Res$$

On the spi calculus side, the process can evolve like

$$P\sigma = (case\ \{M\}_{N^\sim}\ of\ \{x\}_{N^\sim}\ in\ Q)\sigma \xrightarrow{\tau} Q\sigma\left[{}^{M}/{}_{x}\right]$$

So the same labeled transition happened in both systems, and the simulation relation S can be shown to still hold for the final states.

The initial state of a Java program could be an internal state, if the translated spi calculus process P begins with a restriction. However, it is formally shown in appendix A that the translation of a restriction process leads the Java program to an external state where the simulation relation S holds. So, even the translation of a restriction process is handled, enabling theorem 2, thus getting to the final result that the Java code simulates the spi calculus process from which it has been generated.

Thanks to the formal definition of the SpiWrapper library given in this work, formal verification of the generated code can be modularized in an assume-guarantee style. In particular, theorem 2 only deals with the Java code implementing the protocol logic, assuming that all low level details, such as dealing with the JCA or sockets, is correctly implemented. Providing a correct implementation of such details is an orthogonal verification problem that can be handled in isolation.

A bi-simulation relation cannot be proven, because it is not possible to show that for any spi calculus trace there exists a corresponding Java trace. As counter-example, consider a Java program stopping execution because of some low-level errors (e.g. wrong message marshaling, or using wrong cryptographic parameters) that are not caught by the spi calculus specification. In these cases the spi calculus trace can continue, while the Java program stops. If bi-simulation could be proven, more kinds of trace properties, like liveness ones, could be preserved from the spi calculus specification to the Java implementation. This is not a real limitation in fact, because such kinds of properties cannot be proven with a Dolev-Yao attacker anyway.

Verification of a SpiWrapper Implementation

As stated by theorem 2, the generated Java code simulates the spi calculus process from which it has been generated, by relying on the custom SpiWrapper library, whose behavior is formally specified. As a step further, an implementation of part of the SpiWrapper library is now presented, that is shown to be correct with respect to its specification reported in tables 3.1, 3.2 and 3.3. In order to reason on executions of the Java code implementing the library, the Middleweight Java (MJ) framework [21, 64] is used. Essentially, MJ specifies a small-step operational semantics for a rich subset of sequential Java. States are called *configurations*; a configuration is a four-tuple made of (H,VS,CF,FS), where

H is the heap, mapping object identifiers (oids) to their type and to a field function. A field function is a map from field names to values.

VS is the variable stack, mapping variable names to their type and to the referred oid or to their value.

CF is a closed frame of Java code to be evaluated.

FS is a frame stack, that is the program context in which CF is being evaluated.

In the original MJ, transitions are not labeled, because all side effects are captured by the subsequent configuration. In this work, transitions that produce input or output of message M on channel c, are labeled with $c?M$ and $c!M$ respectively, while all other transitions are labeled with τ.

A relation between Java states, defined as (j,Val,Res), and MJ configurations is defined, so that the two frameworks can be related. Indeed, states and MJ configurations are very similar, with MJ configurations storing more information, which is unneeded for SpiWrapper behavior specification. In fact, j corresponds to the MJ CF, that is the code to be evaluated; Res serves the same purpose as VS, although VS also stores some other information about variable scopes and types. Since we are interested in single method behaviors, rather than in full program behaviors, the FS context is set empty (denoted by []) before the execution of the method, and it will be empty after method executed. A note must be made on the relation between Val and MJ H, which completes the two frameworks relation. On one hand, Val maps oids to the implemented spi calculus terms; on the other hand, H maps oids to their internal state, which is the value of their fields. So, $Val(o) = M$ (read "object o implements spi calculus term M") *means* that $H(o) = v$ (read "object o has fields set as described by v") for some v. For example

$$\begin{aligned} Val(v_p) = (M,N) \quad \Leftrightarrow \quad & H(v_p) = \left(PairS, \begin{array}{l} \texttt{left} \;\to v_{\mathcal{M}} \\ \texttt{right} \to v_{\mathcal{N}} \end{array} \right) \wedge \\ & PairS <: Pair \wedge \\ & Val(v_{\mathcal{M}}) = M \wedge Val(v_{\mathcal{N}}) = N \end{aligned} \tag{3.1}$$

states that "object v_p implements the pair (M,N)" means that object v_p has a subtype of the $Pair$ type, and it has exactly two fields, namely `left` and `right`, pointing to two objects $v_{\mathcal{M}}$ and $v_{\mathcal{N}}$, implementing the M and N terms respectively. Note that the relation between Val and H is implementation dependent.

MJ does not support generic types, and there is no need to add such support, because, as explained before, generic types are implemented by erasure, so they can be disregarded when analyzing run-time behavior.

The `Pair` class is now verified. In the Spi2Java framework, the `Pair` class is abstract, meant to be extended by other classes implementing marshaling functions. For simplicity, since marshaling functions are irrelevant in this context, the `Pair` class is considered to be concrete, and marshaling functions are neglected.

Figure 3.4 shows a possible implementation of the `Pair` class that fits in the MJ framework.

```
package it.polito.spi2java.spiWrapper;

public class Pair extends Message {
  protected Message left;
  protected Message right;

  public Pair(Message left, Message right) {
    super();
    this.left = left;
    this.right = right;
  }

  public Message getLeft() {
    return this.left;
  }

  public Message getRight() {
    return this.right;
  }

  public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Pair) {
      Pair otherPair = (Pair) obj;
      boolean leftOK = this.getLeft().equals(otherPair.getLeft());
      result = leftOK && this.getRight().equals(otherPair.getRight());
    }
    return result;
  }
}
```

Figure 3.4: A possible implementation of the `Pair` class.

As an example, the rest of this section shows the proof of correctness of the constructor. Appendix A contains the proofs of correctness for the remaining `getLeft()`, `getRight()` and `equals()` methods.

Proof of correctness of the constructor. By looking up the (T-Pair) semantic rule, the initial state of the `Pair` constructor invocation is

$$\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{)},\{(\mathcal{A},A),(\mathcal{B},B)\},\emptyset$$

In the presented implementation, no additional marshaling parameters are required, so the '`,params`' argument can be neglected.

This state corresponds to the initial MJ configuration

$$(\underbrace{\{(\mathcal{A},(T_A,\mathbb{F}_A)),(\mathcal{B},(T_B,\mathbb{F}_B))\}}_{H_0},\underbrace{[]}_{VS_0},\underbrace{\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{);}}_{CF_0},\underbrace{[]}_{FS_0})$$

for some types T_A,T_B <: $Message$ and mapping functions $\mathbb{F}_A$,$\mathbb{F}_B$ such that the $\mathcal{A}$ and $\mathcal{B}$ objects implement the A and B spi calculus terms respectively.

The following steps lead to the final configuration. Note that all transitions are labeled with τ (which is omitted for brevity), because no input or output operations occur. In order to make the evaluation steps more readable, each transition is marked with the labels specified in [21].

$$\begin{array}{ll}
 & (H_0,VS_0,CF_0,FS_0) \\
\overset{\text{E-New}}{\rightarrow} & (\underbrace{H_0 \cup \{(o,\left(Pair,\begin{array}{l}\texttt{left} \rightarrow \texttt{null}\\ \texttt{right} \rightarrow \texttt{null}\end{array}\right))\}}_{H_1}, \\
 & \underbrace{\left(\left\{\begin{array}{l}\texttt{this} \rightarrow (o,Pair)\\ \texttt{left} \rightarrow (\mathcal{A},T_A)\\ \texttt{right} \rightarrow (\mathcal{B},T_B)\end{array}\right\} \circ []\right) \circ VS_0}_{VS_1}, \\
 & \texttt{super(); ...},(\texttt{return } o\texttt{;}) \circ FS_0) \\
\overset{\text{EC-Seq}}{\rightarrow} & (H_1,VS_1,\texttt{super();},\underbrace{(\texttt{this.left = left; ...}) \circ (\texttt{return } o\texttt{;}) \circ FS_0}_{FS_1}) \\
\overset{\text{EC-ExpState}}{\rightarrow} & (H_1,VS_1,\texttt{super()},FS_1) \\
\overset{\text{E-Super}}{\rightarrow} & (H_1,\{\texttt{this} \rightarrow (o_s,Message)\} \circ [] \circ VS_1,\{\},(\texttt{return } o_s\texttt{;}) \circ FS_1)
\end{array}$$

For brevity, the constructor of $Message$ is considered empty, instead of recursively invoking the constructor of $Object$, which is actually empty. In practice, considering the

constructor of *Object* would lead to the same evaluation steps, modulo one stack level.

$$
\begin{array}{cl}
\xrightarrow{\text{E-BlockElim}} & (H_1,[] \circ VS_1,\texttt{;},(\texttt{return } o_s\texttt{;}) \circ FS_1) \\
\xrightarrow{\text{E-Skip}} & (H_1,[] \circ VS_1,\texttt{return } o_s\texttt{;},FS_1) \\
\xrightarrow{\text{EC-ExpState}} & (H_1,[] \circ VS_1,\texttt{return } o_s,FS_1) \\
\xrightarrow{\text{E-Return}} & (H_1,VS_1,o_s,FS_1) \\
\xrightarrow{\text{E-Sub}} & (H_1,VS_1,\texttt{this.left = left; ...},(\texttt{return } o\texttt{;}) \circ FS_0) \\
\xrightarrow{\text{EC-Seq}} & (H_1,VS_1,\texttt{this.left = left;}, \\
& \underbrace{(\texttt{this.right = right;}) \circ (\texttt{return } o\texttt{;}) \circ FS_0}_{FS_2}) \\
\xrightarrow{\text{EC-FieldWrite1}} & (H_1,VS_1,\texttt{this},(\cdot\texttt{.left = left;}) \circ FS_2) \\
\xrightarrow{\text{E-VarAccess}} & (H_1,VS_1,o,(\cdot\texttt{.left = left;}) \circ FS_2) \\
\xrightarrow{\text{E-Sub}} & (H_1,VS_1,o\texttt{.left = left;},FS_2) \\
\xrightarrow{\text{EC-FieldWrite2}} & (H_1,VS_1,\texttt{left},(o\texttt{.left = }\cdot) \circ FS_2) \\
\xrightarrow{\text{E-VarAccess}} & (H_1,VS_1,\mathcal{A},(o\texttt{.left = }\cdot) \circ FS_2) \\
\xrightarrow{\text{E-Sub}} & (H_1,VS_1,o\texttt{.left = }\mathcal{A},FS_2) \\
\xrightarrow{\text{E-FieldWrite}} & (H_0 \cup \{(o,\left(Pair, \begin{array}{l}\texttt{left} \to \mathcal{A} \\ \texttt{right} \to \texttt{null}\end{array}\right))\},VS_1,\texttt{;},FS_2)
\end{array}
$$

By performing the same steps, the `right` field is assigned the $\mathcal{B}$ value, leading to the state

$$
\begin{array}{cl}
& (\underbrace{H_0 \cup \{(o,\left(Pair, \begin{array}{l}\texttt{left} \to \mathcal{A} \\ \texttt{right} \to \mathcal{B}\end{array}\right))\}}_{H_2},VS_1,\texttt{;},(\texttt{return } o\texttt{;}) \circ FS_0) \\
\xrightarrow{\text{E-Skip}} & (H_2,VS_1,\texttt{return } o\texttt{;},FS_0) \\
\xrightarrow{\text{E-Return}} & (H_2,VS_0,o,FS_0)
\end{array}
$$

Since in the final configuration the right side of (3.1) is true, this configuration corresponds to the final state

$$o,\{(\mathcal{A},A),(\mathcal{B},B),(o,(A,B))\},\emptyset$$

□

3.1.3 Discussion

The work in these sections defines a provably correct refinement from spi calculus specifications into Java code implementations, thus enabling automatic generation of the implementations, while preserving the security properties verified on the specifications. This refinement relation has been obtained by defining a type system, that allows to assign static types to the untyped spi calculus terms, and then to use the same types for Java data representing the spi calculus terms. Moreover, a translation function from well-formed sequential spi calculus processes to Java code has been formally defined, so that

it is possible to reason on the relation between the spi calculus source processes, and the generated Java code. The first result is that the translation of a well-formed spi calculus process always lead to the generation of well-formed Java code, that is code that compiles. Some features, like disposable resources, protocol return parameters and interoperability of the generated application, which is achieved by letting the user implement the marshaling functions, are also taken into account by the translation function.

As a further step, the generated Java code has been proven to be a correct refinement of the spi calculus specification in a modular way. First it has been shown that the generated Java code implementing the spi calculus protocol logic is correct, by assuming correctness of the underlying SpiWrapper Java library. In order to achieve this result, the formal definition of the intended behavior of the SpiWrapper library has been formalized. Then, it has been shown that the intended behavior of this library can be related to the formal semantics of the spi calculus, so that the generated Java code, by properly using the SpiWrapper library, can simulate the spi calculus specification.

Finally, it has been shown how an implementation of a class belonging to the SpiWrapper library can be verified correct with respect to the intended behavior. Correctness is proven by evaluating the Java code implementing the class within the MJ framework, and by relating that framework with the one presented here. This result increases confidence about the correctness of the whole system, because only correctness of standard libraries, like the JCA, is assumed, while the custom SpiWrapper library can be proven correct.

The translation function that has been formalized in this work has been implemented, in a richer version supporting `else` branches, in the Spi2Java tool, which has been used to successfully generate interoperable implementations of real cryptographic protocols. See chapter 4 for a description about using Spi2Java for the implementation of a client of the SSH Transport Layer Protocol.

It is believable that the extension of the results shown in these sections to other (statically typed or not) programming languages is straightforward.

The results presented here fit with the overall proposed MDD methodology, by providing a formal link between the spi calculus models and the corresponding Java code. This is in fact a step further towards the main goal of designing a fully formally-based MDD framework for security protocols. However, correctness of the generated implementation comes with a cost. Performances of the generated code may not be optimized, and manually modifying the code to increase its performances could compromise its security properties. Moreover, only new implementations of protocols can be obtained with model-driven-development, requiring some switching costs to substitute the legacy implementation with the newly generated one.

There are still open issues that would complete and improve this work. For instance, more classes belonging to the SpiWrapper library could be proven correct by using the same approach. Another possibility is to link this work to existing proposals [9] of cryptographic libraries that offer provably correct implementations of abstract cryptographic primitives like the ones used in the spi calculus.

3.2 Encodings

As previously shown, a significant amount of effort has been recently put in linking abstract formal models and their concrete implementations. Essentially, two main strategies arose, namely model extraction and code generation. In both strategies, the verified model is notably much more abstract than the refined implementations; yet, some low-level details such as data encoding are still retained in the verified models sometimes. On one hand, the abstracted details could in principle introduce security faults, if their abstraction from the implementation code is not proven sound; on the other hand, the low-level data encoding routines could be further abstracted away from the models, provided rigorous soundness conditions are provided.

In fact, one of the things that can be observed by looking at the results reported in [20], [40] and [18], is that the part of the extracted formal model that describes data encoding and decoding operations can be quite complex, much more complex than the abstract protocol model itself. This occurs even though in [20] and [18] the implementations of some low-level library operations, such as those for basic XML manipulation, are not included in the model but rather assumed to correctly refine their symbolic counterpart. Note that taking data transformations into account when a protocol implementation is analyzed is important, because the wrong implementation of such transformations may be responsible for security faults that can go undetected when they are abstracted away. For example, an incorrect implementation of a marshaling function could unwillingly leak secret data, or the function that encodes some data before applying a hash function to them could erroneously transform part of the data (e.g. a nonce) into a constant, thus enabling replay attacks. Note that these errors do not necessarily infringe interoperability, so they may be difficult to discover by testing.

In the code generation approach proposed here, and implemented in the Spi2Java framework, the user is responsible for manually writing the Java code implementing the encoding layer of the refined application. This manually written code is not linked with any part of the formally verified abstract model, and could in principle break the security properties proven on the model. Having sufficient conditions to be checked on this code, so that it can be safely abstracted in the verified model becomes essential in order to assess that the generated implementation is in fact a sound refinement of the abstract model.

The aim of the work presented in the following sections is to formally state and prove sufficient conditions under which the detailed models of data transformations, such as the ones extracted from protocol code in [18], or the ones manually implemented by a Spi2Java user, can be avoided and substituted by much simpler models or assumptions, that can be checked on sequential code and in isolation (i.e. without considering the behavior of the intruder), while obtaining the same kind of security assurance on the protocol implementation.

The first step in our methodology is the definition of simple formal models of data encoding and decoding transformations. Such models are general, that is they do not describe specific implementations, rather they capture only some general assumptions that are being made on implementations. Verifying a concrete protocol implementation can thus be reduced to verifying that the protocol implementation fulfills the assumptions

made about data encoding and decoding transformations and verifying the corresponding protocol model that includes these simple formal models of data encoding. This approach makes verification modular, according to an assume-guarantee style.

The second step that is made is to show that the models built in this way can be further simplified, using fault-preserving transformations like the ones introduced in [36]. The result that is finally obtained is that, under some further simple assumptions, the protocol models including data encoding can be simplified into abstract protocol models where data encoding is abstracted away, and the classical security faults (i.e. violations of secrecy and authentication) are preserved in this transformation. This means that, provided all the assumptions we made are verified on a given implementation, the formal model of the implementation details can be safely neglected in verifying the desired security properties, under the Dolev-Yao modeling approach.

It is worth noting that this kind of result can be exploited both when using the model extraction approach and when using code generation. In the former case, the assumptions made on data encoding and decoding transformations must be checked on the (sequential) code that implements them. If they hold, this code can be safely abstracted during model extraction.

In the latter case where the starting point is an already verified abstract protocol model, by the results given here, the code implementing the encoding layer can be safely generated automatically, even if it was not present in the abstract model, provided it satisfies the sufficient conditions stated in this work. Alternatively, like it happens for the Spi2Java framework, the user can manually fill the encoding layer implementations, and them some check on the provided code are performed, in order to ensure that it satisfies the required sufficient conditions.

The remainder of this section is organized as follows. Section 3.2.1 introduces the notation and the modeling approach, based on CSP, that is used to reason on security protocols in this context. Then, a distinction is made between two kinds of data encoding and decoding transformations, because they need different assumptions. Section 3.2.2 focuses on encoding and decoding transformations that are applied to the messages when they are sent and received on a communication channel. While showing the results, a generalization of the fault-preserving transformations introduced in [36] is also presented. Section 3.2.3 instead deals with the encoding of key material and of data on which cryptographic operations, such as encryption and hashing, are applied. Then, section 3.2.4 discusses about the application of the results, using as examples the protocols for secure web services and the SSH Transport Layer Protocol. Finally, section 3.2.5 concludes with an overview of the results and their practical impact.

3.2.1 Abstract Protocol Models and Notation

The formalism used in this context is based on CSP [34, 63], and the datatype definitions and protocol models are an extension of the ones used in [36]. Essentially, they follow the Dolev-Yao approach [27]. It is believable that the extended datatype proposed here can be enough to abstractly model the most common security protocols. Nevertheless, further extensions or modifications can be made to the datatype. The results presented here will

still be valid, provided the new datatype satisfies some properties explicitly stated in this work.

CSP was chosen in this context, instead of spi calculus, because it allowed us to directly re-use the results in [36]. In fact, the use of CSP that is done here and the extended datatype closely match what could be done with spi calculus instead. Moreover, although they happen to be expressed in the CSP formalism, the sufficient conditions described in this work are plain first order logical expressions. For these reasons, it is believable that the results presented here can be ported to spi calculus and similar formalisms.

The main extension to the CSP datatype that is introduced in [36], is an added support for non-atomic keys. This extension enables modeling protocols where the key is constructed from non-atomic data. The new datatype is defined as

$$
\begin{aligned}
\textit{Message} ::= \;& \textsc{Atom}\ \textit{Atom} \mid \\
& \textsc{Pair}\ \textit{Message}\ \textit{Message} \mid \\
& \textsc{ShKey}\ \textit{Message} \mid \\
& \textsc{PubKey}\ \textit{Message} \mid \\
& \textsc{PriKey}\ \textit{Message} \mid \\
& \textsc{ShKeyEncrypt}\ \textit{Message}\ \textsc{ShKey}\ \textit{Message} \mid \\
& \textsc{PubKeyEncrypt}\ \textit{Message}\ \textsc{PubKey}\ \textit{Message} \mid \\
& \textsc{PriKeyEncrypt}\ \textit{Message}\ \textsc{PriKey}\ \textit{Message} \mid \\
& \textsc{Hash}\ \textit{Message}.
\end{aligned}
$$

This definition has been developed using the following guidelines:

- Each key is typed. It is possible to obtain a key from generic material (that is, any generic *Message*). It is not possible to use raw material directly as a key; instead, the material must first be fed to a key construction operator.

- There is no longer need (as in [36]) for the inverse K^{-1} of a key K. Indeed, the key construction operators PubKey and PriKey fulfill this role.

- No new types are added in order to represent encoding parameters or encoded data, because the idea is to have a single datatype that can be used to model protocol data at different detail levels.

In this work, M,N,O and K range over $Message$, U and S over $2^{Message}$, A and B over honest protocol agents, P,Q and R over processes. When not explicitly quantified, all of these variables are assumed to be universally quantified over their assigned ranges.

In order to get better reading for processes, the following syntactic sugar is also provided:

Message	Representation
PAIR M M'	(M,M')
SHKEY M	$M^{\sim}$
PUBKEY M	M^{+}
PRIKEY M	M^{-}
SHKEYENCRYPT M SHKEY K	$\{M\}_{K^{\sim}}$
PUBKEYENCRYPT M PUBKEY K	$\{[M]\}_{K^{+}}$
PRIKEYENCRYPT M PRIKEY K	$[\{M\}]_{K^{-}}$
HASH M	$H(M)$

Once the datatype is defined, it is also necessary to update the intruder knowledge derivation relation $\vdash$ which models the intruder data derivation capabilities ($U \vdash M$ means that M can be derived from U). Eleven rules are defined for this new datatype:

member $M \in U \Rightarrow U \vdash M$

pairing $U \vdash M \wedge U \vdash M' \Rightarrow U \vdash (M,M')$

splitting $U \vdash (M,M') \Rightarrow U \vdash M \wedge U \vdash M'$

key derivation $U \vdash K \Rightarrow U \vdash K^{\sim} \wedge U \vdash K^{+} \wedge U \vdash K^{-}$

shared key encryption $U \vdash M \wedge U \vdash K^{\sim} \Rightarrow U \vdash \{M\}_{K^{\sim}}$

public key encryption $U \vdash M \wedge U \vdash K^{+} \Rightarrow U \vdash \{[M]\}_{K^{+}}$

private key encryption $U \vdash M \wedge U \vdash K^{-} \Rightarrow U \vdash [\{M\}]_{K^{-}}$

shared key decryption $U \vdash \{M\}_{K^{\sim}} \wedge U \vdash K^{\sim} \Rightarrow U \vdash M$

public key decryption $U \vdash \{[M]\}_{K^{+}} \wedge U \vdash K^{-} \Rightarrow U \vdash M$

private key decryption $U \vdash [\{M\}]_{K^{-}} \wedge U \vdash K^{+} \Rightarrow U \vdash M$

hashing $U \vdash M \Rightarrow U \vdash H(M)$

The following lemma about the relation $\vdash$ is needed:

Lemma 1.

$$U \vdash M \wedge U \subseteq U' \Rightarrow U' \vdash M, \tag{3.2}$$

$$U \vdash M \wedge U \cup \{M\} \vdash M' \Rightarrow U \vdash M'. \tag{3.3}$$

It has been proven in [36] for the datatype defined there, and can be proven to hold for the definition of $\vdash$ presented above, by structural induction. If the proposed datatype is extended, it is necessary to ensure that the lemma holds for the new datatype.

Honest agents and the intruder remain unchanged from [36]. For completeness, they are briefly recalled here.

A honest agent can take part in a protocol by using the following events:

$send.A.B.M$ agent A sends message M, with intended recipient B;

$receive.A.B.M$ agent B receives message M, apparently from agent A;

$claimSecret.A.B.M$ A thinks that M is a secret shared only with B; if B is not the intruder, then the intruder should not learn M;

$running.A.B.M$ A thinks it is running the protocol with B; M is a message, recording some details about the run in question.

$finished.A.B.M$ A thinks it has finished a run of the protocol with B; M is a message, recording some details about the run in question.

The *send* and *receive* events can also be treated as channels, used by agents to exchange data; the remaining events are used to formally define the desired security properties of the protocol. *Honest* is the set of all honest agents.

The intruder acts as the medium, thus being allowed to see, modify, forge or drop any message. It uses its knowledge derivation relation $\vdash$ to forge new messages from the previously learned messages. The set of messages it can derive from a knowledge S is defined as

$$deds(S) \triangleq \{M \in Message \mid S \vdash M\}.$$

Finally, the formal definition of the intruder is

$$INTRUDER(S) \triangleq \begin{array}{l} \Box_{M \in Message} send?A?B!M \rightarrow INTRUDER(S \cup \{M\}) \\ \Box \\ \Box_{M \in deds(S)} receive?A?B!M \rightarrow INTRUDER(S) \\ \Box \\ \Box_{M \in deds(S)} leak.M \rightarrow INTRUDER(S) \end{array}$$

where *send* and *receive* are the communication channels, and $leak.M$ is the event that signals that the intruder can derive M from its current knowledge. The set of all agents is defined as $Agent = Honest \cup \{INTRUDER\}$.

Like in [36], attacks are specified as trace properties. A trace specification $SPEC(tr)$ is a predicate whose free variable tr represents a trace. A process satisfies a specification if the $SPEC(tr)$ predicate is true for all the traces of the process:

$$P \textbf{ sat } SPEC \Leftrightarrow \forall\, tr \in traces(P) \cdot SPEC(tr).$$

Two predicates, namely secrecy and injective authentication (or simply authentication), define the two most common properties.

Secrecy states that if agent A believes that message M is shared only with honest agent B, then the intruder must not be able to derive M from its knowledge:

$$Secrecy(tr) \triangleq \forall\, A \in Agent; B \in Honest \cdot\ claimSecret.A.B.M \text{ in } tr \Rightarrow \neg leak.M \text{ in } tr$$

In order to define authentication, a formal definition of *AgreementSet* is first needed.

$$\begin{array}{l} M \in AgreementSet \Leftrightarrow \exists\, tr \in traces(P); A \in Agents; B \in Honest \cdot \\ \quad tr \downarrow finished.A.B.M > 0 \vee tr \downarrow running.B.A.M > 0 \end{array} \tag{3.4}$$

where $tr \downarrow e$ is the number of events e occurring in trace tr. Informally, $AgreementSet$ is the set of all the possible messages upon which the agents should agree (e.g. if the agents should agree on a key and a nonce, the $AgreementSet$ includes pairs with the first item that is a key and the second one that is a nonce).

Authentication states that, for each protocol run that A thinks it has finished with B, B must have started a protocol run with A, and both A and B must agree on some message $M \in AgreementSet$:

$$Agreement_{AgreementSet}(tr) \triangleq \forall\ A \in Agent; B \in Honest; M \in AgreementSet \cdot \\ tr \downarrow finished.A.B.M \leq tr \downarrow running.B.A.M$$

Weaker types of authentication have also been defined, for instance non injective authentication, where there is no one-to-one correspondence between the runs of actors A and B, or weak authentication, where there is no agreement on session data; they are described, for example, in [47]. It is believable that the results proven here for injective authentication also hold for weaker forms of authentication.

3.2.2 Handling the Channel Encoding/Decoding Layer

In real applications, each protocol specification that aims to be interoperable, must explicitly specify the encodings applied to the data that are exchanged by protocol actors. That is, all the actors must exchange data with the same specified *external* representation. However, internally, each actor can use any representation that is suitable for its implementation, provided it can translate data from the *internal* representation to the *external* one, and vice versa. In this work it is assumed that, as usual, such translations are implemented separately from the protocol logic. The functions that offer an interface in order to translate data representations are called here the "encoding/decoding layer".

In this section it is shown how the encoding/decoding layer of a generic protocol can be modeled externally from the protocol logic itself, so that the encoding/decoding layer interface is preserved. Then, it is formally shown that, under some constraints that can be checked on real applications, any incorrect implementation of the encoding/decoding layer cannot be more harmful than an intruder is, and can therefore be abstracted away.

Following the CSP modeling approach presented in [36], and recalled in section 3.2.1, an actor performs all of its inputs on the *receive* channel, and all of its outputs on the *send* channel. Moreover, the intruder acts as the medium, thus being allowed to see, modify, forge or drop any message. Then, for actors A and B, the abstract formal model of a protocol can be represented as in figure 3.5.

Figure 3.5: Actors A and B with $INTRUDER$ in $SYSTEM$.

The model representing all the honest agents and the intruder is called $SYSTEM$, and is formally defined as

$$SYSTEM \triangleq (|||_{A \in Honest} P_A) \parallel INTRUDER(IK_0)$$

where, for each $A \in Honest$, P_A is the CSP process that describes A's behavior, and IK_0 is the initial intruder knowledge. The parallel operator $\parallel$ without any subscripted set of events means synchronization on all the events that are in the intersection of the alphabets of the parallel processes; that is:

$$P \parallel Q \triangleq P \underset{\alpha P \cap \alpha Q}{\parallel} Q$$

Thus, in $SYSTEM$ the intruder and the honest agents synchronize on the *send* and *receive* events.

Note that in $SYSTEM$ the actors directly exchange the abstract representation of data with the intruder.

In order to model the encoding/decoding layer, a refined model $SYSTEM'$ is defined as depicted in figure 3.6 for actors A and B.

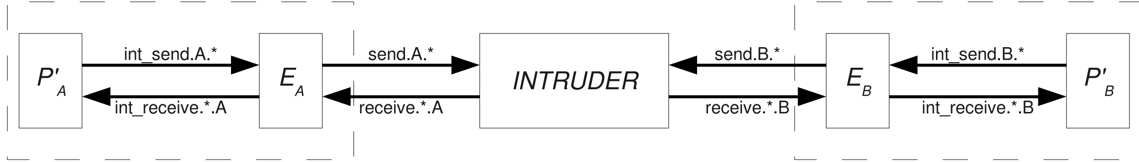

Figure 3.6: Actors A and B with $INTRUDER$ in $SYSTEM'$.

Basically, $SYSTEM'$ acts like $SYSTEM$, but it is explicitly modeled that the *external* representation of data is being sent over *send* and *receive*. More precisely, for each honest agent A, the coupled processes P'_A and E_A represent respectively the protocol logic and the encoding/decoding layer of a program. So each P'_A in $SYSTEM'$ acts *like* its corresponding P_A in $SYSTEM$, but it is explicitly modeled that it sends its *internal* representation to its coupled encoding layer E_A, which in turn sends the encoded data to the intruder, and vice versa.

This model can be described in CSP for all the honest agents as

$$SYSTEM' \triangleq (((|||_{A \in Honest} P'_A) \parallel (|||_{A \in Honest} E_A)) \setminus \{int\}) \parallel INTRUDER(IK'_0)$$

where $int = int_send, int_receive$. Here, set notation for events is sometimes abused so that a set defined as $\{a_1, \ldots, a_n\}$ is the set containing all the events that match one of the a_i forms. So, for instance, $int_send.B.A.M \in \{int\}$, and $receive.C.A.M \in \{receive?X.A\}$. The context will make clear whether proper or abused set notation is being used.

It could be argued that, potentially, this model allows each honest agent to send messages to any encoding layer, and vice versa. However, the implementations of protocol logic and its coupled encoding layer are very often part of the same application, so errors that would lead honest agents or encoding layers to communicate with the wrong process

are not realistic. For this reason, it is assumed that the P'_A model for the protocol logic is defined such that it will only exchange messages with its coupled encoding layer model E_A, and vice versa. Indeed, this assumption implies that such errors cannot happen in the model too. For the same reason, it is reasonable to hide the program internal communication channels *int_send* and *int_receive* from the intruder's view.

The initial intruder knowledge IK'_0 in $SYSTEM'$ has some relation with IK_0 in $SYSTEM$, however this relation now is irrelevant, and can be explained later.

Finally, the relation between each P_A and the corresponding P'_A and the formal definition of each E_A are given. The definitions of P_A and P'_A can be parameterized with respect to the channels used for communication. When not needed, process parameters will not be written. Thus, if s and r are the channel name parameters, $P'_A(s,r)$ can be written as P'_A, when parameters are irrelevant in the context.

For each P_A, P'_A can be built by refining P_A so as to model the information that the protocol agent must provide to the encoding/decoding layer for its proper working. More precisely, $P'_A(s,r)$ is obtained from $P_A(s,r)$ by replacing each s action $s.A.B.M$ in $P_A(s,r)$ with $s.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))$, and each r action $r.B.A.M$ with $r.B.A.(\textsc{Atom}\ \mathcal{L},(a,M))$. Here, $\textsc{Atom}\ \mathcal{L}$ is a special atom not present in the definition of P_A, whose only purpose is to tag the data exchanged on the internal channels, and a is such that $a \in Encoding \subseteq Message$ where *Encoding* is the set of messages that can be used as encoding/decoding parameters, i.e. additional information needed by the encoding/decoding layer when encoding and decoding operations are requested (e.g., an element of *Encoding* may include the name of the encoding algorithm to be applied and any related parameters, such as length of paddings etc.). From now on, a,b,c and d range over *Encoding*, and, unless explicitly quantified, they are assumed to be universally quantified over *Encoding*.

An accurate model must set, for each message M that is sent or received, its correct encoding parameters a, according to the protocol specification documents. It can also be noted that the encoding parameters may or may not be already present in P_A. For example, if the encoding parameters a are being negotiated within the protocol logic, then a will be already present in P_A. Because of this, it is possible that the message (a,M) already exists in P_A. This is why we want to distinguish the messages (a,M) already present in P_A, from those added when deriving P'_A, which is achieved by the special label message $\textsc{Atom}\ \mathcal{L}$, which has the property of never appearing in P_A. Since $\textsc{Atom}\ \mathcal{L}$ is just a syntactic marker, it is assumed that neither P_A nor P'_A ever accept $\textsc{Atom}\ \mathcal{L}$ on inputs or send it on outputs, with the only exception when $\textsc{Atom}\ \mathcal{L}$ is explicitly needed as syntactic marker.

Each process E_A models the behavior of the encoding/decoding layer. Because of this, it can perform two kinds of actions:

- receive from its coupled process P'_A internal representations of data, along with encoding parameters, and send encoded data to the *INTRUDER* process;
- receive encoded data from the *INTRUDER* process, and send to its coupled process P'_A the internal representation, obtained using the decoding parameters specified by P'_A.

Apart from these assumptions on the possible interactions of E_A, it is assumed that internally E_A can behave in any way, thus even including erroneous implementations of data transformations. The only restriction is that E_A can access only the data explicitly provided from outside. This behavior can be represented by the following CSP process (where i_s and i_r represent the internal send and receive channels):

$$\begin{aligned}
&E_A(i_s,i_r,s,r) \triangleq \\
&\mathop{\Box}_{\substack{a \in Encoding \\ M \in Message}} i_s!A?B!(\textsc{Atom}\ \mathcal{L},(a,M)) \rightarrow s!A!B!e_A(a,M) \rightarrow E_A(i_s,i_r,s,r) \\
&\Box \\
&\mathop{\Box}_{y \in Message} r?B!A!y \rightarrow \\
&\qquad \mathop{\Box}_{a \in Encoding} i_r!B!A!(\textsc{Atom}\ \mathcal{L},(a,d_A(a,y))) \rightarrow E_A(i_s,i_r,s,r)
\end{aligned} \tag{3.5}$$

where $e_A(a,M)$ and $d_A(a,y)$ represent the result of the encoding and decoding operations implemented in actor A and are messages such that

$$e_A(a,M) \in deds(\{a,M\}) \ \wedge \ d_A(a,y) \in deds(\{a,y\}) \tag{3.6}$$

Note that both $e_A(a,M)$ and $d_A(a,M)$ are normal messages belonging to the datatype, that is no new special types are added to represent them. By this definition, it is possible to state the properties of the encoding/decoding layer model E_A. The result $e_A(a,M)$ of encoding M with parameters a can be anything that can be derived from M and a, thus accounting for arbitrary complex encoding schemes. Two aspects of this definition are particularly interesting:

- $e_A(a,M)$ can contain the same or less information than M.
- All information in $e_A(a,M)$ that is not present in M must be present in a.

That is, a possibly incorrect encoding function can loose some information on M, but can only use information that comes from the internal representation and from the encoding parameters. In order to model some information that is hard-coded into the encoding function implementation, it is needed to explicitly add that information to a.

The same reasoning applies to the result $d_A(a,y)$ of decoding y with parameters a, but the case when y is not recognized as a valid encoding for parameters a must be taken into account as well. In the latter case, it is assumed that $d_A(a,y) = \textsc{Atom}\ \mathcal{E}$, where $\textsc{Atom}\ \mathcal{E}$ is a special atom that represents a decoding error code. Since this error code is part of the decoding function, it is assumed that $\textsc{Atom}\ \mathcal{E} \in deds(a)$ for any $a \in Encoding$. In the encoding/decoding layer model analyzed here, it is modeled that decoding error conditions are reported to the protocol logic, through the use of the special $\textsc{Atom}\ \mathcal{E}$ error code. An encoding/decoding layer model that gets stuck when a decoding operation fails, so that the protocol logic is never delivered the special $\textsc{Atom}\ \mathcal{E}$, is possible, and is actually a refinement of the model analyzed here. So the results obtained in this work for the encoding/decoding layer model that reports the errors to the protocol logic, are also

valid for the refined model of the encoding/decoding layer that immediately stops in case of error, and does not require the protocol logic to handle error conditions.

Another property implied by this model is that one computation of $e_A(a,M)$ and of $d_A(a,y)$ has no side effects and is memoryless. Encoding mechanisms with memory are not considered here for simplicity, but this model could be extended to include them.

It is worth noting that all the properties of the modeled encoding layer, namely that the only data accessed by the encoding/decoding functions, including hard-coded values, are their input parameters and that no side effect occurs, are information flow properties that can be verified on implementation code, by means of static sequential code analysis techniques.

Model Simplifications

The models of implementation details, introduced above, allow to refine an abstract $SYSTEM$ into a more detailed, and complex, $SYSTEM'$, which takes encoding/decoding functions into account. This section shows how, under some constraints involving the initial intruder knowledge IK_0, the detailed model $SYSTEM'$ can be simplified back to $SYSTEM$, still preserving the security faults that were present in $SYSTEM'$.

The simplification from $SYSTEM'$ to $SYSTEM$ is performed in two steps, as shown in figure 3.7: first the encoding/decoding layer is removed; then the encoding parameters are removed.

Removing the Encoding/Decoding Layer

In order to remove the encoding/decoding layer from $SYSTEM'$, a new process $SYSTEM''$ is defined as

$$SYSTEM'' \triangleq (|||_{A \in Honest} P'_A) \parallel INTRUDER(IK''_0)$$

where the initial intruder knowledge is now called IK''_0 and is assumed to be defined as

$$IK''_0 \triangleq IK'_0 \cup Encoding \cup \{\textsc{Atom}\ \mathcal{L}\} \tag{3.7}$$

Actors A and B in $SYSTEM''$ are depicted in the intermediate system in figure 3.7, that is in the system obtained after step 1, and before step 2.

A refinement relation between $SYSTEM'$ and $SYSTEM''$ is expressed by:

Theorem 3. *Let $comm = \{send, receive\}$, then*

$$P \textbf{ sat } SPEC \Leftrightarrow P \setminus comm \textbf{ sat } SPEC \tag{3.8}$$

$$\Longrightarrow$$

$$SYSTEM'' \textbf{ sat } SPEC \Rightarrow SYSTEM' \textbf{ sat } SPEC$$

That is, if the definition of attack does not involve the *send* and *receive* events, then all security properties defined on traces that are satisfied by $SYSTEM''$, are satisfied by $SYSTEM'$ too. Indeed, expression (3.8) formally states that the desired $SPEC$ property must not depend on the events in *comm*. In particular, the left-to-right implication states

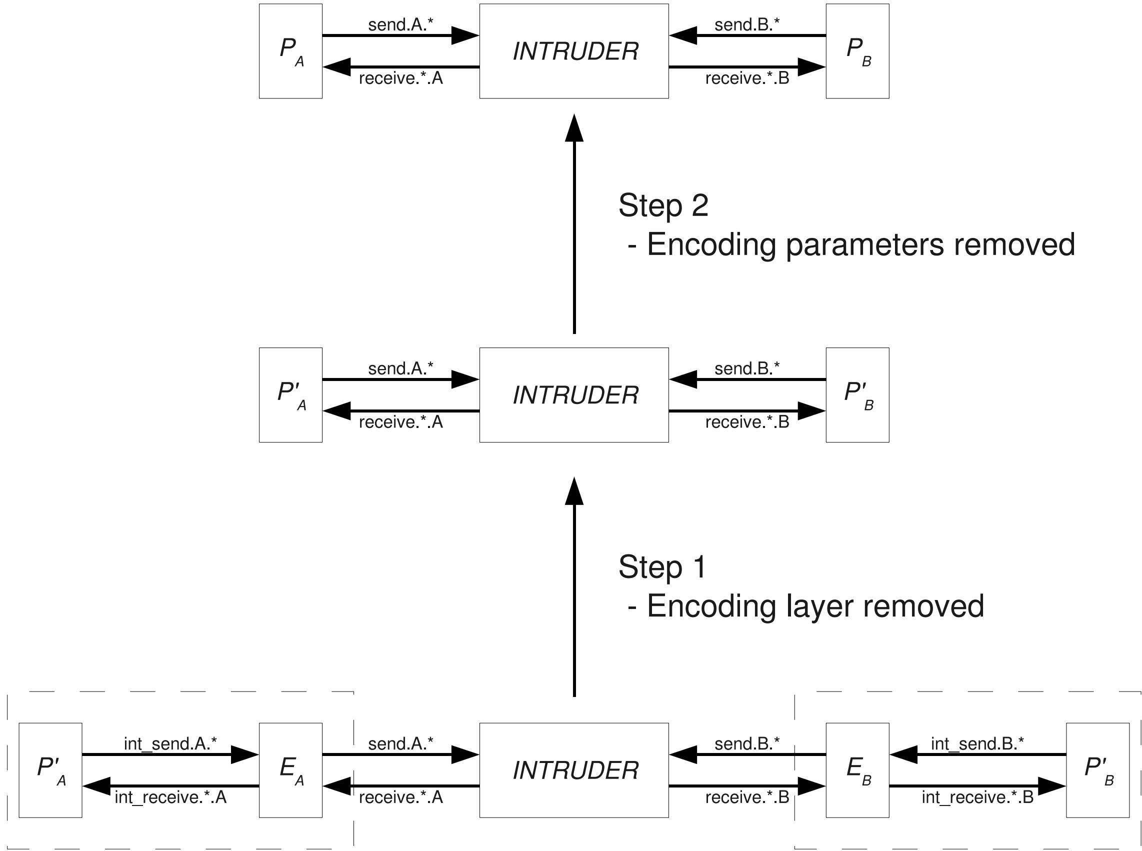

Figure 3.7: The two simplification steps leading from $SYSTEM'$ back to $SYSTEM$.

that, for any trace $tr \in traces(P)$ (that may contain *send* and *receive* events) and its corresponding trace $tr^- \in traces(P \setminus comm)$, if $SPEC(tr)$ holds, then $SPEC(tr^-)$ must hold too, because the *send* and *receive* events in tr do not affect the truth of $SPEC(tr)$. Conversely, the right-to-left implication states that if any number of *send* and *receive* events are added in any place of a trace of $P \setminus comm$, then $SPEC$ must still hold. Condition (3.8) is reasonable, because security properties are normally obtained by correct use of special events, such as $claimSecret$, $running$ or $finished$, and not directly by observing the sequence of messages exchanged on the communication channels.

Proof of theorem 3 is given in appendix B.1. Theorem 3 states that in a protocol specification where the encoding/decoding layer is modeled as previously described, only the protocol logic represented by P'_A is responsible for the security properties of the whole protocol, while any possible implementation of the encoding/decoding layer E_A of arbitrary complexity can be considered as part of the intruder, provided that the latter knows all required encoding schemes and parameters (because $Encoding \subset IK''_0$). It is also needed that the intruder knows the syntactic marker Atom $\mathcal{L}$. This is not an issue, since it is assumed that Atom $\mathcal{L}$ will only be treated as a marker by honest agents.

Note that no assumption on the invertibility of encoding functions has been made,

thus even erroneous specifications of encoding schemes are safe (thought not functional, of course). For instance, an erroneous specification that requires to collapse all nonces into a constant cannot be responsible for replay attacks, since it is protocol logic duty to check that the internal representation of the locally generated nonce is equal to the internal representation of the received unmarshaled nonce. Moreover, since no assumption on implementation correctness has been made, even erroneous implementations of the encoding scheme are safe, provided they satisfy the data flow assumptions made.

Removing the Encoding Parameters

The last step that is useful to perform when dealing with the encoding/decoding layer, is to simplify each P'_A, so that the simplified process is equal to P_A. Since each P'_A is being built from P_A (plus other information), it is possible to find a simplifying transformation that can take from P'_A back to P_A.

Simplifying transformations have been introduced in [36]. A fault-preserving simplifying transformation in its simplest form is a function

$$f : Message \rightarrow Message$$

that defines how messages in the original protocol are replaced by messages in the simplified protocol. The function f is then overloaded to take events, traces and processes, such that all messages in the events, traces or processes are replaced.

As stated in [36], let $SYSTEM^R$ be a protocol model with associated initial intruder knowledge IK_0^R, and $SYSTEM^A = f(SYSTEM^R)$ with associated initial intruder knowledge IK_0^A. If $f(\cdot)$ is a simplifying transformation that satisfies conditions

$$U \cup IK_0^R \vdash M \Rightarrow f(U) \cup IK_0^A \vdash f(M) \tag{3.9}$$

$$f(IK_0^R) \subseteq IK_0^A \tag{3.10}$$

then $SYSTEM^A$ **sat** *Secrecy* $\Rightarrow SYSTEM^R$ **sat** *Secrecy*. Note that (3.9) depends on the derivation relation $\vdash$, so this condition must be checked each time the datatype is updated.

The work in [36] also gives a sufficient condition about $f(\cdot)$ and *AgreementSet*, for $f(\cdot)$ to be a fault-preserving transformation with respect to authentication specifications. This condition requires $f(\cdot)$ to be injective for all messages belonging to *AgreementSet*, that is

$$\forall\ M \in AgreementSet; M' \in Message \cdot \\ M \neq M' \Rightarrow f(M) \neq f(M')$$

However, for some simplifying transformations it turns out to be difficult to find the constraints under which they satisfy this condition.

In this work, a new, weaker sufficient condition for fault preserving transformations with respect to authentication agreement specifications is provided. This condition requires less constraints on the simplifying transformation, so they are easier to be found.

The new sufficient condition states that

$$\forall\ M,M' \in AgreementSet \cdot\ M \neq M' \Rightarrow f(M) \neq f(M') \tag{3.11}$$

That is, $f(\cdot)$ must be *locally* injective. Note that it is possible that for some $M,M' \in Message$ with $M \neq M'$ and $M' \notin AgreementSet$, $f(M) = f(M')$, where M may or may not be in $AgreementSet$.

The following theorem 4 and corollary 1 re-state the main fault-preserving renaming transformation result about authentication by using the weakened condition.

Theorem 4. *If condition* (3.11) *is satisfied, then if a particular trace tr constitutes a failure of authentication on the original protocol, then $f(tr)$ constitutes a failure of authentication on the simplified protocol:*

$$\neg Agreement_{AgreementSet}(tr) \Rightarrow \neg Agreement_{f(AgreementSet)}(f(tr))$$

Corollary 1. *For a renaming transformation f and an AgreementSet that satisfies* (3.11)*,*

$$f(SYSTEM)\ \mathbf{sat}\ Agreement_{f(AgreementSet)} \Rightarrow SYSTEM\ \mathbf{sat}\ Agreement_{AgreementSet}$$

A thorough discussion of the enhancement made on the fault-preserving renaming transformations, including proofs for theorem 4 and corollary 1, can be found in appendix C.

A fault-preserving renaming transformation that transforms P'_A into P_A, and thus $SYSTEM''$ into $SYSTEM$ is now introduced. This transformation collapses a pair (M,M') into its first item M. This transformation is similar in purpose to the one described in [36]. However, it is worth to point out some differences:

- The definition of the renaming function is updated to the new datatype.
- Different constraints are given for $f(\cdot)$ to be fault-preserving with respect to secrecy.
- For the first time this kind of function is proven to be fault-preserving with respect to authentication on *any* message sequence (and not only with respect to message sequences made of atoms, as in [36]), and the required constraints for this result to hold are formally specified.

Let $Pairs$ be the set of pairs (M,M') that must be coalesced to M. The updated definition of $f(\cdot)$ is

$$\begin{aligned}
f(\textsc{Atom}\ A) &= \textsc{Atom}\ A,\\
f(M,M') &= \begin{cases} f(M), & \text{if } (M,M') \in Pairs \wedge \neg isPair(M'),\\ (f(M),f(M')) & \text{if } (M,M') \notin Pairs \wedge \neg isPair(M'),\end{cases}\\
f(M,(M',M'')) &= \begin{cases} f(M,M'') & \text{if } (M,M') \in Pairs,\\ (f(M),f(M',M'')) & \text{otherwise,}\end{cases}\\
f(\{M\}_K) &= \{f(M)\}_{f(K)}\\
f(\{[M]\}_K) &= \{[f(M)]\}_{f(K)}\\
f([\{M\}]_K) &= [\{f(M)\}]_{f(K)}\\
f(H(M)) &= H(f(M))\\
f(K^*) &= f(K)^*
\end{aligned}$$

where K^* ranges over $\{K^\sim,K^+,K^-\}$.

In order to preserve secrecy, $f(\cdot)$ must satisfy conditions (3.9) and (3.10). If we set

$$IK_0^A \supseteq f(IK_0^R) \cup \{f(M')|(M,M') \in Pairs\} \tag{3.12}$$

then, by induction on the relation $\vdash$, condition (3.9) is proven to hold (the proof is very similar to the one given in [36]), and condition (3.10) is clearly satisfied too. Equation (3.12) states that the intruder must already know the information that is going to be collapsed.

In order to preserve agreement, $f(\cdot)$ must satisfy condition (3.11). In order to achieve this, only one additional constraint is required:

$$\begin{aligned} &\forall\ M \in AgreementSet; subM \in subterms(M) \cdot \\ &\quad isPair(subM) \Rightarrow subM \notin Pairs \end{aligned} \tag{3.13}$$

where $subterms(M)$ is the set containing M and all its subterms. Constraint (3.13) means that no subterm of any $M \in AgreementSet$ that is a pair must be in the $Pairs$ set, that is if agreement is required on a pair, then that pair must not be collapsed.

Now it is possible to show how the coalescing pairs function $f(\cdot)$ can be safely used to transform P'_A into P_A, and thus $SYSTEM''$ into $SYSTEM$. It is worth reminding that P'_A has been obtained from P_A by replacing each sent or received message M with $(\text{ATOM } \mathcal{L},(a,M))$. Then, the following two steps are required in order to obtain back P_A from P'_A:

1. $P_A^{tmp} = f(P'_A)$, with $Pairs = \{(\text{ATOM } \mathcal{L},a)|a \in Encoding\}$
2. $P_A = f^{sym}(P_A^{tmp})$, with $Pairs = \{(\text{ATOM } \mathcal{L},M)|M \in Message\}$

where $f^{sym}(\cdot)$ is the symmetric function of $f(\cdot)$, that coalesces pairs of the form (M,M') into their second item M'.

In step 1, the syntactic marker ATOM $\mathcal{L}$ is used to find and remove all encoding parameters that have been added to represent the encoding/decoding layer. Then, step 2 removes the syntactic marker, finally obtaining P_A.

Each one of these transformations preserves secrecy and authentication if the required sufficient conditions (3.12) and (3.13) hold.

In step 1, by setting $IK_0^{tmp} = f(IK''_0) \cup f(Encoding) = f(IK'_0) \cup f(Encoding) \cup \{\text{ATOM } \mathcal{L}\}$, that is, by requiring that the intruder already knows all encoding schemes and parameters, condition (3.12) is clearly satisfied. As stated above, ATOM $\mathcal{L}$ in the intruder knowledge is not an issue.

Moreover, condition (3.13) holds because $Pairs \cap subterms(AgreementSet) = \emptyset$. Indeed, in step 1 each element in $Pairs$ has the form $(\text{ATOM } \mathcal{L},a)$; but ATOM $\mathcal{L}$ can never appear in any $running$ or $finished$ event, and thus in any subterm of the $AgreementSet$, because it is assumed that no honest agent will ever input or internally generate the ATOM $\mathcal{L}$ value, except when the syntactic marker is explicitly needed.

In step 2, condition (3.12) is clearly satisfied if we set $IK_0 = IK_0^{tmp}$; condition (3.13) holds because of the same reasoning used for step 1.

It is worth noting that, by theorem 3, it follows that $SYSTEM''$ satisfies all the security properties that can be checked on $SYSTEM'$ (provided they are not defined on the *send* or *receive* events, and the intruder knows the encoding parameters). However, when passing from $SYSTEM''$ to $SYSTEM$, the faults that are currently proven to be preserved are only those regarding secrecy and authentication specifications. That is, if $SYSTEM$ is verified instead of $SYSTEM'$, then secrecy and authentication properties are also verified for all possible implementation details represented in $SYSTEM'$. On the basis of the results achieved in this work, it is still required to explicitly verify $SYSTEM'$ (or, equivalently, $SYSTEM''$) in order to verify other security properties.

3.2.3 Handling the Encoding of Data to be Ciphered and Key Material

Cryptographic protocols must define, for interoperability, not only the encoding of messages that are sent and received on communication channels, but also the encoding of data on which cryptographic operations are applied. Since the layered approach presented in the previous section does not apply to the latter encodings, a similar but different model is now introduced in order to represent them. For simplicity, in this section encodings of messages sent and received on communication channels are disregarded, but of course the two models can be combined together, as it will be shown in the examples in section 3.2.4.

Given an abstract protocol model *SYSTEM*, where the encoding of data to be ciphered and of key material are abstracted away, a refined *SYSTEM′* that takes these details into account can be built as follows. The $INTRUDER$ process is left untouched, while each process P_A is transformed into a process P'_A, and is coupled with a decoding process DEC_A. Each pair of P'_A and DEC_A processes internally communicates by the $priv_send_A$ and $priv_receive_A$ hidden dedicated channels. *SYSTEM′* with two actors A and B is depicted in figure 3.8.

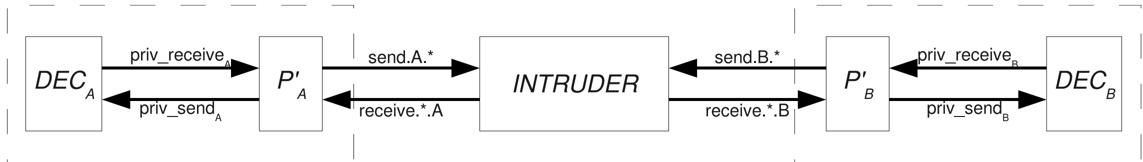

Figure 3.8: Actors A and B with $INTRUDER$ in $SYSTEM'$.

The formal definition of $SYSTEM'$ is

$$SYSTEM' \triangleq INTRUDER(IK'_0) \parallel (|||_{A \in Honest}((P'_A \parallel DEC_A) \setminus priv_send \cup priv_receive))$$

where $priv_send = \{priv_send_A | A \in Honest\}$ and $priv_receive = \{priv_receive_A | A \in Honest\}$

The P'_A process incorporates the capability of encoding data before applying cryptographic operations on them, and delegates to DEC_A the task of decoding data after decryptions. Accordingly, in order to obtain P'_A from P_A, each $send.A.B.M$ action in P_A must be changed into a $send.A.B.M'$ in P'_A, where M' is obtained from M by adding the required data and key encoding details. More precisely:

- each subterm of M taking the form $\{N\}_K$ or $\{[N]\}_K$ or $[\{N\}]_K$ will take the form $\{e_A(a,N)\}_K$, $\{[e_A(a,N)]\}_K$, $[\{e_A(a,N)\}]_K$ respectively in M';
- each subterm of M taking the form $H(N)$ will take the form $H(e_A(a,N))$ in M';
- each subterm of M taking the form of a key K^* will take the form $e_A(a,K)^*$ in M';
- each other subterm of M will remain unchanged.

Like in the previous section, $a \in \mathit{Encoding}$ represents encoding parameters, and $e_A(a,M)$ represents the result of the encoding transformation of M using parameters a. Like in the modeling of the encoding/decoding layer, a is added as to comply with the protocol specification documents.

The meaning of this refinement is that, when sending ciphered data, the encoded plaintext is ciphered, instead of its internal representation. Note that the encoding of key material is taken into account, because K^* messages are transformed into $e_A(a,K)^*$ messages. The same refinement is applied to the *claimSecret*, *running*, and *finished* actions.

A different refinement is needed instead for each *receive.B.A.M* action, which becomes *receive.B.A.M'*, followed by zero or more pairs of $\mathit{priv_send}_A.(y,a) \rightarrow \mathit{priv_receive}_A.N'$ actions, that represent interaction with the DEC_A process. More precisely:

- if M is an encryption $\{N\}_K$, and P_A can decrypt it because it knows the appropriate key, then M' will take the form $\{y\}_{K'}$, where y is a free variable and K' is obtained from K by the same procedure described for the *send* action. Moreover, a pair of $\mathit{priv_send}_A.(y,a) \rightarrow \mathit{priv_receive}_A.N'$ is added in order to represent the interaction with the DEC_A process, needed for decoding y. Here N' is obtained by recursively applying this procedure, which is being explained, to N, thus possibly appending further $\mathit{priv_send}_A$, $\mathit{priv_receive}_A$ action pairs, depending on the form of N. Then, by the $\mathit{priv_send}_A.(y,a)$ action, the encoded plaintext y and the decoding parameters a are sent to the DEC_A process, which returns the decoded representation of y. If $d_A(a,y)$ denotes the result of decoding y using parameters a, by the $\mathit{priv_receive}_A.N'$ action P'_A is forcing the match between $d_A(a,y)$ and N', that is the expected decoded representation of the plaintext. Note that N' may differ from N because N must be refined as well, so it is necessary to instantiate this procedure over N, thus obtaining N'.

  The same reasoning applies to public key encryptions $\{[N]\}_K$ and private key ones $[\{N\}]_K$.

  For example, if an abstract process P_A is ready to perform the *receive.B.A.* $\{[\{\text{ATOM } A\}_{K_1}]\}_{K_2^+}$ action, where K_1 is a shared key known by P_A (in this example K_1 is opaque, i.e. P_A has received it as an opaque message), and K_2^+ is a public key for which the corresponding private key K_2^- is known by P_A, then the refined process P'_A, coupled with its decoding process DEC_A, must be ready to perform the following sequence

of action prefixes

$$\begin{aligned} receive.B.A.\{[y]\}_{e_A(a,K_2)^+} \quad \rightarrow \quad & priv_send_A.(y,b) \rightarrow priv_receive_A.\{x\}_{K_1} \rightarrow \\ & priv_send_A.(x,c) \rightarrow priv_receive_A.\text{ATOM } A \end{aligned}$$

where $a,b,c \in Encoding$ are the encoding parameters prescribed by the protocol specification documents. Moreover, in order to be able to perform decryption operations, P'_A must know the symmetric key K_1 and the private key $e_A(a,K_2)^-$. In this example, P'_A receives an encoded message y that is deciphered by using the known refined asymmetric key $e_A(a,K_2)^-$ (note that private key is required to decrypt public-key ciphered data). Then, the encoded message y along with the decoding parameters b is sent to the coupled DEC_A process, which returns the decoded representation. This message must match the $\{x\}_{K_1}$ message, where x is again the encoded form of the plaintext, as required by the cryptographic algorithm, and K_1 is the opaque symmetric key. Note that in this model the value of K_1 will be the refined form of the symmetric key. However, since this key is opaque and known in its abstract form by P_A, it remains opaque and known in its refined form by P'_A. Finally, P'_A sends the encoded message x and the decoding parameters c to DEC_A, obtaining the decoded plaintext, which is forced to match ATOM A.

- if M is a pair (N,O), then this procedure is applied to N and O. This case is needed to handle messages where encryptions are not top-level messages, but they are contained into, possibly nested, pairs. We make the assumption that this procedure is applied depth-first to N, then to O.

- In all the other cases, M' is obtained from M by the same procedure described for the *send* action, and no pairs of $priv_send_A$, $priv_receive_A$ actions are added. By this case, it is modeled that all the data that cannot be decrypted, for example hashed data, are generated locally, taking encodings into account. Then, locally generated data are compared with received data.

  For example, if an abstract process P_A is ready to perform the $receive.B.A.\{H(\text{ATOM } A)\}_{K^\sim}$ action, then the refined process P'_A, coupled with its decoding process DEC_A, is ready to perform the sequence of action prefixes

$$receive.B.A.\{y\}_{e_A(a,K)^\sim} \rightarrow priv_send_A.(y,b) \rightarrow priv_receive_A.H(e_A(c,\text{ATOM } A)) \tag{3.14}$$

  where $a,b,c \in Encoding$ are the encoding algorithms prescribed by the protocol specification documents. In this example, P'_A receives a message that is deciphered by using the refined symmetric key $e_A(a,K)^\sim$, which is known by P'_A. The obtained plaintext y should be the encoding, required by the cryptographic algorithm, of the original message $H(e_A(c,\text{ATOM } A))$. Here y is treated by P'_A as an opaque message and it is sent along with the encoding parameters b to the coupled decoding process DEC_A. The latter returns the internal representation of y, which must equal $H(e_A(c,\text{ATOM } A))$. Again, P'_A expects to receive $H(e_A(c,\text{ATOM } A))$, and not $H(\text{ATOM } A)$, because it is taken into account that ATOM A must be encoded before being passed to the hashing algorithm.

Each DEC_A process is formally defined as follows:

$$DEC_A \triangleq \square_{\substack{y \in Message \\ a \in Encoding}} priv_send_A!(y,a) \rightarrow (priv_receive_A!d_A(a,y) \rightarrow DEC_A) \not< d_A(a,y) \neq \textsc{Atom}\ \mathcal{E} \not> STOP \quad (3.15)$$

where $P \not< b \not> Q$ means *if* b *then* P *else* Q.

Note that, like in previous section, for $e_A(a,M)$ and $d_A(a,y)$, condition (3.6) holds, and absence of side effects and of memory between calls is assumed. Moreover, an error in decoding y with parameters a is modeled by $d_A(a,y) = \textsc{Atom}\ \mathcal{E}$, where $\textsc{Atom}\ \mathcal{E}$ is the special atom representing a decoding error code, and $\textsc{Atom}\ \mathcal{E} \in deds(a)$. In this section, however, it is assumed that the decoding process stops immediately if decoding fails, which is the most realistic behavior for the kind of encoding considered in this section.

Abstracting the Refined Model

In this section it is shown that, under some conditions, if $SYSTEM$ does not have security flaws, then $SYSTEM'$ does not have any either.

Like in the previous section, although by a technically different reasoning, it is shown in a first step that under some assumptions the refined $SYSTEM'$ meets any security property that is satisfied by a more abstract, intermediate $SYSTEM^*$. Note that this relation holds for any security property that can be defined on traces, provided that it is not defined on the *send* or *receive* events that may appear in a trace. Then, in a second step, it is shown that under further assumptions the intermediate $SYSTEM^*$ can be further simplified to the original abstract $SYSTEM$, by applying a newly introduced simplifying transformation, that still preserves secrecy and authentication.

In order to obtain the intermediate $SYSTEM^*$, let us define

$$priv_A \triangleq \{priv_send_A, priv_receive_A\}$$

and the function $f(\cdot)$ that transforms P'_A into $P^*_A \triangleq f(P'_A)$. Informally, function $f(\cdot)$ removes events on the channels in $priv_A$ without changing the external behavior of the process. Formally, $f(\cdot)$ can be defined as a function on CSP processes that distributes over any CSP operator ω but the action prefix operator, on which $f(\cdot)$ acts by removing the events in $priv_A$, that is

- for any CSP operator ω, with any arity n, except action prefix:

$$f(\omega(P_1, \ldots, P_n)) = \omega(f(P_1), \ldots, f(P_n))$$

- for the action prefix operator $ev \rightarrow P$: if

$$(ev \notin \{receive?B.A\} \cup priv_A) \vee$$
$$(ev \in \{receive?B.A\} \wedge \forall\, pev \in priv_A \cdot P \neq pev \rightarrow P')$$

  then

$$f(ev \rightarrow P) = ev \rightarrow f(P)$$

else

$$f(receive.B.A.M \to priv_send_A.(y,a) \to priv_receive_A.N \to P) = \\ f\left(receive.B.A.M \left[{}^{e_A(a,N)}/_y\right] \to P\right)$$

Although function $f(\cdot)$ is defined for any CSP process, in order to keep the proofs simpler, from now on it will be assumed that P'_A is a sequential process. This assumption does not narrow the generality of our results, since as previously explained multi-threaded implementations of protocol logics can be simulated by corresponding sequential implementations.

The intermediate $SYSTEM^*$ is formally defined as

$$SYSTEM^* \triangleq |||_{A \in Honest} P^*_A \parallel INTRUDER(IK^*_0)$$

where IK^*_0 is the initial intruder knowledge in the intermediate system. It is worth noting that, in $SYSTEM^*$, the DEC_A processes have been removed. Indeed, each intermediate P^*_A process "embeds" decoding capabilities, by expecting to receive only the encoded form of data. For example, if P'_A can perform the action sequence in (3.14), then P^*_A will be able to perform the action $receive.B.A.\{e_A(b,H(e_A(c,\textsc{Atom}\ A)))\}_{e_A(a,K)^{\sim}}$.

Now that a formal relation between $SYSTEM'$ and $SYSTEM^*$ has been defined, the following theorem, needed to get to the final result, can be formulated:

Theorem 5.

$$d_A(a,y) \neq \textsc{Atom}\ \mathcal{E} \Rightarrow e_A(a,d_A(a,y)) = y \tag{3.16}$$
$$\wedge\ IK^*_0 \supseteq IK'_0 \tag{3.17}$$
$$\Longrightarrow$$
$$SYSTEM^* \setminus comm \sqsubseteq SYSTEM' \setminus comm \tag{3.18}$$

Condition (3.16) requires that, for each actor, the implementation of the encoding function $e_A(a,\cdot)$ is the inverse of the implementation of the decoding function $d_A(a,\cdot)$. Condition (3.17) simply requires that at the beginning of the protocol run, the intruder in the intermediate system knows at least the same messages known by the intruder in the refined system. Note, however, that no assumption on implementation correctness with respect to any encoding scheme specification is made, and no relationship is being assumed between encoding scheme implementations of different actors. That is, condition (3.16) can be checked in isolation on every implementation alone, without referring to any encoding scheme specification.

Also note that canonicalization schemes are neglected, because they transform data between two different items of the same equivalence class, which are normally all represented by a single term in a formal CSP model. Indeed, as stated in [43] too, canonicalization does not impact security properties, as long as all elements of the same canonicalization equivalence class have the same meaning (which actually is the aim of canonicalization). Moreover, representing canonicalization operations in abstract models would introduce

non injective functions, whose interaction with some security properties (e.g. authentication) would be rather complex, despite not so significant.

The proof of theorem 5 is given in appendix B.2.

Now, let $SPEC(tr)$ be a generic security property that can be defined on traces.

Corollary 2. *If theorem 5 holds, and $SPEC(tr)$ is such that condition* (3.8) *holds, then*

$$SYSTEM^* \textbf{ sat } SPEC \Rightarrow SYSTEM' \textbf{ sat } SPEC \tag{3.19}$$

As previously explained, condition (3.8) formally states that the desired $SPEC$ property must not depend on the events in *comm*. It is worth recalling that the given condition is reasonable, because security properties are normally obtained by correct use of special events, such as *claimSecret*, *running* or *finished*, and not directly by observing the sequence of messages exchanged on the communication channels.

Proof. Trace refinement (3.18) implies

$$SYSTEM^* \setminus comm \textbf{ sat } SPEC \Rightarrow SYSTEM' \setminus comm \textbf{ sat } SPEC$$

Then, by using the $\Leftarrow$ side of (3.8), it follows that (3.19) holds. □

Summing up the results of theorem 5 and corollary 2, if the intruder knowledge in the abstract system is no less than the intruder knowledge in the refined system, and the implementation of the encoding function of each actor is the inverse of the implementation of the decoding function of the same actor, then the more abstract $SYSTEM^*$ can be verified instead of $SYSTEM'$, for any security property that does not depend on the *send* and *receive* events.

By this result, when a model extraction approach like the one presented in [20] is used, the verification of any security property can be safely divided into two distinct verifications. On one hand, the verification of the property on an abstract protocol model where all the decoding functions that are modeled in this work by the DEC_A processes are left out. On the other hand, the verification that the sequential code of each encoding procedure implements the inverse of the corresponding decoding function implementation.

Let us consider now the further simplification from $SYSTEM^*$ to $SYSTEM$. In order to define this transformation, one more constraint must hold. Let $e(a,M)$ and $d(a,y)$ denote the definitions of the encoding and decoding functions (in contrast with $e_A(a,M)$ and $d_A(a,y)$ that define their implementations in agent A). The additional condition is

$$e_A(a,O) = e(a,O) \tag{3.20}$$

which means that the encoding implementation in each actor is correct with respect to the specification of the encoding scheme. In practice, if (3.20) holds, then all actor's implementations of encoding functions are equivalent, so that implementing actors can be ignored. So, the $e(a,O)$ symbolic form of encoding will be used from now on, regardless of the implementing actor.

By (3.20), it is possible to introduce a fault preserving simplifying transformation $f_d(\cdot)$, defined as the identity function except for the following cases:

$$\begin{array}{rcl} f_d(M,M') & = & (f_d(M),f_d(M')) \\ f_d(\{e(a,M)\}_K) & = & \{f_d(M)\}_{f_d(K)} \\ f_d(\{[e(a,M)]\}_K) & = & \{[f_d(M)]\}_{f_d(K)} \\ f_d([\{e(a,M)\}]_K) & = & [\{f_d(M)\}]_{f_d(K)} \\ f_d(H(e(a,M))) & = & H(f_d(M)) \\ f_d(e(a,K)^*) & = & (f_d(K))^* \end{array}$$

With this definition, $P_A = f_d(P_A^*)$ for any agent A. Preservation of security properties in the refinement from $SYSTEM$ to $SYSTEM^*$ can now be expressed by the following theorems.

Theorem 6.

$$\begin{array}{c} IK_0 \supseteq f_d(IK_0^*) \cup f_d(Encoding) \\ \Longrightarrow \\ SYSTEM \ \textbf{\textit{sat}} \ Secrecy \Rightarrow SYSTEM^* \ \textbf{\textit{sat}} \ Secrecy \end{array} \quad (3.21)$$

Proof. In order to prove that secrecy in the abstract system implies secrecy in the refined system, it is enough to show that $f_d(\cdot)$ is actually a fault preserving simplifying transformation that preserves secrecy, which amounts to check that conditions (3.9) and (3.10) are satisfied.

Satisfaction of condition (3.9) can be proven by induction over the knowledge derivation relation $\vdash$; while satisfaction of condition (3.10) can be proven by hypothesis (3.21), which is reasonable because it simply requires that the intruder in the abstract system knows at least the simplified form of messages that are known by the intruder in the refined system, along with the encoding parameters. □

In order to prove that authentication is preserved when refining $SYSTEM$ into $SYSTEM^*$, one further condition that must hold for messages in the $AgreementSet$ need to be stated.

Let us introduce now the *symbolic expression* of a message, that is a term that represents the message but leaving all data encoding operations in their symbolic form $e(a,O)$ (in contrast to resolve them to the resulting term obtained by encoding message O with parameters a).

Using the symbolic expression concept, the $AgreementSet$ can be partitioned into equivalence classes. Two messages M and M' belong to the same equivalence class if their symbolic expressions are equal, modulo a renaming of the first argument of each encoding operation occurring in them (namely the a argument of $e(a,O)$). The $M \sim M'$ notation means that M and M' belong to the same equivalence class. For example, if $(H(e(a,O)),N)$ is the symbolic expression of M, and $(H(e(b,O)),N)$ is the symbolic expression of M', then $M \sim M'$ is true; in contrast, if $(H(e(b,O)),N')$ is the symbolic expression of M' and $N \neq N'$, then $M \nsim M'$, because their symbolic expressions also differ by the $N \neq N'$ terms. In other words, each equivalence class contains all the

messages that can be obtained by applying encodings with different encoding parameters to the same unencoded message.

Theorem 7. *If* (3.21)*,* (3.20)*, and*

$$\begin{array}{c}\forall\ M,M' \in \mathit{AgreementSet} \cdot \\ M \sim M' \Rightarrow M = M'\end{array} \tag{3.22}$$

hold, then

$$\begin{array}{c}\mathit{SYSTEM}\ \textbf{\textit{sat}}\ \mathit{Agreement}_{f_d(\mathit{AgreementSet})} \Rightarrow \\ \mathit{SYSTEM}^*\ \textbf{\textit{sat}}\ \mathit{Agreement}_{\mathit{AgreementSet}}\end{array}$$

Condition (3.22) states that each equivalence class must have only one element. In other words, it must never happen that *AgreementSet* contains two messages that share the same symbolic expression, except for some encoding parameters. Indeed, as it is done in practice by many real security protocols, this condition can be enforced in practice by explicitly including, within each message upon which agreement is required, the parameters that must be used to encode each part of the message itself. Agreement on the encoding parameters used to encode data on which agreement is required is an authentication property that holds if the negotiation algorithm used in the protocol to establish such parameters is logically correct in an unsafe environment. This can be verified as part of the formal protocol verification task on the abstract protocol, as shown in section 3.2.4.

Proof. In order to prove that $f_d(\cdot)$ preserves agreement, since conditions (3.9) and (3.10) have already been proven for corollary 6, it is enough to prove (3.11), that is $f_d(\cdot)$ is locally injective on *AgreementSet*. In other words, if by hypotheses (3.21), (3.20) and (3.22), $f_d(\cdot)$ satisfies condition (3.11), then, by theorem 4, this theorem is proven.

Now, function $f_d(\cdot)$ is shown to be locally injective on *AgreementSet*.

Let $M,M' \in \mathit{AgreementSet}$ with $M \neq M'$. If $M \nsim M'$, then M and M' are terms with different structures, or, if they have the same structure, there exist two subterms N and N' with $N \neq N'$, in the same position in M and M' respectively, that cause them to differ. In this case, $f_d(M) \neq f_d(M')$ because it can be easily shown, by structural induction over messages, that $f_d(\cdot)$ preserves message structure and does not alter subterms, except for removing symbolic encoding operations, and their first parameter, which is not what is making M and M' different in this case.

If $M \sim M'$, then, by (3.22), if follows that $M = M'$, which contradicts the hypothesis, so this case cannot happen. □

Note that if extensions of the proposed datatype are used, structural inductions used in the proofs of theorems 6 and 7 must be checked to hold for the new datatype.

3.2.4 Applications and Examples

This section shows some practical applications of the results illustrated here. First, a class of concrete encoding schemes of the family discussed in section 3.2.3, that includes XML encodings, is considered. Regarding this class, all the conditions required by theorems 5, 6

and 7 for enabling the safe application of abstractions on an implementation of XML encodings like the one described in [18] are analyzed.

Then, it is shown how incidentally a particular instance of the modeling framework proposed in section 3.2.3 can be used for a somewhat different purpose, i.e. to give sufficient conditions under which cryptographic algorithms and parameters can safely be abstracted away in Dolev-Yao models.

Finally, the modeling of an SSH Transport Layer Protocol client is presented. Both an abstract model and a refined one are provided. The refined model includes altogether marshaling functions, as explained in section 3.2.2, encodings of data to be ciphered, as illustrated in section 3.2.3, and cryptographic algorithms and parameters, as explained in the example in section 3.2.4. Upon the results showed here, the abstractions that can be made on this particular model are shown, along with the conditions that must be checked or assumed on the sequential code that implements the various encoding and decoding functions, in order to safely apply abstractions.

A Class of Data Encodings Including XML Encodings

The encoding schemes considered in this example simply add a header to each part of the message being encoded, and do not alter the message content itself. Formally:

$$\begin{array}{rcll} e(a,M) & = & (head(a,M),M) & ,\text{if } \neg isPair(M) \\ e(a,(M,M')) & = & (e(a,M),e(a,M')) & \end{array} \tag{3.23}$$

where $head(a,M) \in deds(\{a,M\})$ is an header that may depend on parameters a and on message M. The peculiarity of this kind of encoding function is that it distributes headers across pairs. It is possible to define the symmetric encoding function, that adds a trailer, or padding, to data; it is furthermore possible to combine the two.

This class of encoding schemes is general enough to include, for example, XML encodings. Then, it can be used to model the data encodings used in the protocols of the WS-Security [51] standard.

In [18], an implementation of the XML encoding scheme where XML fragments are internally represented as F# (an ML dialect) records is described. For example, the XML security header specified in the WS-Security standard is internally stored as:

```
type security = {
    timestamp: ts;
    utoks: utok list;
    xtoks: xtok list;
    ekeys: encrkey list;
    dsigs: dsig list }
```

Then, a set of `gen*` and `parse*` functions, implemented in F#, is used to translate internal records to and from XML, for example when an XML fragment must be encrypted.

In the CSP modeling framework for cryptographic protocols used here, an F# record (i.e. the internal data representation) can be modeled by means of nested pairs. The F#

functions translating to and from XML, are actually encoding functions that add some XML header and trailer to each element of the record, that is to the nested pairs. Thus, the `gen*` and `parse*` functions behave like the encoding scheme defined in (3.23).

In order to check that the implementation of these functions satisfies theorem 5, it is enough to check that, on one hand, the `gen*` functions only add a tagged fragment before and after any record element, leaving the record element unchanged (with the exception of canonicalization operations, which are abstracted in the model). Note that the added tagged fragment may be anything that can be correctly recognized in the decoding phase; for theorem 5 to hold, correctness of implementation w.r.t. the XML specification is not required. On the other hand, it is enough to check that the `parse*` functions match some expected tagged header and trailer, that must comply with the expected record type and value, and that they store the data between header and trailer into the proper record field without further modification (with the exception of canonicalization operations).

For applying theorems 6 and 7, the correctness of the encoding functions implementation w.r.t. their specification is additionally needed. This amounts to the extra check that the tagged data added by the encoding `gen*` functions, and recognized by the decoding `parse*` functions, actually comply with the XML and WS-Security standards.

In [18], a model extraction approach for security protocols is proposed. That is, a formal Dolev-Yao model is extracted from the implementation code of the protocol, and then security properties are checked on the extracted model. In that work, each time a model extraction is performed, a model is extracted from the `gen*` and `parse*` function implementations too, and a global protocol model including the model of these encoding functions is formally analyzed. Moreover, messages are modeled in their encoded, complex form.

By using the results presented in this work, correctness of the sequential `gen*` and `parse*` functions requires a one-off verification, then it is possible to exclude these functions from the model extraction process, and to use the simpler abstract representation of messages, thus reducing model complexity and required verification resources, while keeping the same security assurance. This is possible, because it has already been proven, by theorems 5, 6 and 7, that secrecy and authentication faults cannot arise from the encoding functions, or because of using encoded forms of messages, provided the stated conditions hold.

A similar result can be obtained when using a code generation approach. If formally verified implementations of `gen*` and `parse*` functions are available, they can be used to correctly refine abstract models into concrete implementations while keeping the same security w.r.t. a Dolev-Yao intruder.

Conditions for Abstracting Cryptographic Algorithms and Parameters

The goal of this example is to show how cryptographic algorithms and parameters can be modeled and safely abstracted away in protocol models, by reusing the results presented in section 3.2.3. For this reason, protocol role implementations are less relevant in this context.

The abstract datatype defined in section 3.2.1, intentionally represents the result of

encryption as a function of a plaintext message and a key. The ciphertext has the property of being restored to plaintext only if the appropriate key is known. In the same way, the hash function is simply a non invertible representation of a message.

However, different real cryptographic algorithms obtain these properties in different, incompatible ways. For example, in a real application, if a message M is ciphered with the key K^+, by using the RSA algorithm, obtaining $\{[M]\}_{K^+}$, then the decryption of $\{[M]\}_{K^+}$ using the key K^- will not get M, if not exactly using the RSA algorithm. An interesting example is given by protocols that require agreement on the value of an hashed message, which, for example, is then used as the material to build a shared session key. If both actors obtain the same message M, and compute $H(M)^\sim$, it is a prerequisite to key agreement that they have previously agreed on the same hashing algorithm and key construction algorithm, otherwise they could obtain the same logical value (a shared key obtained from a non invertible representation of M), but different concrete values. Then, key agreement may fail in the concrete world, even if the actors agree on the abstract $H(M)^\sim$.

In order to faithfully describe this issue, cryptographic algorithms and their parameters are sometimes introduced in abstract descriptions. For example, in [36, 20], encryption functions are distinguished according to the algorithm and parameters they use.

In this work, in order to handle cryptographic algorithms and parameters, one could extend the datatype and the associated derivation relation $\vdash$ shown in section 3.2.1. The obtained datatype would be similar to the one presented in [36]. However, this approach would not give us the opportunity to easily discuss about the conditions under which abstracting these details is safe.

Fortunately, it turns out that the modeling approach in section 3.2.3 can be used in a particular way, such that the models of cryptographic algorithms and parameters can be represented as encodings in the current datatype, rather than being explicitly added to it. This approach has the advantage that the sufficient abstraction conditions already shown in section 3.2.3 can be directly reused in order to abstract cryptographic algorithms and parameters too.

So, referring to the modeling approach presented in section 3.2.3, let us define the following encoding.

$$\begin{aligned} e((\textsc{Atom}\ \mathcal{L},a),M) &= (\textsc{Atom}\ \mathcal{L},(a,M)) \\ d((\textsc{Atom}\ \mathcal{L},a),(\textsc{Atom}\ \mathcal{L},(a,M))) &= M \\ d((\textsc{Atom}\ \mathcal{L},a),N) &= \textsc{Atom}\ \mathcal{E}, \text{ if } N \text{ does not take the form } (\textsc{Atom}\ \mathcal{L},(a,M)) \end{aligned} \tag{3.24}$$

These definitions clearly represent functions that satisfy equation (3.16). As previously explained, $\textsc{Atom}\ \mathcal{L}$ is a syntactic marker used to tag added data, and $a \in \mathit{Encoding}$ are the cryptographic algorithms and parameters chosen according to the protocol specification documents.

The refined model that is obtained in this way corresponds to the one used in [36, 20] where different encryption, decryption and hashing functions are used for each different choice of algorithms and parameters.

For example, the refined encryption

$$\{(\text{ATOM } \mathcal{L},(DES_encrypt,M))\}_{(\text{ATOM } \mathcal{L},(DES_key,K))^{\sim}} \tag{3.25}$$

expressed in the modeling framework being presented here, corresponds, using an approach like the one presented in [36, 20], to a term like $DESEncrypt(DESKeyBuild(K),M)$, i.e. the encryption of M performed using the $DES_encrypt$ cryptographic algorithms and parameters, and a shared key constructed from the material K and the DES_key algorithms and parameters.

It is worth noting that this refined model can fully handle the case where both encryption and key construction have their own, possibly different parameters. Also, the refined hash

$$H(\text{ATOM } \mathcal{L},(SHA1,M))$$

would correspond, using an approach like the one presented in [36, 20], to a term like $SHA1Hash(M)$, which means that the hashing of M is performed using the SHA-1 algorithm.

Another property that is shared between the refined model used here and the datatype presented in [36], is that in both approaches it is impossible to achieve “security through obscurity”. For example, in the refined encryption (3.25), it is enough for the intruder to know the key $(\text{ATOM } \mathcal{L},(DES_key,K))^{\sim}$, or key material K and parameters DES_key, in order to get both the plaintext M and the cryptographic parameters $DES_encrypt$. Similarly, in [36, 20], the intruder can apply any encryption and decryption function, thus being able to deduce which algorithm and parameters have been used for generating an encrypted term. This is a strong assumption, however it is realistic. Many cryptographic algorithms put enough redundancy in the key (and even in the ciphertext), that it is possible to guess what algorithm has been used. Moreover algorithms are a few number, so even brute force attack on all known algorithms is feasible. One can argue that some symmetric encryption algorithms use an initialization vector, which is difficult to guess. This is true, however the assumption that the intruder can guess the initialization vector represents the worst case analysis, which is acceptable, and sometimes even desirable, when dealing with security protocols implementations. If the initialization vector shall be regarded as non-guessable, then a nonce shall be explicitly added to the refined model.

Now, sufficient conditions, so that the model of cryptographic algorithms and parameters that has been encoded into the current datatype can be abstracted away, can be stated, by reusing the results given in section 3.2.3.

First of all, the DEC_A decoding process can be safely removed in this case, because condition (3.16) is satisfied by the definition (3.24) of this specific encoding scheme and condition (3.17) naturally holds.

Moreover, since the encoding defined in (3.24) is injective too, the encoding that represents cryptographic algorithms and parameters can be completely abstracted away in secrecy verifications, provided that condition (3.21) holds, i.e. provided that the intruder is assumed to know cryptographic algorithms and parameters, which is a reasonable assumption. Then, the first result is that, by appropriately setting the intruder knowledge

when verifying an abstract model, secrecy is implied in the refined model for any possible security protocol, regardless of its logic.

For what concerns the verification of authentication, the model of cryptographic algorithms and parameters can be completely abstracted away if conditions (3.20) and (3.22) hold too. Since in this example we are modeling cryptographic algorithms and parameters for all actors by the same encoding and decoding functions, it is safe to assume that (3.20) holds. Nevertheless, unlike (3.21), the truth of (3.22), and thus agreement preservation, depends on the particular protocol logic that is being verified: as stated in the previous section, condition (3.22) can be enforced by specifying (and verifying) agreement on all the cryptographic algorithms and parameters used to build data on which agreement is required. Then, the second result is that cryptographic parameters and algorithms can be safely abstracted away in authentication verification provided that all the cryptographic algorithms and parameters used to build data on which agreement is required are explicitly included in the agreement statements to be verified.

Modeling an SSH Transport Layer Protocol Client

In this example, another syntactic sugar is added: lists of n messages are reduced to nested pairs. So, for example, (M,M',M'') is equal to $(M,(M',M''))$.

The SSH Transport Layer Protocol [77] (SSH-TLP) is part of the SSH three protocols suite [76]; in particular it is the first protocol that is used in order to establish an SSH connection between client and server. SSH-TLP gives server authentication to the client, and establishes a set of session shared secrets.

A simple CSP model of an SSH-TLP client is given here, in order to show how the encoding/decoding layer modeling applies. A full spi calculus SSH-TLP model that can be formally verified and from which an implementation can be derived is given in chapter 4.

Figure 3.9 shows a possible CSP fully abstract model of an SSH-TLP client. Note that

$$
\begin{array}{l}
SSHClient(IDC,me,you,CAlgs) = \\
\quad send!me!you!IDC \rightarrow \\
\quad receive!you!me?IDS \rightarrow \\
\quad \sqcap_{cookieC \in Cookies}\, send!me!you!(cookieC,CAlgs) \rightarrow \\
\quad receive!you!me?(cookieS,SAlgs) \rightarrow \\
\quad g := Negotiate(CAlgs,SAlgs,\text{`}g\text{'}) \in CryptoParameter; \\
\quad p := Negotiate(CAlgs,SAlgs,\text{`}p\text{'}) \in CryptoParameter; \\
\quad \sqcap_{x \in DHSecrets}\, send!me!you!EXP(g,x,p) \rightarrow \\
\quad receive!you!me?(PubKeyS,DHPublicS,[\{H(finalHash)\}]_{PriKeyS}) \rightarrow \\
\quad GO(EXP(DHPublicS,x,p),finalHash,PubKeyS,IDS)
\end{array}
$$

Figure 3.9: A possible CSP abstract model of an SSH-TLP client.

this model is part of the abstract $SYSTEM$, where one or more instances of $SSHClient$ can partake together with one or more instances of the server model for SSH-TLP, and the intruder.

The $SSHClient$ process starts a protocol session with the server by sending it the client identification string denoted IDC. The server responds with IDS, the server identification string. Then the client generates and sends a nonce *cookieC*, followed by the client lists of supported algorithms $CAlgs$. The server responds sending a nonce *cookieS*, followed by the server lists of supported algorithms $SAlgs$. The client then computes the value of the Diffie Hellman (DH) parameters g and p by computing the $Negotiate(CAlgs,SAlgs,Param)$ function, which returns the requested negotiated algorithm parameter named $Param$, obtained from the supported client and server algorithms $CAlgs$ and $SAlgs$. $CryptoParameter \subseteq Message$ is the set of messages that can be used as cryptographic algorithms or parameters. Once g and p have been obtained, the client generates a random private key x and sends $EXP(g,x,p)$ (its DH public key, as explained below), which is a message representing the result of the modular exponentiation $e = g^x \bmod p$. This message is added to the datatype and is defined as

$$\textsc{Exp}\ Message,Message,Message$$

along with the syntactic sugar $\textsc{Exp}\ g,x,p = EXP(g,x,p)$. From the point of view of the intruder knowledge derivation relation $\vdash$, the $EXP(\cdot)$ message is a non invertible function like an hash. Accordingly, only a single **exponentiation** rule is needed, defined as

exponentiation $B \vdash g \wedge B \vdash x \wedge B \vdash p \Rightarrow B \vdash EXP(g,x,p)$

This rule is similar to the **hashing** one. For this reason, all previously proven results still hold.

The following additional property of the $EXP(\cdot)$ function must be modeled

$$EXP(EXP(g,y,p),x,p) = EXP(EXP(g,x,p),y,p) \tag{3.26}$$

so that two messages satisfying equation (3.26) are considered equal by all $Agents$, both protocol actors and the intruder.

When the EXP function is used for the DH key exchange algorithm, x and y can be considered as DH private keys, $EXP(g,x,p)$ and $EXP(g,y,p)$ as DH public keys, and the expression in (3.26) as the DH shared key that can be obtained by each actor. Finally, g and p are the DH group parameters. In the next step of the protocol, the client receives a message containing the opaque server public key $PubKeyS$, the opaque server DH public key $DHPublicS$ and the server signed final hash $[\{H(finalHash)\}]_{PriKeyS}$. Note that $finalHash$ is hashed, because the signature algorithm prescribes to hash, and then cipher, the value that must be signed. The server computes its DH public key as $DHPublicS = EXP(g,y,p)$, where y is the server's DH private key. However the client is modeled to receive an opaque $DHPublicS$ message, because the server DH public key is an opaque value from the client's point of view. Analogous reasoning holds for public and private server keys $PubKeyS$ and $PriKeyS$. The server $finalHash$ is the value upon which agreement is required, and contains all the relevant data of a protocol session, that is

$$\begin{aligned} finalHash = H(&IDC,IDS,(cookieC,CAlgs),(cookieS,SAlgs),PubKeyS,\\ &EXP(g,x,p),DHPublicS,EXP(DHPublicS,x,p)) \end{aligned}$$

The $EXP(DHPublicS,x,p)$, that is used inside the final hash, is the DH shared key as computed by the client. This is the session secret shared between the client and the server. Finally, the $GO(\cdot)$ process is defined as

$$\begin{aligned}&GO(DHKey,finalHash,PubKeyS,IDS) =\\&\quad(claimSecret.me.you.DHKey \rightarrow finished.me.you.finalHash)\\&\not\leqslant PubKeyS == TrustedKeyOf(IDS) \not\geqslant STOP\end{aligned}$$

That is, if the received server public key $PubKeyS$ corresponds to the locally stored trusted key for the server identified by IDS, which is retrieved by the function $TrustedKeyOf(IDS)$, then the protocol run ends well, and all security properties can be claimed, namely the secrecy of the DH shared key $DHKey$, and the agreement on the server signed $finalHash$. Note that agreement on $finalHash$ implies agreement on all the data items on which the final hash is computed. However, since the final hash is used later on with other data in order to establish a set of session keys, it is required that actors agree explicitly on $finalHash$, and not only on its content.

Now that the abstract model of the client has been introduced, it is possible to show how this model can be refined, according to the modeling approach presented here. Three kinds of details that have been studied in isolation, namely the encoding/decoding layer, as modeled in section 3.2.2, the encoding of data to be ciphered or hashed and key material, as modeled in section 3.2.3, and the cryptographic algorithms and parameters, as modeled in section 3.2.4, are applied altogether in this model.

It is worth pointing out that the three different kinds of refinements must be applied on the same system in the correct order, so as to avoid improper interactions: first, data to be ciphered or hashed and key material should be refined; then cryptographic algorithms and parameters should be added; finally the encoding/decoding layer should be introduced.

For better clarity, we will no longer refer to a single set of parameters $Encoding \subseteq Message$. Instead, a different set is defined for each kind of parameter used for refinement. Specifically, *MarshEncoding* will denote messages used as the encoding/decoding layer parameters; *CryptoEncoding* will denote messages used as encoding parameters for data to be ciphered or hashed and for key material; *CryptoParameter* will denote messages used as cryptographic algorithms and related parameters.

The refined model for the SSH-TLP client includes, in addition to a refined model of the client role, an encoding/decoding layer model E_A, and a decoding process model DEC_A. Most of the complexity of the protocol indeed stands in E_A and DEC_A.

The refined client role model can be written as shown in figure 3.10. In this refined model, *string* denotes the SSH string encoding, *bytes* denotes a raw encoding (a sequence of bytes), *mpint* denotes the SSH encoding of a multiple precision integer and *namelists* denotes the SSH encoding of a list of lists of strings. The *bin_pack* parameter specifies that the SSH binary packet encoding must be used, with a payload made up of as many fields as the number of subsequent parameters, each of which in turn specifies the encoding to be applied to each field. So, for example, KEX specifies the encoding of an SSH key exchange packet, which is an SSH binary packet with a payload that includes a sequence of bytes followed by a list of strings. The $FinalHashAlg$, $SignHashAlg$, $SignKeyType$,

$$
\begin{array}{l}
SSHClientRef(IDC,me,you,CAlgs) = \\
\quad send!me!you!(\text{ATOM } \mathcal{L},(string,IDC)) \rightarrow \\
\quad receive!you!me?(\text{ATOM } \mathcal{L},(string,IDS)) \rightarrow \\
\quad \sqcap_{cookieC \in Cookies}\, send!me!you!(\text{ATOM } \mathcal{L},(KEX,(cookieC,CAlgs))) \rightarrow \\
\quad receive!you!me?(\text{ATOM } \mathcal{L},(KEX,(cookieS,SAlgs))) \rightarrow \\
\quad g := Negotiate(CAlgs,SAlgs,\text{`}g\text{'}) \in CryptoParameter; \\
\quad p := Negotiate(CAlgs,SAlgs,\text{`}p\text{'}) \in CryptoParameter; \\
\quad FinalHashAlg := Negotiate(CAlgs,SAlgs,\text{`}FinalHashAlg\text{'}) \in CryptoParameter; \\
\quad SignHashAlg := Negotiate(CAlgs,SAlgs,\text{`}SignHashAlg\text{'}) \in CryptoParameter; \\
\quad SignKeyType := Negotiate(CAlgs,SAlgs,\text{`}SignKeyType\text{'}) \in CryptoParameter; \\
\quad SignMode := Negotiate(CAlgs,SAlgs,\text{`}SignMode\text{'}) \in CryptoParameter; \\
\quad SignPadding := Negotiate(CAlgs,SAlgs,\text{`}SignPadding\text{'}) \in CryptoParameter; \\
\quad \sqcap_{x \in DHSecrets}\, send!me!you!(\text{ATOM } \mathcal{L},((bin_pack,mpint),EXP(g,x,p))) \rightarrow \\
\quad receive!you!me?(\text{ATOM } \mathcal{L},((bin_pack,string,mpint,string), \\
\qquad (PubKeyS,DHPublicS,[\{(\text{ATOM } \mathcal{L},((SignMode,SignPadding), \\
\qquad y))\}]_{PriKeyS}))) \rightarrow \\
\quad int_send_{me}!(y,(SignMode,SignPadding)) \rightarrow \\
\quad int_receive_{me}!H(SignHashAlg,e_{me}(SignHashAlg,finalHash_enc)) \rightarrow \\
\quad GO(EXP(DHPublicS,x,p),finalHash_enc,PubKeyS,IDS)
\end{array}
$$

where:

$$
\begin{aligned}
KEX &= (bin_pack,bytes,namelists) \\
finalHash_enc &= H(FinalHashAlg, \\
&\qquad e_{me}(FinalHashAlg,(IDC,IDS,(cookieC,CAlgs),(cookieS,SAlgs), \\
&\qquad PubKeyS,EXP(g,x,p),DHPublicS, \\
&\qquad EXP(DHPublicS,x,p))))
\end{aligned}
$$

Figure 3.10: A refined version of the CSP model of an SSH-TLP client.

$SignMode$, and $SignPadding$ variables represent the negotiated cryptographic algorithms and parameters (in addition to p and g, which were already present in the abstract model).

Note that the $e_{me}(\cdot)$ function in the expression for $sshash_enc$ represents the implementation of encoding transformations. Its detailed description is omitted here for simplicity.

Let us show now how, by the results presented here, this refined model can be simplified to perform formal verifications of security properties, and what checks are needed on the sequential code that implements encoding/decoding functions in order to safely apply each simplification.

In order to remove the models of the encoding/decoding layer E_A from the refined system model $SYSTEM'$, it is needed that the desired security properties do not depend

on the *send* and *receive* events and that the intruder knowledge used for verification in the abstract system includes encoding parameters (already shown to be reasonable constraints). Moreover, the implementations of the marshaling functions must be checked to be memoryless, and to access no external data but their input parameters (which amounts to check an information flow property on sequential code). Since the properties to be verified for this protocol are secrecy and authentication, even the ATOM $\mathcal{L}$ term and the messages in *MarshEncoding* that have been added by the encoding/decoding layer refinement procedure can be abstracted away without need of further checks.

The models of cryptographic algorithms and parameters only affect data terms including cryptographic operations. For verifying the secrecy property, they can be abstracted away with no additional check. When verifying authentication, they can be abstracted away if condition (3.22) holds, which can be ensured by explicitly adding the negotiated cryptographic algorithms and parameters to the internal actions used to specify agreement. For example, the *finished* action should become

$$finished.me.you.(finalHash_enc,FinalHashAlg)$$

Note that this condition is needed even though the negotiated algorithm is already included in *CAlgs* and *SAlgs*, because it is necessary to ensure that the specific negotiated algorithm is agreed, rather than the sets of algorithms from which it is selected.

Finally, in order to abstract the decoding processes DEC_A away, condition (3.16) must be checked on the sequential code of the implementation of the encoding/decoding functions $e_{me}(\cdot)$ and $d_{me}(\cdot)$, along with the same data flow properties already specified for the marshaling and unmarshaling functions. When verifying secrecy, even data encodings can be abstracted, by removing the $e_{me}(\cdot)$ functions from the client role model, provided the implementation of the sequential encoding/decoding functions is shown to be correct with respect to their specification; this check can be done in isolation. The same simplification can be applied when verifying authentication, but in this case condition (3.22) must be guaranteed to hold too. This condition can be ensured as already explained for the abstraction of cryptographic parameters.

3.2.5 Discussion

The work presented here is a step towards the verification of refined security protocol models that take encoding and decoding data transformations into account, thus allowing formal verification to get closer to protocol code written in a programming language.

The main contribution is the formulation of a set of sufficient conditions under which the models of data encoding and decoding functions can be safely simplified or even completely abstracted away under the Dolev-Yao assumption, and its formal justification.

It has been shown that different conditions apply to different kinds of encoding and decoding operations. Specifically, two kinds of operations can be distinguished.

For what concerns marshaling and unmarshaling operations applied when sending and receiving messages on public channels, a refined protocol model corresponding to a typical layered implementation of such operations has been defined. It has been proven that, in order to verify secrecy or authentication properties on the refined model, it is enough to

verify those properties on the corresponding abstract model, provided that the intruder knows the encoding parameters, which is a reasonable assumption. Alternatively, in order to check a generic security property defined on protocol traces, still a simplified refined model, that excludes explicit models of marshaling and unmarshaling functions but preserves the parameters of such operations, can be used in place of the full one.

The model of encoding schemes that has been developed in this work is general enough to take into account a widely used class of encoding schemes and implementations, namely the memoryless and side effect free ones. Moreover, on the marshaling and unmarshaling operations, no assumption has been made about invertibility, nor about implementation correctness; instead, it is only required that the implementation of encoding and decoding functions satisfies some data flow properties, that can be checked by standard static analysis techniques. The consequence of this result is that, if such data flow properties are satisfied on implementation code, then even erroneous specifications or implementations of encoding schemes cannot be more harmful than a Dolev-Yao intruder is.

For what concerns instead encoding and decoding operations made on key material or on data on which cryptographic operations are applied, a similar general model supporting a wide class of data encoding schemes has been introduced. On this model, it has been shown that the models of the decoding operations can be safely omitted when verifying security properties, but under stricter constraints: it is required that the implementations of encoding functions are the inverses of the corresponding decoding functions for each actor. If secrecy is being verified, then the encoded form of messages can also be safely abstracted away, by additionally checking that the implementation of the encoding and decoding functions is correct w.r.t. their specification. If instead authentication is being verified, then the encoded form of messages can be abstracted if all actors agree on the same encoding parameters too. This condition can be checked as part of the abstract protocol verification.

Therefore, an important distinction in criticality has been shown to exist between channel marshaling and unmarshaling transformations, for which neither invertibility nor correctness are needed, and encoding and decoding operations applied to key material or to data on which cryptographic operations are applied, for which they are needed in order to use the fully abstract model.

A previous work related to our approach is [43], where it is shown that any Dolev-Yao model of a WS-Security protocol, where the XML encoding is embedded into the datatype, can be simplified into a more abstract protocol, still preserving secrecy and agreement, by abstracting away XML tag encodings and just keeping the contents of XML elements. It turns out that the results being presented here generalize in two directions the work in [43]. On the one hand, the work presented here takes the implementation of encoding functions into account, and is not limited to consider only how they are specified. On the other hand, the results proposed here apply to any possible encoding scheme, XML and WS-Security being just a particular instance of it.

Although some of the results presented here are probably not so surprising, all of them have been formally proven for the first time in this work, and they constitute a formal background useful in designing new security protocols, and in getting some insights on how data encoding can interact in currently deployed protocols. Also, these results

find application in improving the development of formally verified implementation code of security protocols, both using the code generation approach or the model extraction approach.

For example, for any abstract protocol model where the intruder is assumed to know the encoding parameters used in marshaling and unmarshaling operations on transmitted and received data, if this abstract model has already been verified to be secure with respect to secrecy or authentication under a Dolev-Yao intruder, then any corresponding refined model taking into account the encoding and decoding transformations as modeled in this work is now implied to be secure too, with no other effort required. This result is especially useful in code generation, because the developer only needs to write and verify the abstract protocol model, and the code generation engine can take care of ensuring that the generated code meets the data flow assumptions made about the refined model. So, an important step towards a formally safe refinement process in methods based on code generation has been made.

A similar approach can be used to deal with encoding and decoding operations made on data on which cryptographic operations are applied. In this case, we can safely analyze secrecy and authentication on the abstract model only, provided that we can show that the implementation of the encoding and decoding functions is correct with respect to their definition and that they satisfy an information flow property. Alternatively, an intermediate model where encoding operations are explicitly modeled while decoding ones are not can be safely used under the weaker constraint of ensuring or assuming that the implementation of encoding is the inverse of the implementation of decoding for that specific actor. By the way, it has been shown that, at least for an important class of cases, the check on the correctness of the encoding and decoding functions is viable.

When adopting a model extraction approach, the results presented here make it possible to avoid extracting from code a complex model that represents all the implementation details of encoding and decoding functions. Instead, a simpler model can be extracted, provided that some static checks are first made on the implementation code.

We can expect that, by abstracting away implementation details from the model, a reduction in the time needed for formal verification and the possibility to analyze more complex protocols are finally achieved. It may be argued that, in order to abstract details away, some properties have to be verified on the implementation code. This is true, however, the implementation code to be checked is sequential and independent from the intruder, and thus simpler to be checked in isolation than it is checking a full protocol model, which is a concurrent system that includes a detailed model of the encoding and decoding functions and an intruder.

Moreover, it should be considered that sometimes the source code that implements encoding and decoding functions may not be available (e.g. for libraries). In such cases, code extraction is even impossible. The results presented here show what are the requirements on this code, which can be used to derive specific testing procedures in order to get a good assurance level. In order to obtain the results achieved in this work, some general extensions to the results presented in [36] have been made (see appendix C for further details). While these extensions are immediately useful in this work, they are also stand-alone achievements that can be used as a starting point for future works.

Some issues on the topics presented here are still open for future work. In particular, it can be interesting to explore under which conditions the final transformation back to the original abstract model is safe for other security trace properties or for security properties not defined on traces (e.g. strong secrecy). Another interesting further work is to consider verification with computational models instead of verification with Dolev-Yao models.

Chapter 4

An SSH Transport Layer Protocol Client Example

In order to validate the MDD methodology illustrated in chapter 2, this chapter presents a case study where a model of the SSH Transport Layer Protocol (SSH-TLP) [77] is defined and formally verified, and an implementation of an SSH-TLP client is soundly derived from its model. The design and development of both model and application are performed with the support of state of the art tools. Formal verification of the specification is done with the ProVerif tool [22], while implementation generation is supported by the Spi2Java framework [55].

The main contribution and motivation of this chapter is to show the viability of the semi-automatic code generation approach by handling the full version of a real protocol. Furthermore, the lessons learned about strengths and weaknesses of the approach are also reported.

For brevity, only part of the full SSH-TLP formal model is reported here. The full model that can be formally verified, and the source code of the generated client can be found in [55].

The remainder of this chapter is organized as follows. Section 4.1 illustrates a possible formal model for the SSH-TLP, while section 4.2 shows how this model is formally verified. Section 4.3 describes the steps needed for semi-automatic generation of an SSH-TLP client from its specification, and section 4.4 gives its interoperability and performances testing results. Finally section 4.5 concludes.

A preliminary and stripped-down version of the work presented here appeared in [56].

4.1 The SSH-TLP Formal Model

As prescribed by the Spi2Java MDD methodology, the spi calculus language is used to specify formal models of security protocols. In typical client-server protocols, like the SSH-TLP one, several client and server instances run concurrently participating in protocol sessions. In order to model this, a top level process $Inst$ is usually defined such that it instantiates an unbounded number of client and server processes, thus generating the

protocol sessions. For example, if C and S are the client and server processes respectively, then the $Inst$ process is defined as

$$Inst \triangleq !C \parallel !S$$

Note that $!(C \parallel S)$ would not be the same, and it would not be correctly modeling reality, as it would implicitly couple server and client instances.

During formal verification of the SSH-TLP, the $Inst$ process is analyzed, in order to reason on the overall behavior of the protocol, checking whether it guarantees the desired security properties. However, when deriving the implementation from the formal model, only the client (resp. server) process is implemented in isolation. Running several instances of the generated and third party client (resp. server) implementations will simulate the formal $Inst$ process.

The SSH-TLP clients and servers all communicate over a public channel c. Moreover, each client and server process also has its own private ks channel, used to exchange data with its key store.

The SSH-TLP protocol is informally specified in [76, 77]. For the sake of clearness, a typical SSH-TLP scenario is provided in figure 4.1. In the first two messages, client

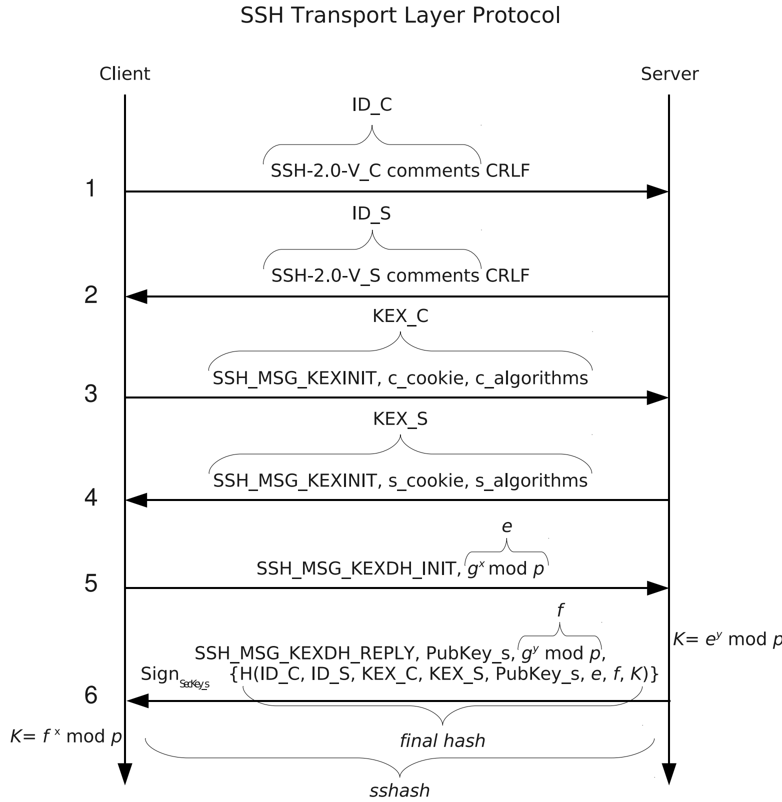

Figure 4.1: SSH Transport Layer Protocol typical scenario.

and server begin a protocol session by exchanging their identifiers V_C and V_S respectively. Then, with messages three and four, client and server negotiate session algorithms by exchanging two random nonces (*c_cookie* and *s_cookie*) and their lists of supported algorithms (*c_algorithms* and *s_algorithms*), from which the agreed ones will be selected. Finally, with the fifth message, the client sends its Diffie-Hellman (DH) public key e to the server, and the server replies with the last message, providing its public key $PubKey_s$ used for digital signature, its DH public key f, and the signed *final hash*, upon which client and server perform authentication agreement. At the end of the protocol, both client and server can derive a shared DH session secret K, that is later used to generate several session keys.

In order to enable formal verification, a spi calculus model of the full SSH-TLP protocol, including client, server, the key stores, other auxiliary processes and the $Inst$ processes, has been manually written. This model differs from already available models (e.g. the ones provided with the ProVerif tool, or by the AVISPA project) in two relevant aspects. On one hand, in this work interaction between protocol agents and their key stores is fully modeled. Moreover, error states of the protocol are handled, and proper error messages are emitted, thus enabling formal verification of error situations too. Finally, algorithms negotiation is modeled, and agreed algorithms are taken into account in both cryptographic primitives and in data upon which authentication agreement is done. Up to our knowledge, this is the first SSH-TLP model including all these kinds of details. On the other hand, the proposed model includes some specific aspects that are irrelevant to formal verification (that is, a model without these aspects would be semantically equivalent), but that are essential to derive a concrete Java implementation from it.

In its full length, the model discussed in this work counts 261 lines. For brevity, only the client process is presented and discussed here. The complete protocol specification can be found in [55].

Using a syntax accepted by the Spi2Java parser, plus some syntactic sugar to enhance readability, a possible spi calculus specification of the SSH-TLP client agent follows:

```
 1: sshClientTLP(cAlgs, ID_C, cAB, cState, keystore, cast) :=
 2:    cAB<ID_C>.
 3:    cAB(ID_S).
 4:    (@c_cookie)
 5:    let KEX_C = (SSH_MSG_KEXINIT, c_cookie, cAlgs, first_kex_packet_follows) in
 6:    cAB<KEX_C>.
 7:    cAB(KEX_S).
 8:    let (x_SSH_MSG_KEXINIT, s_cookie, x_sAlgs, x_first_kex_packet_follows) = KEX_S in
 9:    [ SSH_MSG_KEXINIT is x_SSH_MSG_KEXINIT ]
10:    (
11:        let (c_kex_algorithms, c_server_host_key_algorithms, c_encryption_algorithms_client_to_server,
12:             c_encryption_algorithms_server_to_client, c_mac_algorithms_client_to_server,
13:             c_mac_algorithms_server_to_client, c_compression_algorithms_client_to_server,
14:             c_compression_algorithms_server_to_client) = cAlgs in
15:
16:        let (s_kex_algorithms, s_server_host_key_algorithms, s_encryption_algorithms_client_to_server,
17:             s_encryption_algorithms_server_to_client, s_mac_algorithms_client_to_server,
18:             s_mac_algorithms_server_to_client, s_compression_algorithms_client_to_server,
19:             s_compression_algorithms_server_to_client) = x_sAlgs in
20:
21:        //Extract Alg
22:        let agreed_kex_algorithms =
```

```
23:                H((c_kex_algorithms, s_kex_algorithms)) in
24:        let agreed_server_host_key_algorithms =
25:                H((c_server_host_key_algorithms, s_server_host_key_algorithms) in
26:        let agreed_encryption_algorithms_client_to_server =
27:                H((c_encryption_algorithms_client_to_server, s_encryption_algorithms_client_to_server) in
28:        let agreed_encryption_algorithms_server_to_client =
29:                H((c_encryption_algorithms_server_to_client,s_encryption_algorithms_server_to_client) in
30:        let agreed_mac_algorithms_client_to_server =
31:                H((c_mac_algorithms_client_to_server, s_mac_algorithms_client_to_server) in
32:        let agreed_mac_algorithms_server_to_client =
33:                H((c_mac_algorithms_server_to_client, s_mac_algorithms_server_to_client) in
34:        let agreed_compression_algorithms_client_to_server =
35:                H((c_compression_algorithms_client_to_server, s_compression_algorithms_client_to_server) in
36:        let agreed_compression_algorithms_server_to_client =
37:                H((c_compression_algorithms_server_to_client, s_compression_algorithms_server_to_client) in
38:
39:        cast<H(agreed_kex_algorithms,1)>. cast(g).
40:        cast<H(agreed_kex_algorithms,2)>. cast(p).
41:        cast<H(agreed_kex_algorithms,3)>. cast(q).
42:        cast<H(agreed_kex_algorithms,4)>. cast(DHHash).
43:
44:        cast<H(agreed_server_host_key_algorithms,1)>. cast(csSignHash).
45:        cast<H(agreed_server_host_key_algorithms,2)>. cast(csSignKeyType).
46:        cast<H(agreed_server_host_key_algorithms,3)>. cast(csSignMode).
47:        cast<H(agreed_server_host_key_algorithms,4)>. cast(csSignPadding).
48:
49:        (@x)
50:        cAB<(SSH_MSG_KEXDH_INIT, DHPub(x))>.
51:        cAB((x_SSH_MSG_KEXDH_REPLY, PubKey_s, f, sshash)).
52:
53:        [ x_SSH_MSG_KEXDH_REPLY is SSH_MSG_KEXDH_REPLY ]
54:        (
55:
56:           keystore<(CHECK_IP,(ID_S,cAB))>.
57:           keystore(KeystoreResult).
58:
59:           [ KeystoreResult is keystore_SUCCESS ]
60:           (
61:              keystore<(GET_IP,(ID_S,cAB))>.
62:              keystore(stored_PubKey_s).
63:              [ stored_PubKey_s is PubKey_s ]
64:                (
65:                   case sshash of [{shash}]PubKey_s in
66:
67:                   [ shash is H(H((ID_C, ID_S, KEX_C, KEX_S, PubKey_s, DHPub(x), f, DHKey(x,f)))) ]
68:                   (
69:                     cState< description_CLIENT_SIDE_PROTOCOL_END_CORRECTLY >.
70:                     0
71:                   )
72:                     else
73:                   (
74:                     cAB< (SSH_MSG_DISCONNECT, SSH_DISCONNECT_KEY_EXCHANGE_FAILED,
75:                           description_KEY_EXCHANGE_FAILED, language_tag) >.
76:                     cState< description_KEY_EXCHANGE_FAILED >.
77:                     0
78:                   )
79:                )
80:              else // of [ stored_PubKey_s is PubKey_s ]
81:                (
82:                   cAB< (SSH_MSG_DISCONNECT, SSH_DISCONNECT_PROTOCOL_ERROR,
83:                         description_PUBKEYS_DO_NOT_MATCH, language_tag) >.
84:                   cState< description_PUBKEYS_DO_NOT_MATCH >.
```

```
 85:                   0
 86:                )
 87:           )
 88:           else // of [ KeystoreResult is keystore_SUCCESS ]
 89:           (
 90:                   cAB< (SSH_MSG_DISCONNECT, SSH_DISCONNECT_HOST_KEY_NOT_VERIFIABLE,
 91:                         description_HOST_KEY_NOT_VERIFIABLE, language_tag) >.
 92:                   cState< description_HOST_KEY_NOT_VERIFIABLE >.
 93:                   0
 94:           )
 95:
 96:       )
 97:       else // of [ x_SSH_MSG_KEXDH_REPLY is SSH_MSG_KEXDH_REPLY ]
 98:       (
 99:          cAB< SSH_MSG_UNIMPLEMENTED >.
100:          cState< description_MSG_KEXDH_REPLY_EXPECTED >.
101:          0
102:       )
103:     )
104:     else // of [ SSH_MSG_KEXINIT is x_SSH_MSG_KEXINIT ]
105:     (
106:          cAB< SSH_MSG_UNIMPLEMENTED >.
107:          cState< description_MSG_KEXINIT_EXPECTED >.
108:          0
109:     )
```

The model is now briefly commented, while some modeling aspects are discussed in details.

From line 1 to line 20 the first four messages are exchanged, and the lists of algorithms to be agreed upon are parsed (by the pair splitting processes). Note that in this model, it is assumed that the `first_kex_packet_follows` flag is always set to false, so no guessed key exchange packet will be sent. This assumption allows the model to be significantly simpler: handling it would require to possibly discard received messages or to send some messages twice on the basis of the outcome of algorithm negotiation. This flag was meant to optimize implementations performances; in fact, its handling is rather complex, and this option is hardly ever used in real implementations indeed.

Client and server must agree on ten different algorithms. For each algorithm that must be agreed on, both client and server propose a list of supported algorithms, ordered by preference. Each algorithm is finally agreed by choosing the first algorithm in the corresponding client list that is also available in the corresponding server list. In the model, the ten lists of client supported algorithms are packed into the `cAlgs` term, and the server ones into the `x_sAlgs` term.

Algorithms negotiation is in fact performed at lines 21-37. The negotiation procedure for each one of the ten algorithms to be agreed is abstractly modeled by an hash function, operating on each client-server pair of algorithms lists. Indeed, algorithm agreement "transforms" the pair of algorithms lists into the agreed algorithm. Once the agreed algorithm is known, no other information is available about the original pair of algorithms lists, except they both contained the agreed algorithm. Modeling this with a non invertible hash function may seem too constraining for the attacker knowledge, because, given an agreed algorithm in the hash form, it could not infer anything on the originating lists

pair. However, this is not true for the SSH-TLP, and this also holds for many other protocols, because the full algorithms lists are already known by the attacker anyway, either because being sent cleartext over the insecure medium, or because being publicly known (e.g. available algorithms are listed in the RFC).

At lines 39-47, the client extracts the cryptographic parameters that depend on the agreed key exchange and server host key algorithms, and assigns to each of them a significant name. Diffie-Hellman parameters (that is `g`, `p`, `q`, and `DHHash`) are obtained from `agreed_kex_algorithms`, while the signature algorithm parameters (that is `csSignHash`, `csSignKeyType`, `csSignMode`, and `csSignPadding`) are obtained from `agreed_server_host_key_algorithm`

In order to extract several parameters from the same spi calculus term (e.g. `g`, `p`, `q`, and `DHHash` from `agreed_kex_algorithms`), an hash function is performed on a pair containing the agreed algorithm and an integer number. The integer number works as a parameter for the hash function, identifying the cryptographic parameter to be extracted. Again, the hash abstraction is safe, since the cryptographic parameters are already known by the attacker anyway.

Note that at lines 39-47 the significant name is not assigned by a plain `let` process; instead, a less usual output/input pattern is used. This is needed because spi calculus is an untyped language while Java is statically typed. So, a type must be statically assigned to each spi calculus term before a Java program can be derived. It is worth pointing out that the `let` process only creates a new identifier for the same term, that is, both during formal evaluation and when deriving an implementation, each `let` process will be discarded, and pattern substitution applied. Formally `let n = M in P` will be syntactically transformed in a new process `P'` identical to `P`, where each occurrence of `n` is replaced by `M`. This means that `n` and `M` are the same term, so they are forced to be assigned the same type. This also implies that the `let` process cannot be used to implement type-casts in the derived implementation. Since the DH and digital signature related parameters will require such a type cast, it turns out that the `let` process cannot be used. Instead, the less usual output/input pattern is used. The reacting process (that is the process performing the corresponding input/output operations) is defined as

```
1: castProc(cast) :=
2:   cast(toCast).
3:   cast<toCast>.
4:   0
```

which is just a “relayer”, receiving some data, and forwarding them as is. The semantic effect of this pattern is equivalent to a `let` process, but from a syntactic point of view, now the sent term (e.g. `H(agreed_kex_algorithms,1)` at line 39) is different from the received one (e.g. `g` at line 39), so that they can be assigned two different types. The actual casting algorithm implementation is handled in the marshaling layers of the terms being sent and received over the `cast` channel.

From line 49 to line 54 the two DH key exchange messages are exchanged. At line 56, after the DH key exchange message has been received from the server, the client should check the received server public key `PubKey_s` against the trusted one. This is done by first retrieving the trusted key associated with the connected server from a local key store,

and then by matching the two keys. Interaction with the key store is modeled by pairs of output/input (request/response) message exchanges between the client process and the key store process. This model is flexible enough to allow both formal verification and an implementation derivation. First, the client sends the key store a request, that is a message of the form (*Operand*,*Data*), where *Operand* specifies the requested operation, and *Data* are the application data specific to the given operand. In turn, the key store emits its response by sending an outcome back, whose form depends on the requested operation. In this specification, two operands are used, namely `CHECK_IP` and `GET_IP`. For both operands, application data is a pair containing the server identification string and the public communication channel, which identify the server. When the `CHECK_IP` operand is sent at line 56, the key store replies at line 57 with a boolean answer, telling whether any trusted key associated with the given server has been found. If this is the case (line 59), normal execution continues, otherwise the error handling `else` branch at line 88 is taken. At line 61 the client asks the key store to retrieve the trusted key associated with the server. At line 62 the trusted key `stored_PubKey_s` is read back from the key store, and finally at line 63 the two keys are matched. If the server received key does not match the trusted one, error handling `else` branch is taken at line 80.

At lines 65-67 the client checks that the server signed *final hash* is valid against the locally computed *final hash*. Plain spi calculus only offers basic asymmetric encryption/decryption operations, and not embedded signature check constructs. While this is sufficient to model certain asymmetric encryption schemes such as RSA, special features of other encryption schemes such as DSA would be better modeled with a special signature check process, taking a certificate, a message and its signature, and returning a boolean value telling whether the signature is valid with respect to the given message. While it is believable that all the results presented in this work still hold if this special signature check process is added, this case study only uses the plain spi calculus, so RSA is the only asymmetric encryption scheme supported, while DSA is not. Adding the special process in order to handle DSA would increase the complexity of the model more than its significance.

By using the plain spi calculus processes, the signature check steps are explicitly represented in the specification. In SSH-TLP, digital signature is performed by first hashing data to be signed, and then applying private key encryption. Therefore, the client first decrypts the server signed final hash `sshash` with the server public key `PubKey_s`, and stores the result into the `shash` variable (line 65). Then at line 67 `shash` is matched against the locally reconstructed final hash. Note the double hashing: the outer hash is the digital signature mandatory hash, performed on the signed data; the inner hash is actually the final hash, which includes all relevant session data. Again, cryptographic algorithms and parameters will be added during formal verification and implementation derivation.

If the final hashes match, protocol session ended well, and the session secret is established as `DHKey(x,f)`, computed as $f^x \bmod p$. Note that in plain spi calculus, the `DHPub()` and `DHKey()` terms are represented as hashes (specially handled during formal verification). Here they are syntactically distinguished too, in order to improve understandability of the model.

Upon successful completion of the protocol session, at line 69 a message is sent on the

`cState` channel to signal the success condition to the caller. Else, at line 74, the error condition is handled. In general, error conditions are handled by sending the appropriate SSH-TLP error message to the server, and by sending another message to the `cState` channel, to signal the error condition to the caller.

4.2 Formal Verification of the SSH-TLP Model

Two techniques are mainly used for the automatic formal verification of Dolev-Yao models, namely model-checking [10] and (automated) theorem proving [16, 22]. Several tools are available for both the former (e.g. [49, 28, 48, 71]) and the latter (e.g. [13, 6, 22]) techniques. In this work the ProVerif theorem prover [22] is used. It has been chosen over other available tools for several reasons. First, being a theorem prover, it can handle an unbounded number of protocol sessions, which is usually not the case with model-checkers. Often, (generic) theorem provers have the drawback of requiring manual interaction, and, conversely to model-checkers, they do not provide counter-examples when a security property is proven to be violated. ProVerif instead, by being specific to security protocols, offers full automation and can often report counter-examples. Lastly, both ProVerif and Spi2Java accept (variations of) the same input language, making integration of the tools rather straightforward.

The SSH-TLP model presented in the previous section has been formally verified for secrecy and server authentication. As illustrated in the previous section, the model uses the syntax accepted by the Spi2Java framework, which is slightly different from the ProVerif accepted syntax. Fortunately, Spi2Java comes with a tool, called *Spi2Proverif*, that translates specifications form the Spi2Java syntax to the ProVerif one. Moreover, during translation, if a proper eSpi document is provided, the tool also adds all cryptographic details that were left implicit in the Spi2Java model, so that they are taken into account during verification.

On the basis of the results shown in section 3.2, marshaling and encoding functions can be completely abstracted away, assuming correctness of their implementation and by noting that secrecy and authentication are proven. Cryptographic algorithms and parameters are explicitly included in the model instead, so that the expression of the authentication property can be simplified.

For example, the eSpi document excerpt corresponding to the final hash

```
H((ID_C, ID_S, KEX_C, KEX_S, PubKey_s,DHPub(x), f, DHKey(x,f)))
```

declared at line 67, is the following (with minor modifications to improve readability):

```
<term id="403"
      name="H((ID_C, ID_S, KEX_C, KEX_S,
               PubKey_s,DHPub(x), f, DHKey(x,f)))"
      type="Cryptographic Hashing">
  <codify>CryptoHashingSR</codify>
  <parameters>
    <param name="algorithm" type="var">DHHash</param>
```

```
      <param name="provider" type="const">SUN</param>
    </parameters>
  </term>
```

The `id` attribute univoquely identifies the term in the XML document, while `name` is a human readable form. The `type` attribute refers to the Java type assigned to the term, and the `codify` element refers to the marshaling layer, described later. The `parameter` element instead is the relevant part where cryptographic algorithms and parameters are specified. For this hash, the `algorithm` parameter is of variable (`var`) type, meaning it will be resolved at run-time: `DHHash` is the term whose content (e.g. SHA-1, or SHA-256) will be used as the hashing algorithm at run-time. Conversely, the `provider` parameter, indicating the Java Cryptographic Architecture (JCA) provider to be used, is of constant (`const`) type, meaning its value is assigned at compile-time, and is `SUN` in this example. So, the Spi2Proverif tool, according to the spi calculus specification and the eSpi document, will translate the final hash into the following term:

```
H( (ID_C, ID_S, KEX_C, KEX_S, PubKey_s,DHPub(x),f,DHKey(x,f)),
   (DHHash,SUN) )
```

That is an hash taking two parameters: the first one is the data to be hashed, while the second one encodes the hashing algorithm and the JCA provider by a pair containing the hash-related parameters, specified in the eSpi document.

As another example, the external hash at line 67 performed on the final hash, and required by the digital signature scheme, is translated into

```
H( H( (ID_C, ID_S, KEX_C, KEX_S, PubKey_s,DHPub(x),f,DHKey(x,f)),
      (DHHash,SUN) ),
   (csSignHash,SUN) )
```

Note how different hashings can then use different algorithms and parameters, either negotiated at run-time or known at compile-time.

Another relevant modeling strategy is the DH key exchange. ProVerif supports DH modular exponentiation by defining two functions, `DHPub` for public part generation, and `DHKey` for the shared secret derivation. These two functions are finally related in ProVerif by the linear DH equation $DHKey(x,DHPub(y)) = DHKey(y,DHPub(x))$ [24]. Note that at the implementation level, the `g`, `p` and `q` parameters required to actually compute the DH values are referenced as run-time cryptographic parameters of the spi calculus terms representing the DH operations.

As the MDD workflow implies, the original spi calculus model only includes protocol instances and actors specification, but nothing is stated about the intended security properties that should be satisfied by the model. So, once the model is refined and translated to the ProVerif syntax, it is still necessary to add the relevant security queries. Both secrecy of the derived DH shared secret, and injective server authentication by agreement on session data have been verified on the model.

The secrecy-related queries are quite simple. Let `x` be the client DH secret, and `y` be the server DH secret. On the client side, the shared DH secret is computed as

`DHKey(x,DHPub(y))` (where the client only knows `DHPub(y)`, and not `y`). So, the ProVerif query

```
query attacker:DHKey(x,DHPub(y)).
```

asks ProVerif to check whether the shared DH secret, as computed by the client, remains secret indeed. In order to speed up verification, two secrecy assumptions are made, namely that both `x` and `y` are not known to the attacker. ProVerif verifies that the secrecy assumption are in fact verified, before proceeding with formal verification.

On the server side, the shared DH secret is computed as `DHKey(y,DHPub(x))` (again, the server only knows `DHPub(x)`, and not `x`). Technically, the client-side query above is enough to prove secrecy because, due to the equational system, also the server leaking its own copy of the shared DH secret would be recognized. Nevertheless, it is not much effort (nor a sensible amount of verification time) to add a server-side secrecy query.

The server authentication property is expressed in ProVerif by means of injective agreement on relevant session data [23]. Briefly, injective agreement means that for each time client C believes it has finished a protocol session with server S agreeing on some data M, then server S started a protocol session with client C, agreeing on the same data M. More details on injective agreement can be found in [47].

Two agreement events are defined: `beginServerEnd(x)`, emitted by the server when it believes it has started a session with the client, agreeing on data `x`; and `endServerEnd(x)` emitted by the client when it believes it has finished a session with the server, agreeing on some data `x`. Then, injective agreement is expressed by the following ProVerif query:

```
query evinj:endServerEnd(x) ==> evinj:beginServerEnd(x).
```

Special care must be taken about where to put the agreement events. In general, the "begin" event should be put in the server when all relevant session data are available, but before the last exchanged message, while the "end" event should be put in the client at the end of the protocol run. In SSH-TLP, the server emits the `beginServerEnd` event just before sending its DH key exchange message, which is the last message of the typical scenario in figure 4.1. Since cryptographic algorithms and parameters are explicitly represented in the model and the negotiation algorithm is explicitly modeled too, it is enough to only use the final hash as the `beginServerEnd` event argument. Indeed, the final hash contains server and client identities, all the exchanged messages, the shared DH secret and, thanks to the Spi2Proverif added refinement information, the agreed algorithms too. The client instead emits the `endServerEnd` event on the same final hash, only after it has checked the received signed final hash against the locally reconstructed one, that is between lines 68 and 69.

It is worth pointing out that if the security queries are added to the plain model generated by Spi2Proverif, ProVerif will not be able to prove any property (by not terminating in a reasonable time – days). It turns out that the unusual way in which type-casts have been modeled at lines 39-47 is stressing the ProVerif over approximation techniques, not allowing any result to be obtained. As hinted in the previous section, although the unusual

modeling of type-casts is necessary when deriving the implementation, during formal verification these type-casts assignments can be substituted by formally equivalent plain `let` processes, that are much better handled by ProVerif. For example, line 39 becomes

```
let g = H(agreed_kex_algorithms,1) in
```

Note that `let` assignments and unusual type-casts are semantically equivalent, so substituting the latter with the former is formally justified. Indeed, a let process evolves like

$$let\ n = M\ in\ P \rightarrow P[^{M}/_{n}]$$

Analogously, a pair of output/input operations on the private `cast` channel reacts with the `castProc` process described in the previous section in the following way

$$cast\langle M\rangle.cast(n).P \rightarrow^{*} P[^{M}/_{n}]$$

4.2.1 Checking and Debugging the Model

Safety properties are trivially true when they are verified on erroneous protocol models that never reach the end of any protocol session. This subtle aspect of safety properties (such as secrecy and authentication) is often neglected in small, proof-of-concept models, but it becomes relevant in large size protocol models, such as the SSH-TLP one described here (consider that the ProVerif version, including security queries, is 433 lines long).

For example, suppose an erroneous SSH-TLP server model that computes a wrong final hash, by first including the server identity, and then the client one (as opposite to the correct final hash, that first includes the client identity, and then the server one). In this setting, the match case at line 67 in the SSH-TLP client model would always fail, and the `endServerEnd` event would never be emitted. When analyzing this erroneous model for authentication, the result of formal verification would state that the authentication property is preserved, because *each* `endServerEnd` event (*no* event in this case) has a corresponding `beginServerEnd` event. Notwithstanding this result, the real protocol could still be flawed with respect to authentication.

Informally, a liveness property like "correct protocol sessions should reach the end" should also be taken into account when proving safety properties. However, in a Dolev-Yao environment, where the attacker can always drop messages, it is impossible to prove such liveness properties. Nevertheless, here an approach is proposed where safety properties are exploited, so that such erroneous models can be recognized and corrected with a high level of confidence. The proposed approach can be practically implemented with ProVerif, and has been used on the verified SSH-TLP model. Up to our knowledge, this is the first documented attempt to exploit safety properties to give some confidence about correctness of security protocol models.

The approach consists in putting special events at the end of each actor's protocol run, and to formally verify whether all these special events are executed at least once (which is a safety property). For example, in SSH-TLP, two events have been defined, namely `DEBUGc` for the client, and `DEBUGs` for the server. The `DEBUGc` event has been put in the

client model between lines 68 and 69 (after the `endServerEnd` event), and similarly for the server side. Then, two ProVerif queries have been added:

```
query ev:DEBUGc().
query ev:DEBUGs().
```

Each of these queries is true if the corresponding event is never emitted, false otherwise. Now, if for example the `DEBUGc` query is true, it means that this is an erroneous model, because the client can never reach the end. So, any other security property result may be trivially true, thus not valid. In this case, it is possible to move the `DEBUGc` event upward in the specification, until it becomes false, thus effectively identifying the point where the process becomes erroneously stuck, in fact being a (simplistic) form of debugging for formal models.

4.3 Client-Side Model Refinement and Implementation Generation

This section shows how an executable Java implementation for the client-side of the SSH-TLP can be derived by refining the formal model presented in section 4.1. According to the MDD methodology presented in chapter 2, for each spi calculus term belonging to the abstract model, three kinds of refinement information must be added in order to derive the Java implementation: an eSpi type; the cryptographic algorithms and parameters; and an encoding layer.

eSpi Types and Cryptographic Algorithms and Parameters

In order to complete the formal verification of the SSH-TLP model in section 4.2, some terms already had their type and corresponding cryptographic algorithms and parameters assigned (and discussed in this work). For the remaining terms, this piece of information must be manually refined now, if needed, before an interoperable Java implementation can be generated. Essentially, many names implicitly used as nonces must be manually downcast to the "Nonce" type, so that their associated size can be specified, according to the informal SSH-TLP RFC. Moreover, channels must be downcast to a more specific type; for example, `cAB` is cast to the "TCP/IP Channel" type, or `cState` to the "File Channel" type, so that the protocol outcome is written to a file.

Note that only the `sshClientTLP` process is considered when deriving the client-side Java implementation. The server and the *Inst* processes are obviously not needed; moreover the key store and cast auxiliary processes are implicitly implemented locally within the client-side application, and there is no need to generate an implementation for them.

Within the SpiWrapper library, the key store and cast processes functionalities are implicitly implemented by the Java classes implementing the communication channels used by the client. In particular, the `keystore` spi calculus channel is assigned the type "Java Key Store Channel" in the eSpi document. The SpiWrapper class implementing this type of channel offers the typical `send` and `receive` methods, that are used by the automatically generated protocol logic to interact with the key store using the request/response

paradigm. However, the logic behind the `send` and `receive` methods does not send anything over the network, instead it actually deals with a local Java key store, which is part of the JCA.

Analogously, the `cast` spi calculus channel is assigned the type “Cast Channel” in the eSpi document. On a `send` invocation, the channel will locally buffer the sent term in its marshaled form, and on a `receive` invocation it will return the previously buffered message, that will be unmarshaled in the received term.

Here, it is assumed that both the key store channel implementation and the cast channel implementation correctly refine the behavior of their spi calculus models. For the key store implementation, this assumption is reasonable because only a small amount of “glue-code”, interfacing the `send` and `receive` methods with the Java key store must be trusted. Being part of the JCA, the Java key store is already assumed to be correct, like any other cryptographic function implementation belonging to a JCA provider. Regarding the cast channel, it is reasonable to assume that the buffering strategy described above is a correct refinement of the “relaying” behavior described in the formal model.

Finally, the algorithm negotiation procedure has been abstractly modeled by means of hashing operations. Now, an implementation of that procedure must be provided. Thanks to the modular eSpi type system, two sub-types of the “Hashing” type are defined, namely “Extract Alg” and “Extract Param”. In the model, all the hashing operations made form line 21 to line 37 are assigned the “Extract Alg” type. This type expects the message to be hashed to be a pair consisting of two lists: the client-preferred list of algorithms, and the server-preferred list of algorithms. The result of the hash will be the agreed algorithm.

Likewise, all the hashing operations made from line 39 to line 47 are assigned the “Extract Param” type. This type expects the message to be hashed to be a pair composed of an (agreed) algorithm, and an integer (or more generically, a marker). The result of the hash will be the value of a cryptographic parameter according to the given algorithm and the marker.

Note that this modeling strategy is generic and modular: provided the algorithm negotiation procedure can be modeled by a hash in the first place (discussed in section 4.1), switching the algorithm negotiation procedure accounts to only update the implementation of the “Extract Alg” and “Extract Param” types, while the abstract model remains unchanged. This modeling strategy can also be re-used in other protocol models too.

Encoding Layer

The last kind of refinement information that must be provided in order to get an interoperable implementation concerns the encoding layer. For this case study, an encoding layer implementing the conversions between Java types and the SSH-TLP binary packet protocol [77] and SSH-TLP specific types representations [76] has been implemented. The implementation work was rather straightforward, although the required development time was not negligible compared to the other modeling or refinement steps.

Once the encoding layer is available, the eSpi document is updated. Each term that requires the custom SSH-TLP encoding layer has the corresponding `codify` element modified, so that it points to the proper Java class implementing the SSH-TLP encoding layer.

Note that some terms that are never sent over channels (such as, for example, some constants or the channels themselves) do not need to use any special encoding layer, and the default one can be safely used.

Implementation Generation and Execution

When the spi calculus model and the eSpi document containing refinement information are ready, the Spi2Java code generator can be used to automatically generate the Java code implementing the protocol logic, and the whole application skeleton.

Before running the client however, the spi calculus process input arguments, with the exception of channels, must be manually initialized, since their value cannot be automatically inferred. In this application, the list of client-preferred algorithm lists `cAlgs`, and the client identity `ID_C` must be initialized.

It turns out that having the `cAlgs` term to be manually initialized is rather convenient. Depending on the context where the application will be deployed, the content of `cAlgs` can be hard-coded once, or conversely, before the automatically generated protocol logic is run, the user can be interactively prompted at run-time for the algorithms to be included into `cAlgs`, making the final application highly configurable.

4.4 Experimental Results

4.4.1 Interoperability and Reliability

The generated SSH-TLP client has been tested against seven third party server implementations; five kinds of sessions have been executed with each server, totalizing 35 experiments. Since client-preferred algorithms drive the negotiation procedure, in each kind of session the client lists of preferred algorithms have been properly configured, such that different algorithms would have been negotiated.

Table 4.1 shows, for each kind of session, the significant lists of preferred algorithms that the client sends to the server.

Kind of session	Signature	DH group	Final Hash
1	RSA; DSA	1; 14	SHA-1
2	RSA; DSA	14; 1	SHA-1
3	DSA; RSA	1; 14	SHA-1
4	RSA; DSA	1	SHA-1
5	RSA; DSA	14	SHA-1

Table 4.1: Lists of preferred algorithms

In sessions of kind 1, 2, and 3, the client sends to the server a list with all the algorithms that the SSH-TLP requires to be supported by the actors. If the server supports at least one of the algorithms proposed by the client, then the negotiation algorithm is expected to terminate with success. When the server supports both RSA and DSA keys, sessions

of kind 1 and 2 must end well, while sessions of kind 3 are expected to correctly negotiate the algorithms, but then the client fails, due to the unsupported signature algorithm. With servers only supporting RSA keys, all the three kinds of sessions are expected to end correctly, being the RSA signature scheme always negotiated.

In sessions of kind 4 and 5, for "DH group", the client sends to the server a list with only one group. The negotiation algorithm is expected to fail if the server does not exactly support the client requested group, otherwise the session must end well.

The third party servers used for testing are reported, with some comments, in table 4.2. The "Server Name" column reports the advertised name of the server application, while

Server Name	Server ID string	OS	Comments
dropbear 0.52	dropbear_0.52	Linux	No DH G14
freeSSHD 1.2.6	WeOnlyDo 2.1.3	Windows	All correct
KpyM 1.18	cryptlib	Windows	No DH G14 – RSA only
lsh 2.0.4	lshd-2.0.4 lsh - a GNU ssh	Linux	RSA only
MobaSSH 1.12	OpenSSH_5.1	Windows	All correct
OpenSSH 5.1	OpenSSH_5.1p1	Linux	All correct
WinSSHD 5.15	1.04 FlowSsh: WinSSHD 5.15	Windows	All correct

Table 4.2: Tested servers

the "Server ID string" column reports the identification string (ID_S) sent by the server. The "OS" column specifies the operating system on which the server runs on.

Under the "Comments" column, servers with comment "All correct" support all required algorithms, and all kinds of sessions end as expected. In particular, the negotiated algorithms are always the preferred client algorithms.

Servers with comment "RSA only" only support RSA keys. In this case, sessions of kind 3 correctly terminate by agreeing on the RSA signature scheme.

Servers with comment "No DH G14" only support one of the two requested DH groups, namely group 1. With these servers, sessions of kind 1, 3 and 4 end as expected. Sessions of kind 2 end correctly, but the DH group 1 is agreed. Sessions of kind 5 correctly fail instead, because it is impossible to agree on a DH group.

The experiments illustrated here show in fact that the generated client can execute in real environments, because it is able to correctly interoperate with third party implementation servers.

Moreover, the client has been tested against the SSHredder[1] suite, composed of more than 660 incorrect protocol sessions. Each incorrect session has been correctly rejected by the client, which confirms the reliability of the code developed with the proposed methodology.

Some measures of the client code are reported in table 4.3. As previously explained, the spiWrapper package contains the SpiWrapper library, implementing the low-level cryptographic and network operations. The spiWrapperSSH package contains the manual implementation of the SSH-TLP specific encoding layer, and the sshClient package contains

[1] http://www.rapid7.com/sshredder.zip

Package	TLOC[a]	MLOC[b]	Ca[c]
spiWrapper	3768	2180	102
spiWrapperSSH	2564	1136	5
sshClient	648	565	0

[a]Total Lines of Code: non-blank and non-comment lines in a class.
[b]Method Lines of Code: non-blank and non-comment lines inside method bodies of a class.
[c]Afferent Coupling: number of classes outside a package that depend on classes inside the package.

Table 4.3: Measures of the generated client application.

the automatically generated protocol logic.

The measures taken here are coherent with the ones taken in the reference example in section 2.4. This shows that the proposed approach is scalable. With respect to the reference example, when a real protocol is implemented, the code being automatically generated is larger, because the protocol logic is more complex, including algorithm negotiation, key store interaction, type casts and error handling.

Also the manually developed encoding layer becomes significantly larger, because a fully standard-compliant application is to be obtained. Indeed, with some negligible exceptions, the default encoding layer has been completely substituted by the SSH-TLP specific one. The code composing the custom encoding layer accounts for almost a half of the whole application code. In practice this means that when designing and developing a real application with the proposed MDD methodology, a significant amount of time is required to develop the encoding layer. Nevertheless, the implementation work is rather straightforward, and the developer needs only to concentrate on the specific aspect of encoding, which can make development more efficient.

Like in the reference example, the afferent coupling metric shows that the generated application is structured. The automatically generated protocol logic lays at the top-level, coordinating all cryptographic and network operations.

4.4.2 Performances

Execution times of the generated SSH-TLP client and of third party SSH-TLP clients manually developed in Java have been taken. For each client, 1000 runs of the protocol have been executed with the OpenSSH 5.1p1 server. Average execution times and average deviation are reported in figure 4.2. All runs were executed under constant system load and over the localhost, in order to avoid random network delays; the same cryptographic algorithms were always negotiated, namely DH group 1 with RSA server key. In order to eliminate the latency introduced by the Java class loader, for each client the first run is not considered, and all subsequent runs are all executed in the same virtual machine instance. For all clients, the measured time starts at the beginning of the protocol, including TCP handshake and parsing of the key store file, and stops after the digital signature over the final hash and the server key have been checked.

All the clients, including the one developed in this paper, exhibit comparable execution

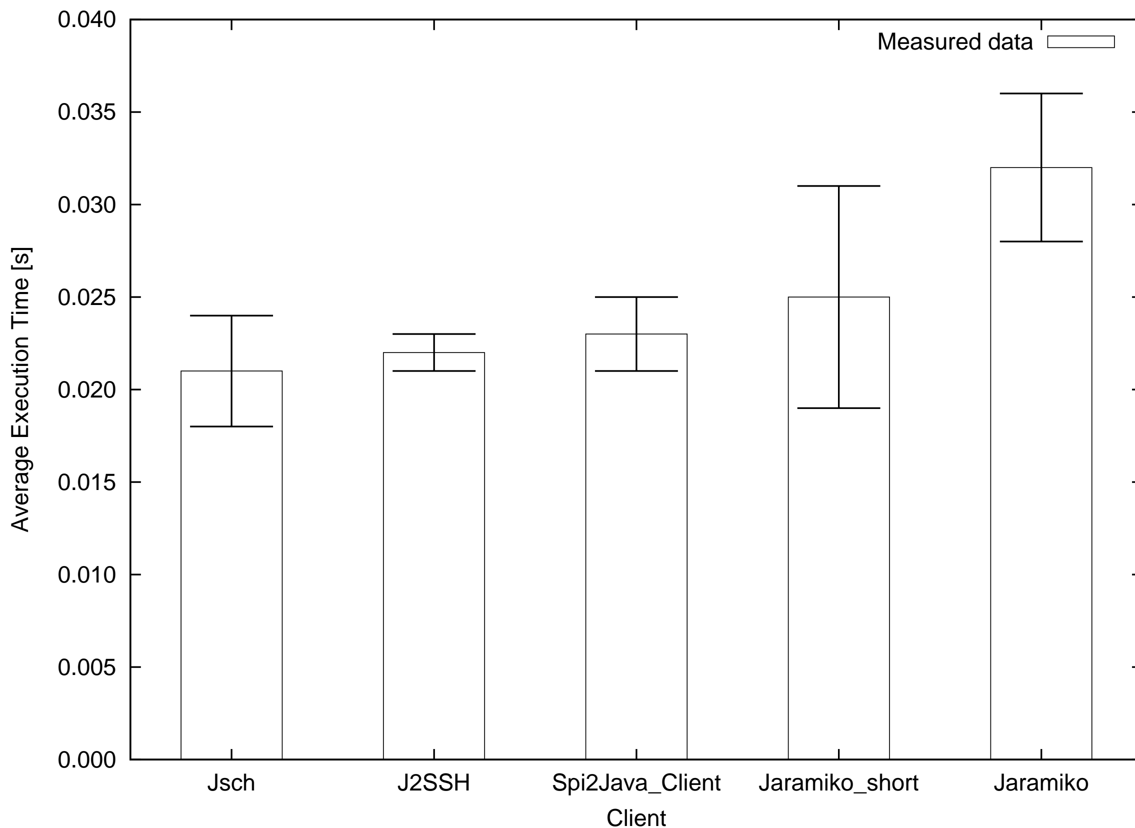

Figure 4.2: Average execution times of different SSH-TLP clients.

times, with the exception of the "Jaramiko" one. In fact, it turns out that the DH modular exponentiation operation is the most computationally intensive one, making all other operations (including other cryptographic operations such as signature check) negligible. In particular, execution time depends on the size, in bits, of the chosen DH secret (`x` in the spi calculus model). Since all clients except the Jaramiko one use the standard JCA implementation to choose the DH secret, it follows that they all execute in comparable time. The Jaramiko client instead uses a larger DH secret on average, making it slower than other clients. This is further confirmed by the fact that when the Jaramiko client was patched to generate DH secrets similar in length to the ones generated by the standard JCA implementation (the "Jaramiko_short" client in the chart), it started executing in comparable times with respect to the other clients.

It is then possible to conclude that the highly reliable and semi-automatically generated SSH-TLP client presented in this paper can in fact replace other manually developed implementations.

It is believable that the observations made for the SSH-TLP also hold for many other security protocols, where the cryptographic operations are very often so computationally intensive, that other operations can be neglected. In this case, implementation efficiency

depends almost entirely on the efficiency of the JCA provider, rather than on the efficiency of the protocol logic. Notably, the Spi2Java framework allows the developer to choose independently for each cryptographic operation the JCA provider to be used by the generated implementation.

4.5 Discussion

The case study illustrated in this chapter shows the viability of the proposed MDD approach, with a real and widely used protocol at hand. By following the proposed workflow, an abstract model of the SSH-TLP has been designed and verified for secrecy and server authentication. Then, the client-side model has been soundly refined, until a client application for the SSH-TLP has been semi-automatically derived.

The abstract model takes into account many protocol features, including algorithm negotiation, interaction with a key store, and error handling. Up to our knowledge, this is the first verified formal specification of SSH-TLP fully modeling all these features. The modeling strategies are generic, allowing different implementations to be derived from the same specification. For example, the key store interaction is generically modeled by a request/response paradigm, allowing applications to be backed by different key store implementations. It is believable that these modeling strategies can be reused in other protocol models too.

The size of the protocol model is significant enough that subtle errors, similar to bugs in applications, can sparsely appear. This issue is significant when dealing with safety properties such as secrecy and authentication, because an erroneous model can make a safety property trivially become true, while the real protocol is in fact flawed. In principle, a liveness property such as "correct protocol sessions always reach the end of the protocol" should be proven; however liveness properties cannot be proven in a Dolev-Yao environment, due to the attacker being able to drop any message. In this work, a preliminary approach to exploit safety properties to increase confidence on the model "liveness" is given, and it has been successfully used on the proposed SSH-TLP model.

The client-side application that has been semi-automatically refined from the model has been shown to be interoperable, by testing it against several third party server implementations. In order to achieve interoperability, an SSH-TLP specific encoding layer has been manually developed. This took a significant amount of time in the whole application development, and the manually written code is currently assumed, and not verified, to satisfy the sufficient information flow and invertibility conditions stated in section 3.2. As a future work, it is believable that automatic generation of the encoding layer, starting from a mathematical model of such encoding functions can both shorten development effort, and increase the overall assurance on the generated application, because some formal reasoning could be done over the mathematical models of the encoding functions.

The generated application has been shown reliable too, by correctly handling and rejecting over 660 maliciously crafted protocol sessions.

The measures gathered on the implementation code show that the generated application is modular. The protocol logic, the low-level cryptographic and network operations,

and the encoding layer are encapsulated in different packages, each containing respectively automatically generated code, library code, and manually developed code. Although the manually developed code accounts for almost half of the whole application code, it only deals with the encoding layer, while the most security-sensitive code is either automatically generated, or it is part of the SpiWrapper library, discussed in this work, and shipped as part of the Spi2Java framework.

Part II

Formally Verifiable Monitoring of Legacy Implementations

Chapter 5

Methodology

In this chapter, the focus is about assessing the correctness of legacy implementations, rather than on the development of correct new implementations. Indeed, it is often the case in practice that a legacy implementation is already in use which cannot be substituted by a new one: for example, when the legacy implementation is strictly coupled with the rest of the information system, making a switch very costly.

In this context, the proposed approach is based on black-box monitoring of legacy security protocols implementations. Using the Dolev-Yao [27] model, cryptographic functions are assumed to be correct, and the focus is about their usage within cryptographic protocols. Moreover, implementations of security protocol actors are the goal of the work, rather than the high level specifications of such security protocols. That is, it is assumed that a given protocol specification is secure (which can be proven using existing tools); instead, by monitoring it, the correctness of a given implementation of one protocol's role with respect to its specification is assessed, making it is resilient to Dolev-Yao attacks.

The overall methodology to design and develop monitors is plugged into the standard MDD architecture of Spi2Java. The way monitoring development plugs into that architecture is depicted in figure 5.1. Given the formal specification of one protocol agent, the "agent to monitor" ($a2m$) function is used to automatically generate a formal specification of the monitor for that protocol role. Then, by using the MDD methodology implied by the Spi2Java tools, an implementation for the monitor model is semi-automatically derived. The generated monitor application is finally run together with the monitored protocol role implementation (not shown in the picture). Note that this approach leverages MDD in order to overcome one of its main limitations: namely not being able to handle legacy implementations. Indeed, MDD is used to obtain a formally sound monitor implementation; then the monitor enforces security properties on the legacy monitored applications.

A monitor implementation differs from a fresh implementation of a security protocol, because it does not execute protocol sessions on behalf of its users. The monitor instead observes protocol sessions started by the legacy implementations, in order to recognize and stop incorrect sessions, in circumstances where the legacy implementations cannot be replaced.

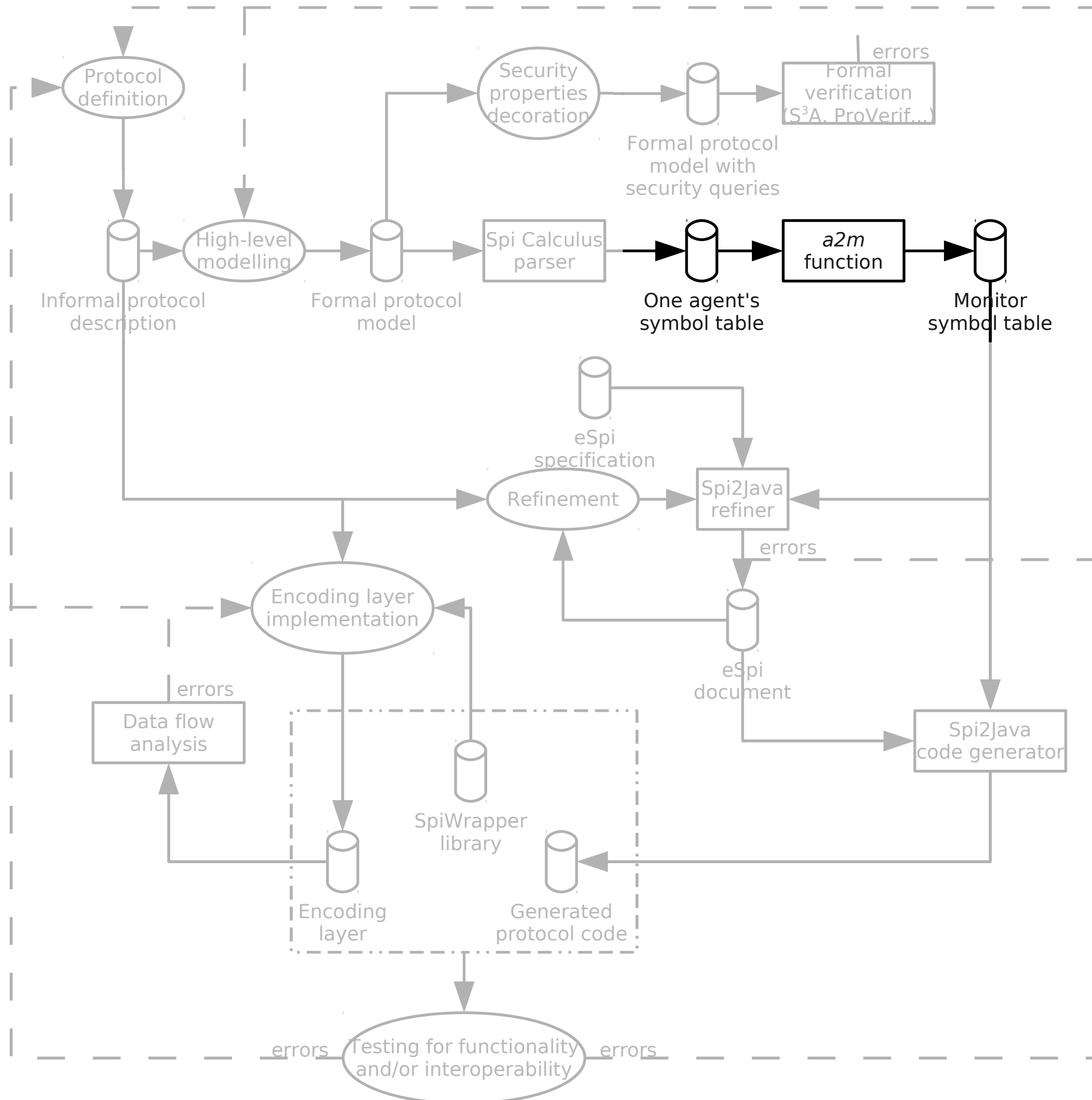

Figure 5.1: Monitor design and development methodology plugged into the Spi2Java architecture.

The monitoring is "black-box" in that the source code of the monitored application is not needed; only its observable behavior (data transmitted over the medium, or *traces*) and locally accessed data are required. Thus any legacy implementation can still be used in production as is, while being monitored.

The correctness of this approach depends on the correctness of the generated monitor. By plugging into the Spi2Java framework, all soundness results given in part I can be reused. The only new element that must be proven sound is the $a2m$ function. For the

$a2m$ function, soundness can be expressed as "the generated monitor stops all incorrect protocol sessions", and completeness as "only incorrect protocol sessions are stopped". It is believable that at least soundness can be proven by induction over the function definition, by noting that checks on exchanged data are inserted properly so as to avoid incorrect protocol sessions to be accomplished; although a formal definition of $a2m$ is given here, the soundness proof is left for future work.

For performance trade-offs, monitoring can be performed either "online" or "offline". In the first case, all messages are first checked by the monitor, and then forwarded to the intended recipient only if they are safe. In the second case, all messages exchanged by the monitored application are logged, and then fed to the monitor for later inspection. The online paradigm prevents a security property to be violated, because protocol executions are stopped as soon as an unexpected message is detected by the monitor, before it reaches the intended recipient. However, online monitoring may introduce some latency. The offline paradigm does not introduce any latency and is still useful to recognize compromised protocol sessions later, which can limit the damage of an attack. For example, if a credit card number is stolen due to an e-commerce protocol attack, and if offline monitoring is run overnight, one can discover the issue at most one day later, thus limiting the time span of the fraud.

In this work, the main goal of monitors is to detect, stop and report incorrect protocol runs. Monitors are not designed for example to assist one in forensic analysis after an attack has been found.

Note that this approach can be exploited during the testing phase as well: One can run an arbitrary number of simulated protocol sessions in a testing environment, and use the monitor to check for the correct behavior.

In order to validate the proposed approach, a monitor for the SSL protocol is presented in chapter 6. The generated monitor stops incorrect sessions that could, for example, exploit a recently found flaw in the OpenSSL implementation.

The rest of this chapter is organized as follows. Section 5.1 illustrates the used notation and the network model. Section 5.2 describes the function translating a spi calculus protocol agent's specification into a monitor specification for that agent. Then, section 5.3 describes related work.

An abridged version that includes the content of this and of the next chapter appeared in [53].

5.1 Notation and Network Model

Notation

By plugging into the spi calculus MDD architecture, the $a2m$ function transforms spi calculus models of protocol agents into spi calculus models of their corresponding monitors. The spi calculus language is amenable to the monitoring approach because the behavior of single security protocol agents is expressed by explicitly specifying checks on received data. Thus, from the spi calculus specifications of protocol agents, the $a2m$ function can derive precise and complete specifications of their monitors.

Note that for each spi calculus term it is possible to build an ordered term tree. For example, the tree for the term $\{(x,y)\}_k$ is depicted in figure 5.2. By overloading notation,

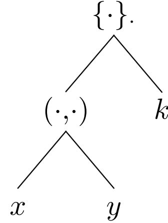

Figure 5.2: Tree of the $\{(x,y)\}_k$ term.

from now on M will represent both a term, and its root node. The $children(M)$ function returns the ordered list of children nodes (of depth 1) for the (root node of) term M. For example, $children(\{(x,y)\}_k)$ returns the list $[(x,y),k]$; if a is a name or variable, $children(a)$ returns [], which denotes the empty list.

Network Model

Many network models have been proposed in the Dolev-Yao setting. For example, sometimes the network is represented as a separate process [66], offering a service between protocol agents. The attacker is then connected to this network, and has the special abilities to eavesdrop messages, drop and modify them, or forge new ones. In other cases, the attacker is the medium [63], and honest agents can only communicate *through* the attacker. Even more detailed network models have been developed [72], where some nodes may have direct, private secured communication with other nodes, while still also being able to communicate through insecure channels, controlled by the attacker. Moreover, in all of these models, each node can in principle be a composition of concurrent processes, that synchronize by communicating over internal (private) channels. This feature allows to tune the granularity of the representation of each participant: two nodes sharing a private channel may be implemented in many different ways: they could be physically different machines, connected by a dedicated infrastructure; or they could be two processes on the same machine, using inter-process communication; or they could even be two functions of the same program, where the secure channel is implemented by function invocation.

In general, different network models and agents granularity justify different positions of the monitor with respect to the monitored agent, affecting the way the monitor is actually implemented. It turns out that there exists a trade-off between model granularity, monitor powerfulness and monitor complexity. The more fine grained is the model, the more data the monitor will be able to check (e.g. including input parameters on function invocations), but the more complex the monitor implementation will be.

As a motivating example, consider the case where the network is modeled as a graph, where vertexes are the network agents, and directed edges are half-duplex communication channels, either public or private, that respectively mean under the attacker control or not. In such a setting, as hinted before, the implementation of a private channel could

range from a physical channel, to a function invocation. In the former case, monitoring the secure channel may be feasible, and with minimal effort; in the latter case, monitoring that internal private channel implemented by function invocation may be hardly feasible. However, when monitoring a single agent, it is in general useful to monitor its private channels too, because some secret data may unwillingly flow to other agents through private channels, and then being put on to a public channel by the other agents, which are not monitored: monitoring private channels would avoid such errors. Nevertheless, whether a private channel is easily monitorable or not depends on the granularity of agents representation, and is implementation dependent.

A reasonable trade-off can be found by focusing on a scenario simple enough to let monitor implementations be not too complex; at the same time, this scenario is commonly found in practice, so that many security protocols can be faithfully modeled. In this scenario, that is depicted in figure 5.3, the attacker is the medium, and every protocol agent can only send data over a single insecure channel c, and private channels between different agents are not allowed. Moreover, agents are sequential and non-recursive. In practice,

Figure 5.3: Agents A and B with the attacker.

this model represents the common case where different agents running on different nodes use an untrusted network to exchange data (e.g. a client-server protocol, or a protocol involving trusted third parties); all inter-process communication within a node is kept in its internal state. Although it is believable that this scenario can simulate more complex ones, where multiple public channels are used, or where agents can be multi-threaded or open-ended (recursive), or where different agents share private channels, the formal development of this theory, and where to put the monitor in the enhanced scenarios, is left for future work.

Let A be defined as the (correct) model of the agent to be monitored, and M_A as the model of the monitor for agent A.

In order to effectively monitor a protocol role, M_A must be able to gather all messages sent and received by A, and also to access all local data accessed by A. For instance, if A is deciphering some data using its private key, the monitor should have access to that private key, in order to check the content of the plaintext too. It may be argued that giving M_A access to the private keys of A may increase the probability of information leak, or even compromise some non-repudiation properties of the protocol. Although in principle this is true, it must be pointed out that M_A is considered a trusted application (more trusted than the legacy application at least), and the use of formal methods for its generation can give high confidence about its correctness.

In order to give M_A access to all exchanged messages and to locally accessed data, it is reasonably assumed that M_A runs in the same environment as A: for example they run on the same operating system with same user privileges.

Following the given scenario, when the monitor is not present, A communicates directly with the attacker on channel c, and the attacker will then forward messages to the other parties, still being able to forge, drop or modify messages. When the monitor is present, A communicates with M_A only, through the use of a private channel c_{AM}, not known by the attacker, while M_A is directly connected to the attacker by channel c, as depicted in figure 5.4, where the dashed box denotes that A and M_A run in the same environment. Note that in A channel c is in fact renamed to c_{AM}. The assumption about c_{AM} not being

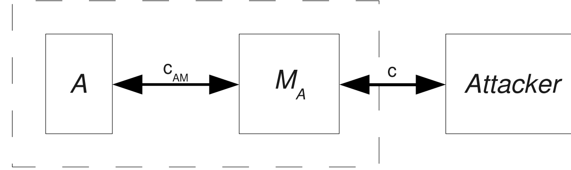

Figure 5.4: Agent A monitored by M_A and the attacker.

known by the attacker is reasonable, because we assume that M_A and A run in the same environment.

5.2 The Monitor Generation Function

The $a2m$ function is formally defined here: it translates a sequential protocol role specification into a monitor specification for that role. Before introducing the function definition, the concepts of *known* and *reconstructed* terms are given. For any spi calculus state, a term T is said to be *known* by the monitor through variable $_T$, iff $_T$ is bound to T. This can happen either because the implementation of M_A has access to the agent's memory location where T is stored; or because T can be read from a communication channel, and M_A stores T in variable $_T$. A compound term T (that is not a name or a variable) is said to be *reconstructed*, if all the terms in the $children(T)$ list are known or reconstructed.

Let us clarify these concepts by an example. Consider this made up specification for A:

$$(\nu n)\overline{c}\langle H(n)\rangle.\overline{c}\langle n\rangle.\mathbf{0}$$

where in the initial state, both n and $H(n)$ are assumed to be not known by M_A, for example because accessing the memory area where their values are stored by A would require too much effort. When $H(n)$ is sent over channel c, the monitor stores this term in the $_H(n)$ variable, so $H(n)$ becomes known through $_H(n)$, and M_A does not perform any check on it. However, when n is sent over c, the monitor stores it in $_n$, so n becomes known through $_n$, and $H(n)$ can be reconstructed as $H(_n)$. Now $_H(n)$ can be matched against the reconstructed value $H(_n)$, to ensure data coherence.

Note that the monitor implementation presented here does not enforce that nonces are actually different for each protocol run. To enable this, the monitor should track all previously used values, in order to ensure that no value is used twice. However, this check could be highly costly, both in time and resources, especially in the online mode. In

order to drop this check, it is needed to assume that the random value generator in the monitored agent is correctly implemented.

Formally, let $SpiTerm$ be the set of spi calculus terms, then $known : SpiTerm \rightarrow SpiTerm$ is a function mapping each known term to the variable where it is stored. In general, more than one variable could be bound to the same term, making *known* not a function. However, the monitor generator function is defined such that *known* is a function. Note that *known* is only defined on the known terms, that is $dom(known)$ only contains those terms for which there exist a variable binding.

The $reconstructed : SpiTerm \rightarrow SpiTerm$ function returns, when possible, a term that "reconstructs" the given term by using known children and, if they are not available, by using the reconstructed ones. The formal definition is trivial but quite verbose, while an example can better help in understanding its purpose: if $known(M) = _M$ and $known(H(M)) = _H(M)$, then $reconstructed((H(M),M))$ returns $(_H(M),_M)$; note that it is not the case that $reconstructed((H(M),M))$ returns $(H(_M),_M)$, because known children are preferred, when available.

The domain of *reconstructed* is formally defined as

$$dom(reconstructed) = \{T : SpiTerm \mid children(T) \neq \emptyset \\ \wedge\ \forall\, t \in children(T) \cdot t \in dom(known) \cup dom(reconstructed)\}$$

where the $\in$ symbol is overloaded to lists, meaning that the given term appears at least once in the list. The $children(T) \neq \emptyset$ predicate avoids that names or variables are in $dom(reconstructed)$, while the universally quantified predicate ensures that each child of the root node of term T is either known or reconstructed. Again, from the definition it follows that terms that cannot be reconstructed are not in $dom(reconstructed)$. Note that, as terms become known, the *reconstructed* function is updated too. In the example given above, if M was not known, then it was not possible to reconstruct $(H(M),M)$; however, if later M became known (for example, because it was sent over a channel), then $(H(M),M)$ would become reconstructed.

The $choose : SpiTerm \rightarrow SpiTerm$ function returns the known or, if it is not available, the reconstructed term for the given term, and is defined as

function $choose(T)$
  if $T \in dom(known)$ **then**
    return $known(T)$
  else if $T \in dom(reconstructed)$ **then**
    return $reconstructed(T)$

with $dom(choose) = dom(known) \cup dom(reconstructed)$.

Let Spi be the set of spi calculus processes, then $a2m : Spi \rightarrow Spi$ is the function that translates a spi calculus agent specification into the spi calculus specification of the corresponding monitor. Besides the aforecited functions, the *a2m* function also uses a list Q. If Q and Q' are two lists, then $Q|Q'$ denotes the concatenation of the two. In order to keep the notation simple, each function does not take Q, *known* and *reconstructed* as input parameters, instead they are updated by the functions side-effects. The *a2m*

function is formally defined in figure 5.5. For brevity, in this and the following function definitions, asymmetric encryption is not shown; it is handled like symmetric encryption, when the appropriate key is known, or like hashing, otherwise.

function $a2m(P)$
  if P is $[M\ is\ N]P'$ **then**
    if $M,N \in dom(choose)$ **then**
      return $[choose(M)\ is\ choose(N)]\ a2m(P')$
  else if P is $let\ (x,y) = M\ in\ P'$ **then**
    if $M \in dom(choose)$ **then**
      $known := known \cup \{x \to x, y \to y\}$
      return $let\ (x,y) = choose(M)\ in\ a2m(P')$
  else if P is **0** **then**
    return $emptyQ()$ **0**
  else if P is $case\ M\ of\ 0 : P'\ suc(x) : P''$ **then**
    if $M \in dom(choose)$ **then**
      $br1 := a2m(P')$
      $known := known \cup \{x \to x\}$
      $br2 := a2m(P'')$
      return $case\ choose(M)\ of\ 0 : br1\ suc(x) : br2$
  else if P is $case\ L\ of\ \{x\}_N\ in\ P'$ **then**
    if $L,N \in dom(choose)$ **then**
      $known := known \cup \{x \to x\}$
      return $case\ choose(L)\ of\ \{x\}_{choose(N)}\ in\ a2m(P')$
  else if P is $c(x).P'$ **then**
    $known := known \cup \{x \to x\}$
    $Q := Q|[x]$
    return $c(x).a2m(P')$
  else if P is $\overline{c}\langle M\rangle.P'$ **then**
    return $emptyQ()\ c_{AM}(_M)\ check(M,_M)\ \overline{c}\langle _M\rangle\ a2m(P')$
  else if P is $(\nu n)P'$ **then**
    return $a2m(P')$
  error Required data not monitored; monitor not sound.

Figure 5.5: From agent specification to monitor specification.

In general, the function operates based on the kind of process. For each process, if all the free terms are known or reconstructed, then the process is translated into the monitor process; else an error is thrown, because some terms cannot be monitored and the otherwise generated monitor would not be sound.

Some significant cases are now explained. If the process P to be translated is a match $[M\ is\ N]P'$ process, then the $[choose(M)\ is\ choose(N)]$ process is returned, and the function is recursively invoked on P'. Note that the *choose* function is used to resolve the

actual names of M and N, because they may be mapped to some known or reconstructed term.

If P is *let* $(x,y) = M$ *in* P', then it is translated into *let* $(x,y) = choose(M)$ *in* $a2m(P')$. Note that x and y are bound in $a2m(P')$, and *known* is updated accordingly, so that in later accesses to x or y, they will be resolved as $known(x) = x$ and $known(y) = y$.

In the base case where P is the **0** process, the content of Q is forwarded to A by the *emptyQ* function (more on this later), then the **0** process is returned.

If P is $c(x).P'$, then the monitor receives message x on behalf of the agent, buffers x in Q, and then behaves following what is done by the agent; x is bound by the input process, and the *known* function is updated accordingly. The received message stored in x is not forwarded to A immediately, because this could lead A to receive some malicious data, that could for example enable some denial of service attack. Instead, the received data are buffered, and will be forwarded to A only when necessary: that is when the process should end (**0** case), or when some output data from A are expected: A should receive all buffered data, because they may be needed to compute the value to be sent. Often P' will contain some checks on the received data, and they will be implemented by the monitor as well. Note that if two ore more inputs are made by A before an output or the process end, they will all be buffered, and only forwarded in the right order when necessary.

If P is $\overline{c}\langle M\rangle.P'$, then, as explained before, all buffered data are forwarded to A, so that it is able to compute M; then M is received by M_A on the private channel, and stored in the $_M$ variable. Before the monitor forwards $_M$ to the attacker, all possible checks on $_M$ are done, to check that it matches with M. These checks are introduced by the *check* function, explained in details below; what checks can be performed on the received data depends on what terms are known or reconstructed. Intuitively, M should become known through $_M$: this is fully handled by the *check* function.

Finally, any restriction process $(\nu n)P'$ is ignored, because the monitor shall never create fresh data, rather it should *know* what fresh data has been generated by A. If the monitor can read the value of n from the agent's state, then $known(n) = n$ since the initial state; otherwise, $n \notin dom(known)$ at the beginning, however n (or some term that depends on it, like $H(n)$) may become known later, if it is sent over a channel.

The auxiliary function *emptyQ* outputs the buffered variables over the private channel, so that buffered data are actually forwarded to the monitored agent. Its formal definition is

function *emptyQ*
    if Q is [] **then**
        return
    else
        read Q as $[x]|Q'$
        $Q := Q'$
        return $\overline{c_{AM}}\langle x\rangle.\ emptyQ()$

The *check* : $SpiTerm \times SpiTerm \rightarrow Spi$ takes the expected term E as first parameter, and the variable bound to the received term R as second parameter. The function performs all possible checks between them. In particular, if E is already known or reconstructed,

then it is checked whether the variable already bound to E matches R. In case E is not known or reconstructed, the *explode* function is invoked that, if possible, tries to de-construct R and check its subterm, otherwise, by calling the *backcheck* function, E is set to be known by R. Finally, as mapping a new known term enables new terms to be reconstructed, the *backcheck* function also adds all those checks that are made now possible because of the new reconstructed terms.

The formal definitions of the *check*, *explode* and *backcheck* functions are given in figure 5.6. Some relevant cases of the *explode* function are commented. Note that, when

function *check*(E,R)
  if $E \in dom(choose)$ **then**
    return $[R\ is\ choose(E)]$
  else
    explode(E,R)

function *explode*(E,R)
  if E is name or variable **then**
    return *backcheck*(E,R)
  else if E is (E',E'') **then**
    return $let\ (R',R'') = R\ in\ check(E',R')\ check(E'',R'')$
  else if E is $\{E'\}_K$ **then**
    if $K \in dom(choose)$ **then**
      return $case\ R\ of\ \{R'\}_{choose(K)}\ in\ check(E',R')$
    else
      return *backcheck*(E,R)
  else if E is $H(E')$ **then**
    return *backcheck*(E,R)

function *backcheck*(E,R)
  $formerlyKnown :=$
    $\{E' \in SpiTerm | E' \in dom(known)\ \wedge\ E' \notin dom(reconstructed)\}$
  $known := known \cup \{E \to R\}$ ▷ Note that *reconstructed* gets updated too
  for all $E' \in formerlyKnown \cap dom(reconstructed)$ **do**
    $res := res\ [known(E')\ is\ reconstructed(E')]$
  return res

Figure 5.6: Auxiliary functions: *check*, *explode* and *backcheck*.

the *explode* function is invoked, $E \notin dom(choose)$, and in particular $E \notin dom(known)$ is true; also note that the top level **if** switches are made on E, and not on R, because E is the "expected term" that has a known structure, while R is a variable that should have the same structure, which is the reason why R is de-constructed when possible.

If E is a name or a variable, then R cannot be de-constructed, so, by the *backcheck*

function, E is set to be known through R, and all newly enabled checks are added. For example, after E is known through R, $H(E)$ is reconstructed by $H(R)$; in case $H(E)$ was previously known, say through $_H(E)$, the *backcheck* function will ensure that the $[_H(E)\ is\ H(R)]$ process is added.

If E is the pair (E',E''), then it should be possible to de-construct R by using the pair splitting process; if the pair splitting fails, then R was not a pair, thus storing a wrong term. After R is split into R' and R'', its parts are recursively checked against their expected terms. Note that in this case E is *not* set as known through R. Indeed, if R can be split into R' and R'', then it is always possible to reconstruct it by using its parts; if E was set known through R, then the *backcheck* function could possibly introduce later the $[R\ is\ (R',R'')]$ check, which is redundant (making the monitor implementation inefficient).

If E is the hashing $H(E')$, then there is no way to de-construct it, so the *backcheck* function is invoked, like in the name or variable case.

Finally, if E is $\{E'\}_K$, then two cases are possible. If K is known or reconstructed, then it is possible to try to de-construct the encryption, getting the plaintext R', that is then checked against its expected term E'; like in the pair splitting case, E is not set to by known trough R. Else, when K is not known or reconstructed, it is not possible to de-construct the encryption, so, like in the hashing case, the *backcheck* function is invoked.

Finally, the definition of M_A is

$$M_A \triangleq preChecks()\ a2m(A)$$

with Q being the empty list [] in the initial state, and *preChecks* a consistency function that checks that all terms which are both known and reconstructed in the initial state are the same. Formally

function *preChecks*
  for all $T \in SpiTerm \,|\, T \in dom(known)\ \wedge\ T \in dom(reconstructed)$ **do**
    $res := res\ [known(T)\ is\ reconstructed(T)]$
  return *res*

Also note that if all free variables and free names in A are known, then the error state of $a2m$ is unlikely to be reached. The error state could still be reached for example if an unknown fresh value is checked against some data, immediately after its creation: $(\nu n)[n\ is\ M]$ or $(\nu n)[n\ is\ n]$; however these two cases are rather pointless because the first check would always fail, and the second one always pass (and can thus be removed), by definition.

In order to better understand how the $a2m$ function actually works, an example is now shown. Note that the focus of this example is on the way the $a2m$ function operates on a given agent, rather than on monitoring a security protocol. A full monitoring example dealing with the SSL protocol is given in chapter 6.

Figure 5.7(a) shows a specification A for an agent, and figure 5.7(b) shows its derived monitor specification M_A. At line 1a the agent A process is declared: it has two free variables, a message M and a symmetric key k. At line 2a A sends the encryption of M with key k over the communication channel (which is c_{AM} when being monitored, but would be c in the original specification). Then, at line 3a it receives a message that is

```
1a: A(M,k) :=
2a:   cAM<{M}k>.



3a:   cAM(x).
4a:   [x is H(M)]
5a:   0
```

(a) Agent A specification.

```
1m: MA(k,_H(M)) :=
2m:   cAM(_{M}k).
3m:   case _{M}k of {_M}k in
4m:   [_H(M) is H(_M)]
5m:   c<_{M}k>.
6m:   c(x).
7m:   [x is _H(M)]
8m:   cAM<x>.
9m:   0
```

(b) Monitor specification derived from agent A one.

Figure 5.7: Example specification of agent A along with its derived monitor M_A.

stored into variable x, and, at line 4a, the received message is checked to be equal to the hashing of M: if this is the case, the process correctly terminates.

At line 1m, the monitor M_A is declared: to make this example significant, it is assumed that in the initial state $known(k) = k$ and $known(H(M)) = _H(M)$, that is the monitor knows the key k used by A through the variable k, but does not know M (for example because those data cannot be accessed); instead the monitor has access to the location where $H(M)$ is stored, and this value is bound to the variable $_H(M)$ in the monitor. The *preChecks* function does not add any match process, since there is no term being both known and reconstructed in the initial state.

When line 2a is translated by $a2m$, lines 2m–5m are produced. By looking at the $a2m$ definition, first the *emptyQ* function is invoked, which returns an empty statement, since the list Q is empty in the initial state. Then the data sent by A are received by the monitor at line 2m, and stored in variable $_\{M\}_k$. Afterwards, the $check(\{M\}_k,_\{M\}_k)$ function is invoked. Since $\{M\}_k$ is not known (by hypothesis) or reconstructed (because M is not known or reconstructed), the $explode(\{M\}_k,_\{M\}_k)$ function is invoked, to possibly dissect the received value. Since $\{M\}_k$ is an encryption, and k is known, the decryption case process is output at line 3m, binding $_M$ to the value of the plaintext, that should be M. Now the $check(M,_M)$ function is invoked. Since M is not known or reconstructed, the $explode(M,_M)$ is called in turn. Since M is a name, $_M$ cannot be dissected; instead, the $backcheck(M,_M)$ is called, to let M be known through $_M$. Note that, before the $known(M) = _M$ relation is defined in *backcheck*, $H(M)$ is known through $_H(M)$, but it is not reconstructed, so $H(M) \in formerlyKnown$ holds. After the assignment, note that $reconstructed(H(M)) = H(_M)$, so $H(M)$ is a term that was previously only known through $_H(M)$, but that can be now be reconstructed by $H(_M)$. This idea is captured by $formerlyKnown \cap dom(reconstructed) = \{H(M)\}$, which leads to the match process output at line 4m, checking that the previously known term for $H(M)$ and the current reconstructed term are the same. Now, we are back in the $a2m$ function, that puts the output process in line 5m, forwarding received data to the attacker, if all checks passed.

Then, line 3a is translated into line 6m. Note that $known(x) = x$ will hold, and that Q is now $[x]$, because x has been received and buffered by the monitor. Line 4a is then

translated into line 7m, where $choose(x) = x$ and $choose(H(M)) = _H(M)$. Note that testing x against $_H(M)$ or $H(_M)$ would actually be the same, since they have already been checked to match. Finally, line 5a is translated into lines 8m and 9m. First, all buffered data (x in this case) are forwarded to A, then the monitor correctly ends.

5.3 Related Work

Several attempts have been made to check that a protocol role implementation is correct w.r.t. its specification which can be grouped in four main categories: (1) Model Driven Development (MDD); (2) Static Code Verification; (3) Refinement Types; (4) Online Monitoring and Intrusion Detection Systems (IDSs).

In this section, these approaches are discussed with respect to their capability of handling legacy implementations of security protocols, when their source code is not available.

As discussed above, the plain MDD approach, as presented for example in part I or in the related work presented in section 2.5, can only be used to generate new provably correct implementations of security protocols, however, it has the drawback of not handling legacy implementations of such security protocols.

The second approach starts from the source code of an existing implementation, and extracts a formal model which is verified for the desired security properties [41, 20]. In principle, this approach can deal with legacy implementations, but their source code must be available, and sometimes manually annotated too, which may not always be the case for legacy systems.

The third approach proves security properties of an implementation by means of a special kind of type checking on its source code [17]. Working on the source code, it shares the same advantages and drawbacks of the second approach.

The fourth approach comes in two versions. With online monitoring, the source code of an existing implementation is instrumented with assertions: program execution is stopped if any assertion fails at run-time [15]. Besides requiring the source code, the legacy implementation must be substituted with the instrumented one, which may not always be the case. IDSs are systems that monitor network traffic and compare it against known attack patterns, or expected average network traffic. By working on averages, in order not to miss real attacks, IDSs often report false positive warnings. In order to reduce them, sometimes the source code of the monitored implementation is also instrumented [39], sharing the same advantages and drawback of online monitoring.

In principle, a fifth approach that could handle legacy implementations may also be possible, that is formal verification of assembly code. Unfortunately, this approach is not viable in practice, because it is too difficult to abstract and identify the security relevant details from the flattened assembly code, making verification unfeasible.

Another branch of research focused on security wrappers for legacy implementations. In [2], a formal approach that uses security wrappers as firewalls with an encrypting tunnel is described. Any communication that crosses some security boundary is automatically and transparently encrypted by the wrappers. That is, the wrappers *add* security to a distributed legacy system. In our approach, the monitor *enforces* the security already

present in the system. Technically, our approach derives a monitor based on the security requirements already present in the legacy system, instead of adding a boilerplate layer of security.

Analogously, in [30] wrappers are used, among other things, to transparently encrypt local files accessed by library calls. However, distributed environments are not taken into account. Finally, in [68] wrappers are used to harden software libraries. However, cryptography and distributed systems are not considered, and the approach is test-driven, rather than formally based.

Chapter 6

An SSL Server Monitor Example

This chapter presents a case study about monitoring legacy SSL server implementations, by using the methodology discussed in chapter 5. A possible formal spi calculus model of an SSL server is introduced here. This model is then translated by the $a2m$ function into the formal model of the corresponding monitor. The SSL server model is generic, in that it is not tied to any particular implementation; thus, the obtained monitor model is generic too. This means that the monitor implementation generated from its formal model will be generic enough to be able to monitor different SSL server implementations without modifications or customizations. In general, the only implementation-dependent part in a monitor implementation is the code reading *known* data from the monitored application memory, if any. This can be easily handled by plugins, where each plugin is responsible for gathering known data from a specific application.

Section 6.1 presents the proposed spi calculus model for an SSL server, and its derived monitor model, by application of the $a2m$ function. Then, section 6.2 briefly describes how a monitor implementation was derived from its model, mainly focusing on interesting steps that had to be performed within the standard Spi2Java MDD workflow. Finally, section 6.3 discusses some experimental results. They include some considerations about performances of the online monitoring approach against several different legacy SSL server implementations. Moreover, it is shown how the monitor actively avoids protocol violations when coupled with a flawed OpenSSL implementation.

6.1 Monitor Specification

As shown in figure 5.1, in order to get the monitor specification, a spi calculus specification of the server role for the SSL protocol is needed. The full SSL protocol is rather complex: many scenarios are possible, and different sets of cryptographic algorithms (called *ciphersuites*) can be negotiated. For simplicity, this example considers only one scenario of the SSL protocol version 3.0, where the same cipher suite is always negotiated. Despite these simplifications, it is believable that the considered SSL fragment is still significant, and that adding full SSL support would increase the example complexity more than its significance.

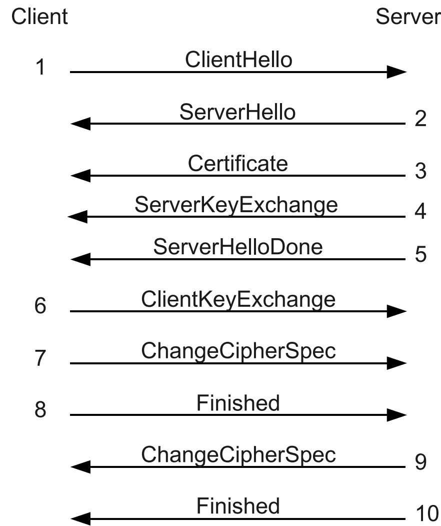

Figure 6.1: Typical SSL scenario.

The chosen scenario requires the server to use a DSA certificate for authentication and data signature. Although RSA certificates are more common in SSL, using a DSA certificate allowed us to stress a bug in the OpenSSL implementation, showing that the monitor can actively drop malicious sessions that would be otherwise accepted as genuine by the flawed OpenSSL implementation. The more common RSA scenario has been handled too, through a dedicated case-study that can be found in [54].

The SSL Server Model

The SSL scenario considered in this example is depicted informally in figure 6.1, while figure 6.2 shows a possible spi calculus specification of a server for the chosen scenario. Here, the Diffie-Hellman (DH) key exchange is modeled by the $EXP(L,M,N)$ term, which expresses the modular exponentiation $L^M \bmod N$, along with the equation

$$EXP(EXP(g,a,p),b,p) = EXP(EXP(g,b,p),a,p)$$

To avoid ambiguities, an explicit `rename n = M in P` process is introduced too, that simply renames the term M to n and then behaves like P.

The first message is the ClientHello, sent from the client to the server. It contains the highest protocol version supported by the client, a random value, a session ID for session resuming, and the set of supported cipher suites and compression methods. In the server specification, the ClientHello message is received and split into its parts at line 2S. In the chosen scenario, the client should send at least 3.0 as the highest supported protocol version, and it should send 0 as session ID, so that no session resuming will be performed. Moreover, the client should at least support the always-negotiated cipher suite, namely `SSL_DHE_DSS_3DES_EDE_CBC_SHA`, with no compression enabled. All these constraints are checked at lines 3S–4S.

```
 1S Server() :=
 2S   c(c_hello).  let (c_version,c_rand,c_SID,c_ciph_suite,c_comp_method) = c_hello in
 3S   [ c_version is THREE_DOT_ZERO ]  [ c_SID is ZERO ]
 4S   [ c_ciph_suite is SSL_DHE_DSS_3DES_EDE_CBC_SHA ]  [ c_comp_method is comp_NULL ]
 5S   (@s_rand)  (@SID)
 6S   rename S_HELLO = (THREE_DOT_ZERO,s_rand,SID,SSL_DHE_DSS_3DES_EDE_CBC_SHA,comp_NULL) in
 7S   (@DH_s_pri)  rename DH_s_pub = EXP(DH_Q,DH_s_pri,DH_P) in
 8S   rename S_KEX = ((DH_P,DH_Q,DH_s_pub),[{H(c_rand,s_rand,(DH_P,DH_Q,DH_s_pub))}]s_PriKey) in
 9S   c<S_HELLO,S_CERT,S_KEX,S_HELLO_DONE>.
10S   c(c_kex).  let (c_kexHead,DH_c_pub) = c_kex in  rename PMS = EXP(DH_c_pub,DH_s_pri,DH_P) in
11S   rename MS = H(PMS,c_rand,s_rand) in  rename KM = H(MS,c_rand,s_rand) in
12S   rename c_w_IV = H(KM,C_WRITE_IV) in  rename s_w_IV = H(KM,S_WRITE_IV) in
13S   c(c_ChgCipherSpec).  [ c_ChgCipherSpec is CHG_CIPH_SPEC ]
14S   c(c_encrypted_Finish).  case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
15S   let (c_Finish,c_MAC) = c_Finish_and_MAC in  [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
16S   let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
17S   [ final_Hash_MD5 is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,MD5) ]
18S   [ final_Hash_SHA is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,SHA) ]
19S   c<CHG_CIPH_SPEC>.
20S   rename DATA = (c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex,c_Finish) in
21S   rename S_FINISH = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
22S   (@pad)  c<{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH),pad}(KM,S_WRITE_KEY)~>.
23S   0
```

Figure 6.2: A possible spi calculus specification of an SSL server.

In the second message the server replies by sending its ServerHello message, that contains the chosen protocol version, a random value, a fresh session ID and the chosen cipher suite and compression method. The server random value and the fresh session ID are generated at line 5S, then the ServerHello message is declared at line 6S. Again, in the chosen scenario, the server chooses protocol version 3.0, and always selects the `SSL_DHE_DSS_3DES_EDE_CBC_SHA` cipher suite, with no compression enabled.

Then the server sends the Certificate message to the client: in the chosen scenario, this message contains a DSA certificate chain for the server's public key, that authenticates the server.

In the fourth message, named ServerKeyExchange, the server sends the DH key exchange parameters to the client, and digitally signs them with its public key. In the server specification, the DH server secret value `DH_s_pri` and the corresponding public value `DH_s_pub` are computed at line 7S. Then, at line 8S, the ServerKeyExchange message is declared: it consists of the server DH parameters, along with a digital signature of the DH parameters and the client and server random values, in order to ensure signature freshness.

The fifth message is the ServerHelloDone. It contains no data, but signals the client that the server ended its negotiation part, so the client can move to the next protocol stage. In the server specification, these four messages are sent all at once at line 9S.

In the sixth message, the client replies with the ClientKeyExchange message, that contains the client's DH public value. Note that there is no digital signature in this message, since the client is not meant to be authenticated to the server. In the server specification the ClientKeyExchange is received at line 10S, where the payload is split from the message header too. Both client and server derive a shared secret from the DH

key exchange. This shared secret is called Premaster Secret (PMS), and it is used by both parties to derive some shared secrets used for symmetric encryption of application data. The PMS is computed by the server at line 10S. By applying an SSL custom hashing function to the PMS and the client and server random data, both client and server can compute the same Master Secret (MS). The bytes of the MS are then extended (again by using a custom SSL hashing algorithm) to as many byte as required by the negotiated ciphersuite, obtaining the Key Material (KM) (line 11S). Finally, different subsets of bytes of the KM are used as shared secrets and as initialization vectors (IVs). Note that IVs, that are extracted at line 12S, are never referenced in the specification. They will be used as cryptographic parameters for subsequent encryptions, during the code generation step, which is detailed in section 6.2.

The seventh message is the ChangeCipherSpec, received and checked at line 13S. This message contains no data, but signals the server that the client will start using the negotiated cipher suite from the next message on.

The client then sends its Finished message. Message Authentication Code (MAC) and encryption are applied to the Finished message sent by the client, as the client already sent its ChangeCipherSpec message. The client Finished message is received and decrypted at line 14S. The decryption key used `(KM,C_WRITE_KEY)~` is obtained by creating a shared key, starting from the key material `KM` and a marker `C_WRITE_KEY` that indicates which portion of the KM to use. At line 15S the MAC is extracted from the plaintext, and verified. The unencrypted content of the Finished message contains the final hash, that hashes all relevant session information: all exchanged messages (excluding the ChangeCipherSpec ones) and the MS are included in the final hash, plus some constant data identifying the protocol role (client or server) that sent the final hash. In fact, the Finished message includes two versions of the same final hash, one using the MD5 algorithm, and one using the SHA-1 algorithm. Both versions of the final hash are extracted and checked at lines 16S–18S. As spi calculus does not support different algorithms for the same hash, they are distinguished by a marker (`MD5` and `SHA` respectively) as the last hash argument, making them syntactically different.

Then the server sends its ChangeCipherSpec message to client (line 19S), and its Finished message, that comes with MAC and encryption too (lines 20S–22S). Encryption requires random padding to align the plaintext length to the cipher block size. This random padding must be explicitly represented in the server specification, so that the monitor can recognize and discard it, and only check the real plaintext. Otherwise the monitor would try to locally reconstruct the encryption, but it would always fail, because it could not guess the padding. The protocol handshake is now complete, and next messages will contain secured application data.

In order to verify any security property on this model, the full SSL specification, including the client and all other auxiliary processes is required. However, this is outside the scope of this work; SSL security properties have already been verified, for example, by the AVISPA project [71]. Here it is assumed instead that the specification of the server is correct, and thus secure, so that the monitoring approach can be shown.

```
 1M Monitor_S(DH_s_pri) :=
 2M   c(c_hello).
 3M   let (c_version, c_rand, c_SID, c_ciph_suite, c_comp_method) = c_hello in
 4M   [ c_version is THREE_DOT_ZERO ]  [ c_SID is ZERO ]
 5M   [ c_ciph_suite is SSL_DHE_DSS_3DES_EDE_CBC_SHA ]  [ c_comp_method is comp_NULL ]
 6M   /* (@DH_s_pri) we assume to know the generated DH_s_pri for this session */
 7M   c_int<c_hello>.  c_int(s_hello_s_cert_s_kex_s_hello_done).
 8M   let (s_hello,s_cert,s_kex,s_hello_done) = s_hello_s_cert_s_kex_s_hello_done in
 9M   /* checks for s_hello */
10M   let (x_THREE_DOT_ZERO, s_rand, SID, x_SSL_DHE_DSS_3DES_EDE_CBC_SHA, x_comp_NULL)
11M     = s_hello in
12M   [ x_THREE_DOT_ZERO is THREE_DOT_ZERO ]
13M   [ x_SSL_DHE_DSS_3DES_EDE_CBC_SHA is SSL_DHE_DSS_3DES_EDE_CBC_SHA ]
14M   [ x_comp_NULL is comp_NULL ]
15M   /* checks for s_cert */
16M   [ s_cert is S_CERT]
17M   /* checks for s_kex */
18M   let (DH_s_values,DH_DSS_Signature) = s_kex in
19M   [ DH_s_values is (DH_P,DH_Q,EXP(DH_Q,DH_s_pri,DH_P)) ]
20M   check DH_DSS_Signature of H(c_rand,s_rand,DH_s_values) with S_CERT in
21M   /* checks for s_hello_done */
22M   [ s_hello_done is S_HELLO_DONE ]
23M   c<s_hello_s_cert_s_kex_s_hello_done>.
24M   c(c_kex).
25M   let (c_kexHead,DH_c_pub) = c_kex in
26M   rename PMS = EXP(DH_c_pub,DH_s_pri,DH_P) in  rename MS = H(PMS,c_rand,s_rand) in
27M   rename KM = H(MS,c_rand,s_rand) in
28M   rename c_w_IV = H(KM,C_WRITE_IV) in  rename s_w_IV = H(KM,S_WRITE_IV) in
29M   c(c_ChgCipherSpec).  [ c_ChgCipherSpec is CHG_CIPHER_SPEC ]
30M   c(c_encrypted_Finish).
31M   case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
32M   let (c_Finish,c_MAC) = c_Finish_and_MAC in  [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
33M   let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
34M   [ final_Hash_MD5 is
35M     H((c_hello,s_hello_s_cert_s_kex_s_hello_done,c_kex),C_ROLE,MS,MD5) ]
36M   [ final_Hash_SHA is
37M     H((c_hello,s_hello_s_cert_s_kex_s_hello_done,c_kex),C_ROLE,MS,SHA) ]
38M   c_int<c_kex>.  c_int<c_ChgCipherSpec>.  c_int<c_encrypted_Finish>.
39M   c_int(s_ChgCipherSpec).  [ s_ChgCipherSpec is CHG_CIPHER_SPEC ]
40M   c<s_ChgCipherSpec>.
41M   c_int(s_encrypted_Finish).
42M   case s_encrypted_Finish of {s_Finish_and_MAC_and_pad}(KM,S_WRITE_KEY)~ in
43M   let (s_Finish_and_MAC, pad) = s_Finish_and_MAC_and_pad in
44M   rename DATA = (c_hello,s_hello_s_cert_s_kex_s_hello_done,c_kex,c_Finish) in
45M   rename local_s_finish = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
46M   rename local_s_MAC = H((KM,S_MAC_SEC)~,local_s_finish) in
47M   [ s_Finish_and_MAC is (local_s_finish,local_s_MAC) ]
48M   c<s_encrypted_Finish>.
49M   0
```

Figure 6.3: The *a2m* generated spi calculus monitor model for an SSL server.

The Generated Monitor Model

The *a2m* function described in section 5.2 is applied on the server model. The output of the function is the online monitor model for the server role, which is shown in figure 6.3.

It is assumed that the monitor has access to the server private DH value, which is then *known*, while it is not able to read the freshly generated server random value `s_rand`, the

session ID SID and the random padding which are then not known nor reconstructed at generation time. Often, the server will generate a fresh DH private value for each session, and it will usually only store it in memory. In general, some effort will be required for the monitor to be able to directly read this secret from the legacy application memory, without the need of the source code. Alternatively, this effort may be mitigated by feeding both the legacy application and the monitor with the same source of randomness, in fact making the monitor able to "guess" the server private DH values. Nevertheless, in a testing environment, if the source code of the monitored application happens to be available, it is possible to patch the monitored application, so that it explicitly communicates the DH private value to the monitor. Indeed, this is reasonable because the monitor is part of the trusted system, and is actually more trusted than the monitored application.

Note that each free variable in the server specification is constant data, so it is reasonable to assume that they are all known to the monitor. That is, for each free variable x in the server specification, $known(x) = x$ holds. In order to make the automatically generated monitor specification more readable, some rename processes have been added, and bound variables have been given significant names.

At line 2M the monitor receives on the public channel `c` the ClientHello message `c_hello` on behalf of the server. By looking at the $a2m$ function definition, after the input process is translated, $known(c_hello) = c_hello$ holds, and c_hello is added to the queue of buffered messages. Then, from line 3M to 5M the translation continues replicating in the monitor all the checks that are performed in the server. All the restriction and rename processes from line 5S to line 8S in the server specification are dropped; for convenience at line 6M a comment remarks that the server private DH value is assumed to be known.

Then, the output process at line 9S is translated: first all buffered messages (`c_hello` in this case) are relayed to the server over the private channel `c_int` (line 7M), then the ServerHello, Certificate, ServerKeyExchange and ServerHelloDone messages are received from the server, and stored in the `s_hello_s_cert_s_kex_s_hello_done` variable (line 7M). Afterwards, the latter variable is checked against the term to which it should be bound, namely `(S_HELLO,S_CERT,S_KEX,S_HELLO_DONE)`. Since the `s_rand` and `SID` names composing the `S_HELLO` term are not known in the monitor, it is not possible to reconstruct the latter term, making it impossible to reconstruct the whole `(S_HELLO,S_CERT,S_KEX,S_HELLO_DONE)` term. For this reason, the `s_hello_s_cert_s_kex_s_hello_done` variable is exploded.

Following the *explode* definition, at line 8M the variable is split into its parts, that should match the ServerHello, Certificate, ServerKeyExchange and ServerHelloDone messages, then the *check* function is invoked on each of the split variables. The `s_hello` variable is checked from line 9M to 14M. Note that variables that should be bound to constant terms are matched against them, while `s_rand` and `SID` are just bound and not checked, because they become known through the fresh names generated by the server. As `s_rand` and `SID` become known, the `S_HELLO` term becomes reconstructed, but it would be now useless to check the reconstructed term against its expected term, because the match would certainly pass. Indeed, the *backcheck* function is properly defined, so as to avoid the introduction of this redundant check.

The `s_cert`, `s_kex` and `s_hello_done` variables are then checked from line 15M to 22M.

Note that the `EXP(DH_Q,DH_s_pri,DH_P)` term at line 19M can be built, because it is assumed that `DH_s_pri` is known; otherwise, it would have been impossible to perform that decryption, and thus to generate a sound monitor specification. The signature check at line 20M uses a special signature check process, instead of the standard spi calculus one, in order to cope with some specific features of the DSA signatures. Briefly, signing twice the same message with the same DSA certificate produces two semantically equivalent but byte-wise different signatures. For this reason, the standard spi calculus signature check process is not suitable to handle DSA signatures (while it can fully handle other signatures schemes, like for example RSA). So the *check* M *of* N *with* L *in* P process is used: this process behaves like P if M is a valid signature of N made with certificate L; otherwise it is stuck. It is believable that the results proven for the MDD methodology in part I and for the $a2m$ function here also hold when this special signature check process is added. The Spi2Java tools have been specifically patched for this case study in order to support the special signature check process. At line 23M the received `s_hello_s_cert_s_kex_s_hello_done` message is finally forwarded over the public channel.

The $a2m$ function continues translating line 10S, so at line 24M the ClientKeyExchange message is received and buffered by the monitor, and translation continues. Indeed, from line 24M to 37M, all checks made by the server from line 10S to 18S are plainly replicated into the monitor specification, and all input operations are replicated and their content buffered in the queue. When translation reaches line 19S, first the buffered messages are delivered in order to the server (line 38M), then the server ChangeCipherSpec message is received, checked and relayed on the public channel by the monitor (lines 39M–40M).

The server Finished message is received, checked, and relayed on the public channel at lines 41M–48M. It is essential that the random padding used to align the plaintext length to the block cipher size for encryption is explicitly represented in the server specification. Indeed, if the padding was not represented, the server Finished message would have had the form `{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH)}(KM,S_WRITE_KEY)~`, and it would have been in *dom*(*reconstructed*). For this reason, after line 41M the monitor would have matched the `s_encrypted_Finish` variable against the reconstructed term. Although this would be correct at the specification level, the monitor implementation could never reconstruct the same encrypted Finished message, because it could not guess the server chosen random padding for that encryption. That is, the monitor is able to reconstruct the encrypted Finished message, modulo the server chosen random padding. By letting the padding be explicitly represented, the `{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH),pad}(KM,S_WRITE_KEY)~` term becomes not reconstructed (because `pad` is not known nor reconstructed), so the encrypted Finished message is dissected, and all of its parts, except for the padding which is "discarded" (technically, it becomes known), are checked. One may argue that the same reasoning should then be applied to the client Finished message too. Although this could be possible, it is not needed. The server specification already prescribes to check the content of the client Finished message (and not, for example, to check it against a reconstructed term), and the monitor replicates the server checks; padding will be handled (and discarded) by the cryptographic function implementation.

After the server Finished message is relayed over the public channel at line 48M, line 23S is translated. Since there are no buffered messages in the queue, the monitor

terminates at line 49M.

6.2 Monitor Implementation

The source code of the monitor implementation can be found in [54]. In order to generate the monitor implementation, the Spi2Java MDD framework presented in part I is used.

In the SSL monitor case study, a two-tier marshaling layer has been implemented. Tier 1 handles the Record Layer protocol of SSL, while tier 2 handles the upper layer protocols. When receiving a message from another agent, tier 1 parses one Record Layer message from the input stream, and its contained upper layer protocol messages are made available to tier 2. The latter implements the real marshaling functions, for example converting US-ASCII strings to and from Java String objects. Analogous reasoning applies when sending a message. The marshaling layer functions only check that the packet format is correct. No control on the payload is needed: it will be checked by the automatically generated protocol logic.

The SSL protocol defines custom hashing algorithms, for instance to compute the MS from the PMS, or to compute the MAC value. These custom algorithms are not provided by any of the standard JCA providers. To overcome this limitation, the Spi2Java type systems modularity has been exploited. For each custom SSL hashing algorithm, a custom sub-type of the *Hashing* type has been defined in the eSpi type hierarchy. Then, each of the added types has been implemented in a Java class, according to the SSL specification and fitting in the SpiWrapper library.

Finally, it is worth pointing out some details about the IVs used by cryptographic operations (declared in the server specification at line 12S). For each term of the form $\{M\}_K$, the eSpi document allows its cryptographic algorithm (such as DES, 3DES, AES) and its IV to be specified. However, the IV is only known at run-time. The Spi2Java framework allows cryptographic algorithms and parameters to be resolved either at compile-time or at run-time. If the parameter is to be resolved at compile-time, the value of the parameter must be provided (e.g. AES for the symmetric encryption algorithm, or a constant value for the IV). If the parameter is to be resolved at run-time, the identifier of another term of the spi calculus specification must be provided: the parameter value will be obtained by the content of the referred term, during execution. In the SSL case study, this feature is used for the IVs. For example, the `{c_Finish_and_MAC}(KM,C_WRITE_KEY)~` term uses the `H(KM,C_WRITE_IV)` term as IV. Technically, this feature enables support for cipher suite negotiation. However, as stated above, this would increase the specification complexity more than it would increase its significance, and is left for future work.

6.3 Experimental Results

The monitor has been coupled in turn with three different SSL server implementations, namely OpenSSL[1] version 0.9.8j, GnuTLS[2] version 2.4.2 and JESSIE[3] version 1.0.1.

Since the online monitoring paradigm is used in this case study, the monitor is accepting connections on the standard SSL port (443), while the real server is started on another port (4433). Each time a client connects to the monitor, the latter opens a connection to the real server, starting data checking and forwarding, as explained above.

It is worth noting that switching the server implementation is straightforward. In the testing scenario, assuming that the server communicates its private DH value to the monitor, it is enough to shut down the running server implementation, and to start the other one; the monitor implementation remains the same, and no action on the monitor is required. Otherwise, it is enough to restart the monitor too, enabling the correct plugin that gathers the private DH value from the legacy application memory. In other words, in a production scenario, the same monitor implementation can handle several different legacy server implementations; in the monitor, the only server-dependent part is the plugin that reads the DH secret value from the server application memory.

In order to generate protocol sessions, three SSL clients have been used with each server; namely the OpenSSL, GnuTLS, and JESSIE clients. During experiments, the monitor helped in spotting a bug in the JESSIE client: This client always sends packet of the SSL 3.1 version (better known as TLS 1.0), regardless of the negotiated version, that is SSL 3.0 in our scenario. The monitor correctly rejected all JESSIE client sessions, reporting the wrong protocol version.

When the OpenSSL or GnuTLS clients are used, the monitor correctly operates with all the three servers. In particular, safe sessions are successfully handled; conversely, when exchanged data are manually corrupted, they are recognized by the monitor and the session is aborted: corrupted data are never forwarded to the intended recipient.

In order to estimate the impact on performances of the online monitoring approach, execution times of correctly ended protocol sessions with and without the monitor have been measured. Thus, performances regarding the JESSIE client are not reported, as no correct session could be completed, due to the discovered bug. That is, the measured performances all correspond to valid executions of the protocol only. Communication between client, server and monitor happened over local sockets, so that no random network delays could be introduced; moreover system load was constant during test execution. Table 6.1 shows the average execution times for different client-server pairs, with and without monitor enabled. For each client-server pair, the average execution times have been computed over ten protocol runs. Columns "No Monitor" and "Monitor" report the average execution times, in seconds, without and with monitoring enabled respectively. When monitoring is not enabled, the clients directly connect to the server on port 4433. The "Overhead" columns show the overhead introduced by the monitor, in seconds and in

[1] Available at: http://www.openssl.org/
[2] Available at: http://www.gnu.org/software/gnutls/
[3] Available at: http://www.nongnu.org/jessie/

Client	Server	No Monitor [s]	Monitor [s]	Overhead [s]	Overhead [%]
OpenSSL	OpenSSL	0.032	0.113	0.081	253.125
GnuTLS	OpenSSL	0.108	0.132	0.024	22.253
OpenSSL	GnuTLS	0.073	0.128	0.056	76.552
GnuTLS	GnuTLS	0.109	0.120	0.011	10.313
OpenSSL	JESSIE	0.158	0.172	0.014	8.986
GnuTLS	JESSIE	0.144	0.148	0.004	2.788

Table 6.1: Average execution times for protocol runs with and without monitoring.

percentage respectively. In four cases out of six, the monitor overhead is under 25 milliseconds. From a practical point of view, a client communicating through a real distributed network could hardly tell whether a monitor is present or not, since network times are at least one order of magnitude higher. On the other hand, in the worst cases online monitoring can slow down the server machine up to 2.5 times. Whether this overhead is acceptable on the server side depends on the number of sessions per seconds that must be handled. If the overhead is not acceptable, the offline monitoring paradigm can still be used.

6.3.1 The OpenSSL Security Flaw

The client side of the OpenSSL library prior to version 0.9.8j has been discovered flawed, such that in principle it could treat a malformed DSA certificate as a good one rather than as an error.[4] By inspecting the flawed code, such a malformed certificate, which exploits the affected versions, was forged. This malformed DSA certificate must have the q parameter one byte longer than expected. Up to our knowledge, this is the first documented and repeatable exploit for this flaw.

Without monitoring enabled, protocol sessions between an SSL server sending the offending certificate, and both OpenSSL clients version 0.9.8i (flawed) and 0.9.8j (fixed) are generated. By using the `-state` command line argument, it is possible to conclude that the 0.9.8i version completes the handshake by reaching the “read finished A” state (after message 10 in figure 6.1); while the 0.9.8j version correctly reports an “handshake failure” error at state “read server certificate A”, that is immediately after message 3 in figure 6.1.

When monitoring is enabled, the malformed server certificate is passed to the monitor as an input parameter, that is, the server certificate is *known* by the monitor. In this case the monitor actually refuses to start. Indeed, when loading the server certificate, the monitor spots that it is malformed, and does not allow any session to be even started. If we drop the assumption that the monitor knows the server certificate, then the monitor starts, and checks the server certificate when it is received over the network. During these checks, the malformed certificate is found, and the session is dropped, before the server Certificate message is forwarded to the client. This prevents the aforecited flaw to be

[4] http://www.openssl.org/news/secadv_20090107.txt

exploited on OpenSSL version 0.9.8i.

Conclusion

The main contribution of this work is to provide a formal soundness relation between verified abstract models of security protocols and their implementations. In order to get to this result, the methodology and tools have been provided, and specific issues related to formally sound refinement of security protocol models into executable implementations have been addressed.

The proposed workflow is based on the MDD paradigm. The application developer is first requested to develop the protocol model, then to progressively fill implementation details. This workflow offers modularity in application design and development, because first, when developing the abstract model, only the protocol logic is taken into account; then lower-level implementation details, or aspects, are addressed one at a time.

All the MDD workflow is assisted by a set of automated tools, called together the Spi2Java framework. These tools enable the application to be rapidly prototyped since the early beginning, when only the abstract model is available, by providing default values and implementations for the low-level details that must be manually refined.

The application derived from the abstract model includes many low-level features that are abstracted away in the formal model, whereas they are essential to make the generated application interoperable. For example, for each cryptographic operation it is possible to choose the corresponding cryptographic algorithms and parameters. Moreover, such algorithms and parameters can be resolved either statically at compile-time, or dynamically at run-time. The latter option in fact enables support for per-session algorithm negotiation, as shown in the SSH-TLP case study.

Another interoperability-enabling factor is the capability of the generated application to marshal exchanged data, and to encode data to be encrypted or hashed, according to some protocol description. The proposed MDD methodology lets the developer implement such marshaling and encoding functions, thus allowing for the maximum flexibility. However, letting the developer manually write some code that is plugged into the generated application may rise some concern about the whole application soundness, when this manually written code is executed. Regarding the marshaling functions, in this work it has been formally shown that, if some information flow conditions hold on the manually written code, then no security flaws can be introduced by such marshaling functions. These information flow conditions can be checked in isolation on the sequential code implementing the functions. This kind of sequential verification is simpler than including a model of such marshaling functions during verification of the distributed protocol. Regarding functions encoding data to be encrypted or hashed, sufficient conditions are stricter, requiring

also encoding/decoding function implementations to be proven invertible. One way to ensure this property could be again by using an MDD approach: the developer starts from a mathematical description of the encoding function, on which the invertibility property can be proven, and then a sound implementation is automatically generated. Although feasible in principle, the automatic implementation of the marshaling and encoding functions is left for future work.

While expressing the sufficient conditions for the marshaling and encoding functions to be safe, some results about fault-preserving renaming transformations for security protocols have been enhanced. Although these enhancements immediately find practical application in this work, they are stand-alone results that can be re-used in other works.

The translation function from spi calculus processes to Java statements have been formally expressed and proven sound in this work. This result has been reached in a modular way. First, it has been shown that the generated Java code correctly refines the original spi calculus process, by relying on a custom library, called SpiWrapper, whose semantics has been formally specified. Then, part of a SpiWrapper implementation has been shown to be correct with respect to its intended semantics. Verification of the full SpiWrapper library is left for future work.

Finally, the approach has been validated by a case study on the SSH Transport Layer Protocol (SSH-TLP), by using the Spi2Java framework and implied methodology to implement a client of such protocol. The case study covers a rich portion of the SSH-TLP, by including support for run-time algorithm negotiation, error conditions handling and key store handling. The model of the full SSH-TLP, including client, server and auxiliary processes, has been formally verified for secrecy and authentication. So, the generated client application is implied to be secure with respect to secrecy and authentication, modulo the manual implementation of the encoding functions, for which invertibility has not been formally proven. Nevertheless, such encoding functions are rather simple, and they have been extensively tested and manually reviewed, so that high confidence about their invertibility can be achieved.

Sometimes the MDD methodology cannot be used to generated new secure implementations, because legacy implementations are already in place, that cannot be substituted. This problem has been taken into account, and in such circumstances a monitoring approach is proposed in this work to overcome the former limitation of the MDD approach. The monitoring approach plugs into the MDD methodology: starting form the formal model of a protocol agent, a monitor implementation for that agent is generated, instead of a new implementation of the agent itself. The obtained monitor is then coupled with several different legacy implementations; incorrect protocol sessions that could possibly be incorrectly accepted by the legacy implementations are stopped by the monitor.

Technically, the formal model of a protocol agent is transformed into the model of its monitor, then the monitor is generated by using the proposed MDD methodology. The transformation function has been formally defined in this work, so that it is possible to reason about its properties, but the formal proof of soundness is left for future work. In principle, it should be possible to claim that the monitor stops all incorrect protocol sessions.

The monitoring approach has been validated by developing a monitor for the server

role of the SSL protocol. When using the online paradigm, that is when the monitor checks all data before they reach the intended recipient, the monitor effectively stops malicious sessions, avoiding known vulnerabilities of legacy implementations such as OpenSSL to be exploited. The monitor even hinted us in spotting a new flaw in the JESSIE client implementation. Online monitoring introduces some delay; while in most cases this delay is smaller than typical network times, and affect performances on a low percentage, an offline paradigm is also proposed. With the offline paradigm, data exchanged by the legacy application are logged and checked later in a batch process. This paradigm introduces no delay, and can still timely report attacks, if the batch checks are run often enough.

Appendix A

Proofs for Section 3.1

This appendix contains the proofs of theorem 1 and theorem 2. Moreover, the proofs of correctness for the `getLeft()`, `getRight()` and `equals()` methods of the `Pair` class belonging to the SpiWrapper library are given.

A.1 Proof of Theorem 1

Formally, the context is denoted by $\mathcal{C}[\cdot]$, where the central dot '$\cdot$' represents the hole in the context; by stating that a context $\mathcal{C}[\cdot]$ compiles, we mean that the Java program $\mathcal{C}[\text{“”}]$, obtained by filling the hole in the context with an empty string, compiles.

In order to prove theorem 1, a lemma is first introduced.

Lemma 2. *Let $\Gamma \vdash M : A$, built $\supseteq fnv(M)$ and return $\subseteq SpiTerm$. If built is closed under the subterms function, if $\mathcal{C}[\cdot]$ is a Java context that compiles, and that includes the declaration and initialization of all and only Java variables that correspond to the terms in built, then $\mathcal{C}[tr_t(M,built,return)\ \cdot]$ is a context that compiles, and that includes the declaration and initialization of all and only Java variables that correspond to the terms in ub(M,built).*

Proof. If $M \in built$, then tr_t always returns the empty string and, by hypotheses, $\mathcal{C}[\text{“”}\ \cdot] = \mathcal{C}[\cdot]$ compiles and declares and initializes all terms in $ub(M,built) = built$, so the case is proven. The interesting case is when $M \notin built$.

The proof is carried out by structural induction over M. The base case, where M is a name or variable n, is trivial: $fnv(n) = \{n\} \subseteq built$, so this case cannot happen (it is an instance of the $M \in built$ case). Some interesting inductive cases are now analyzed.

case $M = (N,N')$ By definition, $fnv((N,N')) = fnv(N) \cup fnv(N')$. In particular, $built \supseteq fnv(N)$, so the inductive hypotheses apply, and

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(N,built,return)\ \cdot]$$

compiles and builds all and only terms in $ub(N,built)$. Similarly, since $ub(N,built) \supseteq fnv(N')$, the inductive hypotheses apply to the code that follows too, so

$$\mathcal{C}''[\cdot] = \mathcal{C}'[tr_t(N',ub(N,built),return)\ \cdot]$$

compiles and builds all and only terms in $ub(N',ub(N,built)) = ub(N,built) \cup ub(N',built)$. Note that properly updating the *built* set allows us to satisfy the inductive hypotheses and to avoid duplicate variable declarations when invoking tr_t on N' too.

By the canonical forms lemma, it follows that (N,N') must have the $A_1 \times A_2$ type, whose corresponding SpiWrapper class offers the $T((M,N))$`(A_1 left, A_2 right, ...)` constructor. `A_1` and `A_2` are the ASCII representations of the A_1 and A_2 types respectively. The dots '`...`' represent auxiliary parameters, matching the number of cryptographic algorithms and parameters specified in the eSpi specification XML document, needed to make the generated application interoperable; they can be neglected in this context. By the stated assumptions, $Ts((M,N)) <: T((M,N))$, so the constructor $Ts((M,N))$`(A_1 left, A_2 right, ...)` exists. Moreover, since (N,N') is well formed, and the (T-Pair) is the only rule that could be applied, it follows that N must have type A_1 and N' must have type A_2, so it is verified that the generated code is well formed in $\mathcal{C}''[\cdot]$, and all and only the terms in $ub((N,N'),built)$ are built.

case $M = \{L\}_N$ By definition, $fnv(\{L\}_N) = fnv(L) \cup fnv(N)$. So, like in the pair case, the inductive hypotheses apply, and

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(L,built,return)\ tr_t(N,ub(L,built),return)\ \cdot]$$

compiles and builds all and only the variables in $ub(L,built) \cup ub(N,built)$. By the canonical form lemma, $\{L\}_N$ must have type $ShKC\langle A\rangle$, whose SpiWrapper class offers the $T(\{L\}_N)$`(A plaintext, ShKey key,...)` constructor. Again, it is assumed that the corresponding constructor in $Ts(\{L\}_N)$ exists. Moreover, by the well formedness of $\{L\}_N$, it follows that L must have type A and N must have type $ShKey$, so the generated code is well formed in $\mathcal{C}'[\cdot]$, and all and only the terms in $ub(\{L\}_N,built)$ are built.

□

Proof of theorem 1. The following will be proven, which implies the theorem. Let *built* be closed under the *subterms* function, and $built \supseteq fnv(P)$. For any context $\mathcal{C}[\cdot]$ that compiles and that includes the declaration and initialization of all and only Java variables that correspond to the terms in *built*, the $\mathcal{C}[tr_p(P,built,return)]$ Java program compiles too. Note that the main theorem is implied, because it is a particular instance of this predicate: $built = fnv(P)$, and all free names and variables are declared and assigned in the context.

The proof is carried out by structural induction over the process P. The base case, where P is $\mathbf{0}$, is trivial, because the empty string is well formed by hypotheses.

Some significant inductive cases are now reported.

case P **is** $\overline{M}\langle N\rangle.Q$ By the well formedness of P, being (P-Out) the only applicable rule, it follows that M must have type $Channel$ and N must have type $Message$. By definition, $fnv(P) = fnv(M) \cup fnv(N) \cup fnv(Q)$. Note that, being a $Channel$,

the type system enforces M to be a free name. For this reason, M is already built by hypotheses. Formally $built \supseteq fnv(M) = subterms(M) = \{M\}$, so there is no need to build M invoking $tr_t(\cdot)$ on it. Regarding N, by hypotheses, it follows that $built \supseteq fnv(N)$, and lemma 2 can be applied. So

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(N,built,return)\ \cdot]$$

compiles and builds all and only terms in $ub(N,built)$.

Since M is a $Channel$, and the SpiWrapper class implementing the $Channel$ type offers the `void send(Message toSend)` method, the following context

$$\mathcal{C}''[\cdot] = \mathcal{C}'[\texttt{M.send(N);}\ \cdot]$$

compiles and does not declare any other Java variable.

Since

$$ub(N,built) \supseteq built \supseteq fnv(P) \supseteq fnv(Q)$$

it follows that the inductive hypotheses hold, so

$$\mathcal{C}''[tr_p(Q,ub(N,built),return)]$$

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

case P **is** $M(x).Q$ By well formedness of P, it follows that M must have type $Channel$ and M must be a free name. By definition, $fnv(P) = fnv(M) \cup fnv(Q) \setminus \{x\}$ because x is bound by the input process. By the hypothesis stating that free and bound variables are disjoint sets, $x \notin fnv(M)$ must hold too. So M is already built by hypotheses, formally $built \supseteq fnv(M) = subterms(M) = \{M\}$.

The SpiWrapper class implementing the $Channel$ type offers the `T receive(Class<T> x,...)` method, that can indeed be invoked. Moreover, whatever is the type inferred for x, by the assumption $Ts(x) <: T(x)$, the assignment of variable x is well formed. So

$$\mathcal{C}'[\cdot] = \mathcal{C}[\texttt{T x = M.receive(Ts.class,...);}\ \cdot]$$

compiles. Also note that the generated code declares and assigns `x`, so in $\mathcal{C}'[\cdot]$ the declared and assigned variables are $ub(x,built) \supseteq fnv(Q)$. So the inductive hypotheses hold, and

$$\mathcal{C}'[tr_p(Q,ub(x,built),return)]$$

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

case P **is** $(\nu n)Q$ By definition, $fnv(P) = fnv(Q) \setminus \{n\}$, which does not ensure that $built \supseteq fnv(n) = \{n\}$, so, in general, lemma 2 cannot be applied. Nevertheless, if $n \in built$, then $tr_t(n,built,return)$ generates the empty string, and n is already built in $\mathcal{C}[\cdot]$ by hypotheses, so $\mathcal{C}'[\cdot] = \mathcal{C}[\text{“”}\ \cdot]$ compiles and builds all and only terms in $ub(n,built) = built$. (By the way, the former case can never happen, because a fresh

name can never be already built.) Conversely, if $n \notin built$, by inspection of tr_t, it follows that

$$\mathcal{C}'[\cdot] = \mathcal{C}[\texttt{T n = new Ts(...); }\cdot]$$

compiles and all and only terms in $ub(n,built)$ are built in $\mathcal{C}'[\cdot]$.

By the hypothesis $built \supseteq fnv(P)$, it follows that $ub(n,built) \supseteq fnv(Q)$, so by inductive hypotheses,

$$\mathcal{C}'[tr_p(Q,ub(n,built),return)]$$

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

The proof structure is very similar for the remaining cases. In general, it is shown that

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(\ldots)\ \cdot]$$

compiles and ensures that all needed names and variables are built. Then, it is shown that the method `m` called on object `o` is available, and all its parameters match, because of the well formedness of P. Finally, it is shown that the inductive hypotheses apply, so that the translation of the remaining process is well formed, letting the whole generated code be well formed in $\mathcal{C}[\cdot]$. □

A.2 Proof of Theorem 2

In order to prove theorem 2, some lemmata are first introduced.

Lemma 3. *For any term M such that $\Gamma \vdash M : A$, for any Val,Res such that $dom(Val \circ Res \circ J)$ is closed under the subterms function, and for any substitution σ, there exists Val',Res' such that*

$$\begin{gathered}\sigma_{|fnv(M)} = Val \circ Res \circ J_{|fnv(M)} \ \wedge \ \sigma \supseteq Val \circ Res \circ J \\ \wedge \ (tr_t(M,dom(Val \circ Res \circ J),return),Val,Res \xrightarrow{\tau}{}^{*} unit,Val',Res') \\ \Rightarrow M\sigma = Val'(Res'(J(M))) \ \wedge \ \sigma \supseteq Val' \circ Res' \circ J\end{gathered}$$

and $dom(Val' \circ Res' \circ J)$ is closed under the subterms function.

Informally, this lemma assesses that, after execution, the Java code generated to build M, actually creates a Java object representing M and a variable $J(M)$ storing the reference to that object. More precisely, this lemma states that if the Java program memory correctly represents the terms in the corresponding spi calculus process, and the generated Java code building M can correctly execute, then in the final state, the Java memory is representing M, and all of its subterms.

Proof. If $M \in dom(Val \circ Res \circ J)$, by hypotheses, $M\sigma = Val(Res(J(M)))$. Moreover, tr_t returns the empty string, letting the final state equal to the initial state, so this case is proven.

If $M \notin dom(Val \circ Res \circ J)$, the proof is carried out by induction over the structure of M.

base case: M **is** n By definition, $fnv(n) = \{n\}$, which implies $n \in dom(Val \circ Res \circ J)$, so this case cannot happen, and it is proven as a particular instance of the $M \in dom(Val \circ Res \circ J)$ case.

inductive case: M **is** (N,N') By definition, $fnv((N,N')) = fnv(N) \cup fnv(N')$, in particular, for any name or variable n

$$n \in fnv(N) \Rightarrow n \in fnv((N,N')) \Rightarrow n \in dom(Val \circ Res \circ J)$$

So the inductive hypotheses hold, and

$$tr_t(N,dom(Val \circ Res \circ J),return),Val,Res \xrightarrow{\tau}^* unit,Val',Res'$$

where $N\sigma = Val'(Res'(J(N)))$, $\sigma \supseteq Val' \circ Res' \circ J$ and $dom(Val' \circ Res' \circ J)$ is still closed under the *subterms* function. Note that the inductive hypotheses hold for the appended code too, so

$$tr_t(N',dom(Val' \circ Res' \circ J),return),Val',Res' \xrightarrow{\tau}^* unit,Val'',Res''$$

where $N'\sigma = Val''(Res''(J(N')))$, $\sigma \supseteq Val'' \circ Res'' \circ J$ and $dom(Val'' \circ Res'' \circ J)$ is closed under the *subterms* function.

In particular, $Val''(Res''(J(N))) = N\sigma$ and $Val''(Res''(J(N'))) = N'\sigma$, so by the (T-Pair) semantic rule the appended code executes as

$$\begin{array}{l} T((N,N'))\ J((N,N')) \\ \quad \texttt{= new}\ Ts((N,N'))\texttt{(}J(N)\texttt{,}J(N')\texttt{,}Param((N,N'))\texttt{)},Val'',Res'' \xrightarrow{\tau}^* \\ \qquad T((N,N'))\ J((N,N'))\ \texttt{=}\ o,Val''',Res'' \xrightarrow{\tau} \\ \qquad\quad unit,Val''',Res''' \end{array}$$

where $Val''' = \{(o,(N,N')\sigma)\} \cup Val''$ and $Res''' = \{(J((N,N')),o)\} \cup Res''$

Finally, by noting that $dom(Val''' \circ Res''' \circ J)$ is still closed under the *subterms* function, it is possible to state that this case is proven.

Same reasoning applies to the remaining inductive cases. □

Lemma 4. *For any spi calculus process $((\nu\overline{n})P)\sigma$ such that P does not begin with a replication, Val,Res such that $dom(Val \circ Res \circ J)$ is closed under the subterms function*

$$\begin{array}{c} \sigma_{|fnv((\nu\overline{n})P)} = Val \circ Res \circ J_{|fnv((\nu\overline{n})P)} \ \wedge\ \sigma \supseteq Val \circ Res \circ J \ \wedge \\ tr_p((\nu\overline{n})P,dom(Val \circ Res \circ J),return),Val,Res \xrightarrow{\tau}^* \\ tr_p(P,dom(Val' \circ Res' \circ J),return),Val',Res' \\ \Rightarrow S(((\nu\overline{n})P)\sigma,(tr_p(P,dom(Val' \circ Res' \circ J),return),Val',Res')) \end{array}$$

This lemma handles the restriction process, showing that its translation corresponds to the translation of the process that follows, where the memory has been correctly set up.

Proof. The proof is carried out by induction over the number of restricted names.

case $(\nu\overline{n})$ **is empty** If there is no restricted name, then the initial state is equal to the final state and, by hypotheses, the simulation relation S holds in the final state.

case $(\nu\overline{n}) = (\nu m)(\nu\overline{m'})$ Note that $m\sigma = m$, because

$$((\nu m)(\nu\overline{m'})P)\sigma = (\nu m)((\nu\overline{m'})P\sigma)$$

and m is not a free name in $((\nu m)(\nu\overline{m'})P)$, so σ acts as the identity on m.

By the definition of tr_p, the

$$tr_t(m,dom(Val \circ Res \circ J),return)\ tr_p((\nu\overline{m'})P,ub(m,dom(Val \circ Res \circ J)),return)$$

functions are invoked. If $m \in dom(Val \circ Res \circ J)$, then by lemma 3, tr_t evolves in *unit* and $m\sigma = Val(Res(J(m)))$. (By the way, this case can never happen, because a fresh name can never be already built.) If $m \notin dom(Val \circ Res \circ J)$, then, by the definition of tr_t and by the (P-Restr) semantic rule

$$\begin{array}{c} T(m)\ J(m)\ \texttt{= new}\ Ts(m)\texttt{(}Param(m)\texttt{)},Val,Res \xrightarrow{\tau}{}^{*} \\ T(m)\ J(m)\ \texttt{=}\ o,Val',Res \xrightarrow{\tau} \\ unit,Val',Res' \end{array}$$

where $Val' = \{(o,m)\} \cup Val$ and $Res' = \{(J(m),o)\} \cup Res$. Note that the code appended by the *ret* function does not alter the state of the program, as far as the spi calculus simulation is concerned.

The inductive hypotheses hold, because $dom(Val' \circ Res' \circ J) = ub(m,dom(Val \circ Res \circ J))$, so

$$\begin{array}{l} tr_p((\nu\overline{m'})P,dom(Val' \circ Res' \circ J),return),Val',Ret' \xrightarrow{\tau}{}^{*} \\ \qquad\qquad tr_p(P,dom(Val'' \circ Res'' \circ J),return),Val'',Ret'' \end{array}$$

can execute, and

$$S(((\nu\overline{m'})P)\sigma,(tr_p(P,dom(Val'' \circ Res'' \circ J),return),Val'',Ret''))$$

holds. Since $m\sigma = m = Val''(Res''(J(m)))$, it follows that

$$S(((\nu m)(\nu\overline{m'})P)\sigma,(tr_p(P,dom(Val'' \circ Res'' \circ J),return),Val'',Res''))$$

holds too, thus proving the case.

□

Proof of theorem 2. The proof is carried out by inspection on the form of spi calculus processes. Note that the case when P is $\mathbf{0}$ can be neglected, because the generated Java code, which is the empty string, cannot evolve in any other state, so the theorem cannot be applied.

The most interesting cases are now reported. Since S holds in the initial state, it is not possible that the translated process P is a restriction. Restriction processes are indeed handled by lemma 4.

case P **is** $\overline{M}\langle N\rangle .Q$ By definition, $fnv(P) = fnv(M) \cup fnv(N) \cup fnv(Q)$. Moreover, being a $Channel$, M must be a free name, so $fnv(M) = \{M\}$ and by hypotheses $M\sigma = Val(Res(J(M)))$ holds. Regarding N,

$$\forall\, n \cdot n \in fnv(N) \Rightarrow n\sigma = Val(Res(J(n)))$$

It follows that lemma 3 can be applied, so

$$tr_t(N,dom(Val \circ Res \circ J),return),Val,Res \xrightarrow{\tau}{}^{*} unit,Val',Res'$$

where $N\sigma = Val'(Res'(J(N)))$ and $dom(Val' \circ Res' \circ J)$ is still closed under the *subterms* function.

In particular, $M\sigma = Val'(Res'(J(M)))$ and $N\sigma = Val'(Res'(J(N)))$. Then, by the (P-Out) semantic rule, the code executes as

$$J(M)\texttt{.send(}J(N)\texttt{)},Val',Res' \xrightarrow{\tau}{}^{*} \xrightarrow{M\sigma!N\sigma} \xrightarrow{\tau}{}^{*} unit,Val',Res'$$

(The steps where the Java variables $J(M)$ and $J(N)$ are substituted by their referenced values, $Res'(J(M))$ and $Res'(J(N))$ respectively, are included within the $\xrightarrow{\tau}{}^{*}$ transitions.)

As explained above, the *ret* function only adds an irrelevant side effect w.r.t. the spi calculus simulation. Let

$$j' = tr_p(Q,ub(N,dom(Val \circ Res \circ J)),return) = tr_p(Q,dom(Val' \circ Res' \circ J),return)$$

The state of the Java program is then (j',Val',Res').

On the spi calculus side, the process can evolve like

$$P\sigma = (\overline{M}\langle N\rangle .Q)\sigma \xrightarrow{M\sigma!N\sigma} Q\sigma$$

In general, Q can be a restriction $(\nu\overline{n})Q'$ (with zero or more restricted names). Since $fnv(P) \supseteq fnv(Q)$, lemma 4 can be applied, getting to the final result that the Q' process is proven to be S-related with its Java translation, which proves this case.

In principle, as opposed to the spi calculus semantics, the Java `send()` method could asynchronously return before the data are actually received by the other party; for example it could return as soon as data are buffered locally for later dispatch by the

underlaying operating system. This is not an issue: until data are not forwarded, this asynchronous behavior is simulated by the spi calculus traces where the attacker does not use the received data. The asynchronous channel behavior would be an issue with restricted channels, because the Java code could evolve to the next external state, while the spi calculus process would be stuck. The type system avoids restricted channels to be created, thus ruling out this issue.

case P **is** $M(x).Q$ By definition, $fnv(P) = fnv(M) \cup fnv(Q) \setminus \{x\}$. Being a $Channel$, M must be a free name. Moreover, since bound and free names and variables are disjoint sets, $x \notin fnv(M) = \{M\}$ holds. It follows that $M\sigma = Val(Res(J(M)))$.

By the (P-In) semantic rule,

$$T(x)\ J(x) = J(M)\texttt{.receive(}Ts(x)\texttt{.class,}Param(x)\texttt{)},Val,Res \xrightarrow{\tau}{}^{*} \xrightarrow{M\sigma?N\sigma} \xrightarrow{\tau}{}^{*}$$
$$T(x)\ J(x) = \mathcal{N},Val',Res \xrightarrow{\tau} unit,Val',Res'$$

where $Val' = \{(\mathcal{N},N)\} \cup Val$, and $Res' = \{(J(x),\mathcal{N})\} \cup Res$. Again, the code inserted by the ret function does not alter the state. Let

$$j' = tr_p(Q,ub(x,dom(Val \circ Res \circ J)),return) = tr_p(Q,dom(Val' \circ Res' \circ J),return)$$

The Java state is (j'',Val'',Res'').

On the spi calculus side, the process can evolve like

$$P\sigma = (M(x).Q)\sigma \xrightarrow{M\sigma?N\sigma} Q\sigma[^{N\sigma}/_{x}]$$

Like in the output case, Q can be a restriction process $(\nu\overline{m})Q'$; by lemma 4, the translation of Q' is shown to be S-related with its Java translation, thus proving the case.

case P **is** $[M\ is\ N]Q$ Lemma 3 can be applied twice, so

$$tr_t(M,dom(Val \circ Res \circ J),return)\ tr_t(N,ub(M,dom(Val \circ Res \circ J)),return),Val,Res$$
$$\xrightarrow{\tau}{}^{*}\ unit,Val',Res'$$

where $M\sigma = Val'(Res'(J(M)))$, $N\sigma = Val'(Res'(J(N)))$, and $Val' \circ Res' \circ J$ is still closed under the $subterms$ function.

Two cases must be considered, either $M\sigma \neq N\sigma$ or $M\sigma = N\sigma$. If $M\sigma \neq N\sigma$, by the (P-Match – false) semantic rule, and by applying the standard congruence and

evaluation rules for the `if` statement, we have

$$\begin{array}{l}
\texttt{if (!}J(M)\texttt{.equals(}J(N)\texttt{))} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau}{}^{*} \\
\texttt{if (!false)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau} \\
\texttt{if (true)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau} \\
\texttt{throw new MatchException()}{,}Val'{,}Res' \\
\hfill \xrightarrow{\tau} \\
\bot
\end{array}$$

Since this case does not lead to an external state, no simulation must be shown w.r.t. the spi calculus, so it can be disregarded.

If $M\sigma = N\sigma$, then by the (P-Match – true) semantic rule, and by applying the standard congruence and evaluation rules for the `if` statement, we have

$$\begin{array}{l}
\texttt{if (!}J(M)\texttt{.equals(}J(N)\texttt{))} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau}{}^{*} \\
\texttt{if (!true)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau} \\
\texttt{if (false)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res' \\
\hfill \xrightarrow{\tau} \\
tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return){,}Val'{,}Res'
\end{array}$$

Let

$$\begin{aligned}
j' = tr_p(Q{,}ub(M{,}dom(Val \circ Res \circ J)) \cup ub(N{,}dom(Val \circ Res \circ J)){,}return) \\
= tr_p(Q{,}dom(Val' \circ Res' \circ J){,}return)
\end{aligned}$$

then $(j'{,}Val'{,}Res')$ is the Java state after the `if` statement.

On the spi calculus side,

$$P\sigma = [M\sigma \ is \ N\sigma](Q\sigma) \xrightarrow{\tau} Q\sigma$$

By noting that $fnv(P) \supseteq fnv(Q)$, and that Q can be a restriction process $(\nu\overline{n})Q'$, by lemma 4, the translation of Q' is shown to be S-related with its Java translation, thus proving the case.

$\square$

Note that the initial state of a Java program could be an internal state, if the translated spi calculus process P begins with a restriction. Nevertheless, it is reasonable to assume that

$$\sigma_{|fnv(P)} = \sigma = Val \circ Res \circ J = Val \circ Res \circ J_{|fnv(P)}$$

because this means assuming that the user provided sensible values for all and only the free names and variables in P. So lemma 4 can be applied, enabling theorem 2, thus getting to the final result that the Java code simulates the spi calculus process from which it has been generated.

A.3 Proof of Correctness of the Pair Class

The proof of correctness for the `getLeft()`, `getRight()` and `equals()` methods is given here. Proof of correctness of the class constructor is given in section 3.1.2, where the source code of the `Pair` class and the intended semantics of the SpiWrapper library are also given.

A.3.1 GetLeft() and GetRight() Methods

Proof of correctness of the `getLeft()` *method.* By looking up the (P-SPlit – left) semantic rule, the initial state for `getLeft()` method invocation is

$$o.\texttt{getLeft();},\{(o,(M,N)),(\mathcal{M},M),(\mathcal{N},N)\},\emptyset$$

From expression (3.1),

$$Val(o) = (M,N) \Rightarrow H(o) = \left(Pair, \begin{array}{l}\texttt{left} \ \rightarrow \mathcal{M}\\ \texttt{right} \rightarrow \mathcal{N}\end{array}\right)$$

So, the initial MJ configuration for the `getLeft()` method invocation is

$$(\underbrace{\{(o,\left(Pair, \begin{array}{l}\texttt{left} \ \rightarrow \mathcal{M}\\ \texttt{right} \rightarrow \mathcal{N}\end{array}\right)),(\mathcal{M},(T_M,\mathbb{F}_M))\} \cup H_0}_{H_1}, \underbrace{[]}_{VS_0}, o.\texttt{getLeft();}, \underbrace{[]}_{FS_0})$$

for some type $T_M <: Message$ and some mapping function $\mathbb{F}_M$, such that the $\mathcal{M}$ object implements the M spi calculus term, and where H_0 contains information on $\mathcal{N}$ implementing N, which is irrelevant for this method execution.

The following steps lead to the final state.

$$
\begin{array}{rl}
 & (H_1, VS_0, o.\texttt{getLeft();}, FS_0) \\
\overset{\text{E-Method}}{\rightarrow} & (H_1, \underbrace{(\{\texttt{this} \rightarrow (o, Pair)\} \circ []) \circ VS_0}_{VS_1}, \texttt{return this.left;}, FS_0) \\
\overset{\text{EC-Return}}{\rightarrow} & (H_1, VS_1, \texttt{this.left;}, (\texttt{return } \cdot) \circ FS_0) \\
\overset{\text{EC-FieldAccess}}{\rightarrow} & (H_1, VS_1, \texttt{this}, (\cdot\texttt{.left}) \circ (\texttt{return } \cdot) \circ FS_0) \\
\overset{\text{E-VarAccess}}{\rightarrow} & (H_1, VS_1, o, (\cdot\texttt{.left}) \circ (\texttt{return } \cdot) \circ FS_0) \\
\overset{\text{E-Sub}}{\rightarrow} & (H_1, VS_1, o\texttt{.left}, (\texttt{return } \cdot) \circ FS_0) \\
\overset{\text{E-FieldAccess}}{\rightarrow} & (H_1, VS_1, \mathcal{M}, (\texttt{return } \cdot) \circ FS_0) \\
\overset{\text{E-Sub}}{\rightarrow} & (H_1, VS_1, \texttt{return } \mathcal{M}, FS_0) \\
\overset{\text{E-Return}}{\rightarrow} & (H_1, VS_0, \mathcal{M}, FS_0)
\end{array}
$$

The final MJ configuration corresponds to the

$$\mathcal{M}, \{(o,(M,N)), (\mathcal{M},M), (\mathcal{N},N)\}, \emptyset$$

state, which concludes the proof. □

The `getRight()` method verification is the same as the one for the `getLeft()` method.

A.3.2 Equals() Method

Proof of correctness of the `equals()` *method.* In order to assess correctness of the `equals()` method, all the three possible cases must be covered:

1. the two objects are equal, and the method returns `true`;
2. the two objects have the same type, but their contents differ, and the method returns `false`;
3. the two objects have different types, and the method returns `false`.

For brevity, only the first case is reported here, which we believe is significant enough. Moreover, only significant configurations are explicitly written. Finally, since some extensions introduced in [64] are required, transitions will be marked with the labels specified in that work, instead of those specified in [21].

By looking up the (P-Match – true) semantic rule, the initial state is

$$a\texttt{.equals(}b\texttt{);}, \{(a,(M,N)), (b,(M,N))\} \cup Val, Res$$

By expression (3.1), there exist $\mathcal{M}_a, \mathcal{M}_b, \mathcal{N}_a, \mathcal{N}_b$ such that

$$
\begin{array}{c}
Val(a) = (M,N) \Rightarrow \\
H(a) = \left(Pair, \begin{array}{l}\texttt{left} \rightarrow \mathcal{M}_a \\ \texttt{right} \rightarrow \mathcal{N}_a\end{array}\right) \wedge \\
Val(\mathcal{M}_a) = M \ \wedge \ Val(\mathcal{N}_a) = N
\end{array}
$$

and

$$
\begin{gathered}
Val(\mathfrak{b}) = (M,N) \Rightarrow \\
H(\mathfrak{b}) = \left(Pair, \begin{array}{l} \texttt{left} \;\; \rightarrow \mathcal{M}_{\mathfrak{b}} \\ \texttt{right} \rightarrow \mathcal{N}_{\mathfrak{b}} \end{array} \right) \wedge \\
Val(\mathcal{M}_{\mathfrak{b}}) = M \;\wedge\; Val(\mathcal{N}_{\mathfrak{b}}) = N
\end{gathered}
$$

So, the initial MJ configuration is

$$
(\underbrace{\{(\mathfrak{a}, \left(Pair, \begin{array}{l} \texttt{left} \;\; \rightarrow \mathcal{M}_{\mathfrak{a}} \\ \texttt{right} \rightarrow \mathcal{N}_{\mathfrak{a}} \end{array} \right)),(\mathfrak{b}, \left(Pair, \begin{array}{l} \texttt{left} \;\; \rightarrow \mathcal{M}_{\mathfrak{b}} \\ \texttt{right} \rightarrow \mathcal{N}_{\mathfrak{b}} \end{array} \right))\} \cup H_0}_{H_1}, \underbrace{[]}_{VS_0}, \mathfrak{a}\texttt{.equals(}\mathfrak{b}\texttt{);}, \underbrace{[]}_{FS_0})
$$

where H_0 contains information on $\mathcal{M}_{\mathfrak{a}},\mathcal{M}_{\mathfrak{b}}$ implementing M and $\mathcal{N}_{\mathfrak{a}},\mathcal{N}_{\mathfrak{b}}$ implementing N.

The steps in figures A.1, A.2 and A.3 lead to the final configuration, which corresponds to the final Java state

$$
\texttt{true},\{(\mathfrak{a},(M,N)),(\mathfrak{b},(M,N))\} \cup Val,Res
$$

thus completing the proof. □

$$(H_1, VS_0, a\texttt{.equals(}b\texttt{);}, FS_0)$$

$$\xrightarrow{\text{Call}}$$

$$(H_1, (\{\texttt{this} \to (a, Pair), \texttt{obj} \to (b, Pair)\} \circ []) \circ VS_0,$$
$$\texttt{boolean result = false; if (...) \{...\} return result;}, FS_0)$$

$$\xrightarrow{\text{Sequence}} \xrightarrow{\text{VarDecl}} \xrightarrow{\text{No-op}}$$

$$(H_1, \underbrace{(\{\texttt{this} \to (a, Pair), \texttt{obj} \to (b, Pair), \texttt{result} \to (\texttt{false}, boolean)\} \circ []) \circ VS_0}_{VS_1},$$
$$\texttt{if (obj instanceof Pair) \{...\} return result;}, FS_0)$$

$$\xrightarrow{\text{Sequence}} \xrightarrow{\text{FrmOpen}} \xrightarrow{\text{VarRead}} \xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_1, \texttt{if (}b\texttt{ instanceof Pair) \{...\}}, (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{FrmOpen}} \xrightarrow{\text{IofTrue}} \xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_1, \texttt{if (true) \{ Pair otherPair = (Pair) obj; ... \}},$$
$$(\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{IfTrue}} \xrightarrow{\text{BlkBegin}}$$

$$(H_1, \{\} \circ VS_1, \texttt{Pair otherPair = (Pair) obj; ...}, \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{Sequence}} \xrightarrow{\text{FrmOpen}} \xrightarrow{\text{FrmOpen}} \xrightarrow{\text{VarRead}} \xrightarrow{\text{FrmClose}}$$

$$(H_1, \{\} \circ VS_1, \texttt{(Pair) }b,$$
$$(\texttt{Pair otherPair = }\cdot) \circ (\texttt{boolean leftOK = ...}) \circ \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{Cast}} \xrightarrow{\text{FrmClose}}$$

$$(H_1, \{\} \circ VS_1, \texttt{Pair otherPair = }b,$$
$$(\texttt{boolean leftOK = ...}) \circ \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{VarDecl}} \xrightarrow{\text{No-op}} \xrightarrow{\text{Sequence}}$$

$$(H_1, \underbrace{\{\texttt{otherPair} \to (b, Pair)\} \circ VS_1}_{VS_2},$$
$$\texttt{boolean leftOK = this.getLeft().equals(otherPair.getLeft());},$$
$$\underbrace{(\texttt{result = ...; }) \circ \{\} \circ (\texttt{return result;}) \circ FS_0}_{FS_1})$$

$$\xrightarrow{\text{FrmOpen}} \xrightarrow{\text{FrmOpen}} \xrightarrow{\text{FrmOpen}} \xrightarrow{\text{VarRead}} \xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_2, a\texttt{.getLeft()},$$
$$(\cdot\texttt{.equals(...)}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\tau}{}^{*}$$

$$(H_1, VS_2, \mathcal{M}_a$$
$$(\cdot\texttt{.equals(otherPair.getLeft())}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\text{FrmClose}}$$

Figure A.1: MJ evolution steps for the `equals` method of the `Pair` class – Part I.

$$(H_1,VS_2,\mathcal{M}_a\texttt{.equals(otherPair.getLeft())}$$
$$(\texttt{boolean leftOK = }\cdot)\circ FS_1)$$
$$\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{VarRead}}{\longrightarrow}\overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1,VS_2,\mathcal{b}\texttt{.getLeft(),}$$
$$(\mathcal{M}_a\texttt{.equals(}\cdot\texttt{))}\circ(\texttt{boolean leftOK = }\cdot)\circ FS_1)$$
$$\overset{\tau}{\longrightarrow}{}^*$$
$$(H_1,VS_2,\mathcal{M}_b,$$
$$(\mathcal{M}_a\texttt{.equals(}\cdot\texttt{))}\circ(\texttt{boolean leftOK = }\cdot)\circ FS_1)$$
$$\overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1,VS_2,\mathcal{M}_a\texttt{.equals(}\mathcal{M}_b\texttt{),}$$
$$(\texttt{boolean leftOK = }\cdot)\circ FS_1)$$
$$\overset{\tau}{\longrightarrow}{}^*$$
$$(H_1,VS_2,\texttt{true},(\texttt{boolean leftOK = }\cdot)\circ FS_1)$$

Note that the $\mathcal{M}_a\texttt{.equals(}\mathcal{M}_b\texttt{)}$ method invocation returns `true` because we assume that both $\mathcal{M}_a$ and $\mathcal{M}_b$ are implementing the M spi calculus term, and, by induction, we can assume correctness of that `equals` method implementation.

$$\overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1,VS_2,\texttt{boolean leftOK = true},FS_1)$$
$$\overset{\text{VarDecl}}{\longrightarrow}\overset{\text{No-op}}{\longrightarrow}$$
$$(H_1,\underbrace{\{\texttt{otherPair}\rightarrow(\mathcal{b},Pair),\texttt{leftOK}\rightarrow(\texttt{true},boolean)\}}_{VS_3}\circ VS_1,$$
$$\texttt{result = leftOK \&\& ...;},\underbrace{\{\}\circ(\texttt{return result;})\circ FS_0}_{FS_2})$$
$$\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{VarRead}}{\longrightarrow}\overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1,VS_3\circ VS_1,\texttt{true \&\& this.getRight().equals(otherPair.getRight());},$$
$$(\texttt{result = }\cdot)\circ FS_2)$$
$$\overset{\text{FrmOpen}}{\longrightarrow}$$
$$(H_1,VS_3\circ VS_1,\texttt{this.getRight().equals(otherPair.getRight()),}$$
$$(\texttt{true \&\& }\cdot)\circ(\texttt{result = }\cdot)\circ FS_2)$$

By using the same reasoning as for the `this.getLeft().equals(otherPair.getLeft())` method invocation, this statement evolves to `true`.

Figure A.2: MJ evolution steps for the `equals` method of the `Pair` class – Part II.

$$(H_1, VS_3 \circ VS_1, \texttt{true}, (\texttt{true \&\& } \cdot) \circ (\texttt{result = } \cdot) \circ FS_2)$$
$$\overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1, VS_3 \circ VS_1, \texttt{true \&\& true}, (\texttt{result = } \cdot) \circ FS_2)$$
$$\overset{\text{AND}}{\longrightarrow} \overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1, VS_3 \circ VS_1, \texttt{result = true}, FS_2)$$
$$\overset{\text{VarWrite}}{\longrightarrow} \overset{\text{No-op}}{\longrightarrow}$$
$$(H_1, VS_3 \circ \underbrace{(\{\texttt{this} \rightarrow (a, Pair), \texttt{obj} \rightarrow (b, Pair), \texttt{result} \rightarrow (\texttt{true}, boolean)\} \circ []) \circ VS_0}_{VS_4},$$
$$\{\}, (\texttt{return result;}) \circ FS_0)$$
$$\overset{\text{BlkEnd}}{\longrightarrow} \overset{\text{No-op}}{\longrightarrow}$$
$$(H_1, VS_4, \texttt{return result;}, FS_0)$$
$$\overset{\text{FrmOpen}}{\longrightarrow} \overset{\text{VarRead}}{\longrightarrow} \overset{\text{FrmClose}}{\longrightarrow}$$
$$(H_1, VS_4, \texttt{return true;}, FS_0)$$
$$\overset{\text{Return}}{\longrightarrow}$$
$$(H_1, VS_0, \texttt{true}, FS_0)$$

Figure A.3: MJ evolution steps for the `equals` method of the `Pair` class – Part III.

Appendix B

Proofs for Section 3.2

This appendix contains the proofs of theorem 3 (section 3.2.2) and theorem 5 (section 3.2.3). In order to avoid ambiguities, it is worth reminding that each proof refers to the notation used in the section where the corresponding theorem is stated. For instance, in the two sections, the symbol $SYSTEM'$ refers to two different systems, and so do the corresponding proofs.

B.1 Proof of Theorem 3

Proof. The following statement is proven, that implies the theorem: for all traces tr' of $SYSTEM'$ such that an attack exists in tr', there exists a trace tr'' of $SYSTEM''$, such that an attack exists in tr'' too. Formally,

$$\begin{aligned} &\forall\, tr' \in traces(SYSTEM') \cdot \neg SPEC(tr') \Rightarrow \\ &\quad \exists\, tr'' \in traces(SYSTEM'') \cdot \neg SPEC(tr'') \end{aligned}$$

In order to develop the proof, a new $SYSTEM'''$ is defined. Figure B.1 shows the diagram for actors A and B in this system.

$$SYSTEM''' \triangleq (|||_{A \in Honest} P'_A) || E_INTRUDER$$

where $E_INTRUDER$ is the part inside the dashed box in figure B.1. It is defined by

$$\begin{aligned} E_INTRUDER \triangleq (((&|||_{A \in Honest} E_A(ext,int)) \\ &\parallel INTRUDER'(IK'_0)) \setminus \{int\}) \end{aligned} \tag{B.1}$$

where $ext = send,receive$ and $INTRUDER'(IK'_0) = INTRUDER(IK'_0)[[^{int}/_{ext}]]$.

A lemma is now introduced. It is proven below the proof of this theorem.

Lemma 5. *The intruder of $SYSTEM'''$ is a trace refinement of the intruder of $SYSTEM''$;*

$$INTRUDER(IK''_0) \sqsubseteq E_INTRUDER$$

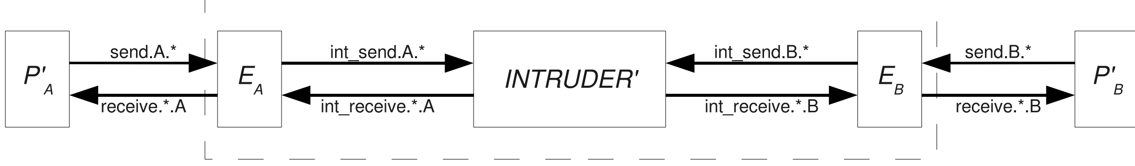

Figure B.1: Actors A and B with $E_INTRUDER$ in $SYSTEM'''$.

By lemma 5 and by refinement properties exposed in [63], it follows that

$$SYSTEM'' \sqsubseteq SYSTEM'''$$

Let us assume that a fault trace tr' exists in $traces(SYSTEM')$. It is worth noting that $SYSTEM'''$ is obtained from $SYSTEM'$, by swapping actual parameters of processes, and by injectively renaming communication channels of the $INTRUDER$ process. In particular, only the events *send* and *receive* are parameterized or renamed, and, since the renaming on $INTRUDER$ is injective, no non-determinism is introduced in the $INTRUDER'$ process. So, by condition (3.8), the trace $tr''' \in traces(SYSTEM''')$, that is obtained by renaming channels in tr', is a fault trace too. Finally, since $SYSTEM'' \sqsubseteq SYSTEM'''$ holds, tr''' is also a (fault) trace in $SYSTEM''$. Then, having shown that faults are preserved from $SYSTEM'$ to $SYSTEM''$, the theorem is proven. □

Proof of lemma 5. In order to carry out the proof, the states reachable from $E_INTRUDER$ must be written explicitly. For this reason, let E_i be defined as a generic state reachable from $|||_{A \in Honest} E_A$. Moreover, for each state E_i, let S_{E_i} be the set of all the encoded messages ready to be delivered to the $INTRUDER'$ component of $E_INTRUDER$, but not yet dispatched to it.

Formally, S_{E_i} is a set of messages that can be defined inductively. In the initial state

$$E_0 = |||_{A \in Honest} E_A$$

we have $S_{E_0} = \emptyset$. The evolution of S_{E_i} after an event e can be represented by an extension of the $\xrightarrow{e}$ state transition relation as follows: $S_{E_i} \xrightarrow{e} S_{E_j}$ means that the occurrence of e in a state where the set of encoded messages ready to be delivered to $INTRUDER'$ is S_{E_i} leads to a new state where the new set is S_{E_j}. Now, since the events e occurring in E_i can only take 4 different forms, the relation $\xrightarrow{e}$ on sets of messages can be defined by enumeration. By the definition of process E_A in (3.5), it follows that

$$\begin{array}{lcl}
S_{E_i} & \xrightarrow{send.A.B.(\text{Atom } \mathcal{L},(a,M))} & S_{E_i} \cup \{e_A(a,M)\} \\
S_{E_i} & \xrightarrow{receive.A.B.(\text{Atom } \mathcal{L},(a,M))} & S_{E_i} \\
S_{E_i} & \xrightarrow{int_send.A.B.e_A(a,M)} & S_{E_i} \setminus \{e_A(a,M)\} \\
S_{E_i} & \xrightarrow{int_receive.A.B.y} & S_{E_i}
\end{array}$$

A generic state of $E_INTRUDER$ then takes the form

$$E_INTRUDER_i \triangleq (E_i \parallel INTRUDER'(S'_i)) \setminus \{int\}$$

where S'_i is the current intruder knowledge. The initial state is

$$E_INTRUDER_0 = E_INTRUDER$$

that is

$$E_INTRUDER_0 = (E_0 \parallel INTRUDER'(S'_0)) \setminus \{int\}$$

where S'_0 is defined to be the set IK'_0.

Finally, let us define the total knowledge S_{T_i} associated with state $E_INTRUDER_i$ as

$$S_{T_i} \triangleq S_{E_i} \cup S'_i$$

It is worth noting that, in state $E_INTRUDER_i$, by lemma 1, $deds(S'_i) \subseteq deds(S_{T_i})$.

Let us preliminarily prove the following

Lemma 6.

$$E_INTRUDER_i \xrightarrow{\tau} E_INTRUDER_j$$
$$\Longrightarrow$$
$$S_{T_i} = S_{T_j}$$

Proof. There are two possible cases: τ comes from an *int_send* event, or τ comes from an *int_receive* event.

if τ comes from event *int_send.A.B.*$e_A(a,M)$**:** then, by the definition of each process E_A, a corresponding *send.A.B.*(ATOM $\mathcal{L}$,(a,M)) must have previously occurred. So, by the definition of S_{E_i},

$$e_A(a,M) \in S_{E_i}$$

and

$$S_{E_j} = S_{E_i} \setminus \{e_A(a,M)\}$$

and, by the definition of $INTRUDER'$,

$$S'_j = S'_i \cup \{e_A(a,M)\}$$

so $S_{T_i} = S_{T_j}$.

if τ comes from event *int_receive.A.B.y***:** then, by the definition of S_{E_i},

$$S_{E_j} = S_{E_i}$$

and, by the definition of $INTRUDER'$,

$$S'_j = S'_i$$

so $S_{T_i} = S_{T_j}$.

□

Now that lemma 6 is proven, the following property can be proven for each trace tr, which implies lemma 5:

$$(E_INTRUDER_0 \stackrel{tr}{\Longrightarrow} E_INTRUDER_f)$$
$$\Longrightarrow$$
$$\exists\ IK''_f \mid (INTRUDER(IK''_0) \stackrel{tr}{\Longrightarrow} INTRUDER(IK''_f) \wedge$$
$$deds(S_{T_f}) \subseteq deds(IK''_f))$$

The proof is based on induction on the length of trace tr.

Base ($length(tr) = 0$) Since tr is the empty trace

$$E_INTRUDER_0 \stackrel{\tau}{\longrightarrow}^{*} E_INTRUDER_f$$

Then, by lemma 6, we have that

$$S_{T_f} = S_{T_0} = S_{E_0} \cup S'_0 = IK'_0$$

Moreover, $INTRUDER(IK''_0)$ cannot execute internal events, so if we take

$$IK''_f = IK''_0$$

then, considering that by definition (3.7) it follows that

$$IK'_0 \subset IK''_0$$

we can conclude $deds(S_{T_f}) \subseteq deds(IK''_f)$.

Induction ($length(tr) = n + 1$) The trace tr is then composed of a subtrace tr' of length n, followed by the $n + 1^{th}$ event. There are three possible cases: the $n + 1^{th}$ event is a *send*,*receive* or *leak* event.

case $n + 1^{th}$ event is a *send* event: by the definitions of processes and by inductive hypotheses

$$E_INTRUDER_0 \stackrel{tr'}{\Longrightarrow} E_INTRUDER_i$$

Moreover:

$$E_INTRUDER_i$$
$$\stackrel{\tau}{\longrightarrow}^{*} \quad \stackrel{send.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))}{\longrightarrow} \quad \stackrel{\tau}{\longrightarrow}^{*}$$
$$E_INTRUDER_f$$

By lemma 6, S_{T_i} will remain unchanged for each τ transition before the *send* event, and S_{T_f} will remain unchanged for each τ transition after the *send* event. Moreover, by definition,

$$S_{T_f} = S_{T_i} \cup \{e_A(a,M)\}$$

At the other side, by inductive hypotheses, there exists IK''_i such that

$$INTRUDER(IK''_0) \stackrel{tr'}{\Longrightarrow} INTRUDER(IK''_i)$$

and

$$INTRUDER(IK''_i) \stackrel{send.A.B.(\text{ATOM } \mathcal{L},(a,M))}{\longrightarrow} INTRUDER(IK''_f)$$

where $IK''_f = IK''_i \cup \{(\text{ATOM } \mathcal{L},(a,M))\}$
By inductive hypotheses, $deds(S_{T_i}) \subseteq deds(IK''_i)$; by definition (3.6) of $e_A(a,M)$ and by lemma 1, it follows that $deds(e_A(a,M)) \subseteq deds((\text{ATOM } \mathcal{L},(a,M)))$. So

$$deds(S_{T_i} \cup \{e_A(a,M)\}) \subseteq deds(IK''_i \cup \{(\text{ATOM } \mathcal{L},(a,M))\})$$

thus $deds(S_{T_f}) \subseteq deds(IK''_f)$.

case $n+1^{th}$ event is a *receive* event: by process definitions and by inductive hypotheses

$$E_INTRUDER_0 \stackrel{tr'}{\Longrightarrow} E_INTRUDER_i$$

Moreover:

$$E_INTRUDER_i \stackrel{\tau}{\longrightarrow}^* \stackrel{receive.A.B.(\text{ATOM } \mathcal{L},(a,d_A(a,y)))}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^* E_INTRUDER_f$$

By lemma 6, S_{T_i} will remain unchanged for each τ transition before the *receive* event, and S_{T_f} will remain unchanged for each τ transition after the *receive* event. Moreover

$$S_{T_f} = S_{T_i}$$

At the other side, by inductive hypotheses, there exists IK''_i such that

$$INTRUDER(IK''_0) \stackrel{tr'}{\Longrightarrow} INTRUDER(IK''_i)$$

Then, we need to show that

$$INTRUDER(IK''_i) \stackrel{receive.A.B.(\text{ATOM } \mathcal{L},(a,d_A(a,y)))}{\longrightarrow} INTRUDER(IK''_f)$$

where $IK''_f = IK''_i$, because the intruder knowledge does not change on *receive* events.

Since, by inductive hypotheses, $deds(S_{T_i}) \subseteq deds(IK''_i)$, and it has been shown that both $S_{T_f} = S_{T_i}$ and $IK''_f = IK''_i$ hold, it is possible to conclude that $deds(S_{T_f}) \subseteq deds(IK''_f)$ holds too. In order to complete the proof of this case, it is enough to show that the *receive* event indeed can happen in $INTRUDER(IK''_i)$, that is, in order to send $(\text{ATOM } \mathcal{L},(a,d_A(a,y)))$ on the *receive* channel, $INTRUDER(IK''_i)$ must be able to derive the required message from its knowledge.

If a and y are derivable from IK''_i, then, by definition (3.6), $d_A(a,y)$ is derivable too, and, by applying the **pairing** rule, $(a,d_A(a,y))$ is derivable too. Message a is derivable from IK''_i with the **member** rule, since

$$a \in Encoding \subset IK''_0 \subseteq IK''_i$$

Message y is derivable from IK''_i because, since the *receive* event can happen in $E_INTRUDER_i$, then

$$y \in deds(S_{T_i})$$

and, by inductive hypotheses,

$$deds(S_{T_i}) \subseteq deds(IK''_i)$$

thus $y \in deds(IK''_i)$.
Finally, if $\textsc{Atom}\ \mathcal{L}$ is derivable from IK''_i, then, by applying the **pairing** rule, the required $(\textsc{Atom}\ \mathcal{L},(a,d_A(a,y)))$ message can be obtained. The message $\textsc{Atom}\ \mathcal{L}$ is derivable from IK''_i with the **member** rule, because

$$\textsc{Atom}\ \mathcal{L} \in IK''_0 \subseteq IK''_i$$

case $n+1^{th}$ **event is a** *leak* **event:** by process definitions and by inductive hypotheses

$$E_INTRUDER_0 \stackrel{tr'}{\Longrightarrow} E_INTRUDER_i$$

Moreover:

$$\begin{array}{c} E_INTRUDER_i \\ \stackrel{\tau}{\longrightarrow}^* \stackrel{leak.M}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^* \\ E_INTRUDER_f \end{array}$$

By lemma 6, S_{T_i} will remain unchanged for each τ transition before the *leak* event, and S_{T_f} will remain unchanged for each τ transition after the *leak* event. Moreover, the *leak* event is not engaged by the E_i process, so $S_{E_f} = S_{E_i}$, and, by the definition of intruder, $S_f = S_i$, so $S_{T_f} = S_{T_i}$.
At the other side, by inductive hypotheses, there exists IK''_i such that

$$INTRUDER(IK''_0) \stackrel{tr'}{\Longrightarrow} INTRUDER(IK''_i)$$

and

$$INTRUDER(IK''_i) \stackrel{leak.M}{\longrightarrow} INTRUDER(IK''_f)$$

where $IK''_f = IK''_i$. So, by inductive hypotheses, it follows that $deds(S_{T_f}) \subseteq deds(IK''_f)$. Finally, the $leak.M$ event can indeed happen in $INTRUDER(IK''_i)$ because, by hypotheses, the $leak.M$ event can happen in $E_INTRUDER_i$, which implies $M \in S_{T_i}$. Since, by inductive hypotheses, $deds(S_{T_i}) \subseteq deds(IK''_i)$, it follows that $M \in IK''_i$ too, so the $leak.M$ event can happen in $INTRUDER(IK''_i)$.

□

B.2 Proof of Theorem 5

Proof. In order to lighten notation, let us define

$$comm_A \triangleq \{send.A, receive?B.A\}$$
$$P_A^{dec-} \triangleq (P'_A \parallel DEC_A) \setminus priv_A$$

Since P_A^{dec-} has been introduced, an equivalent expression for $SYSTEM'$ is the following:

$$SYSTEM' = |||_{A \in Honest} P_A^{dec-} \parallel INTRUDER(IK'_0)$$

In order to complete the proof, the following lemma is needed. It is proven below.

Lemma 7. *If* (3.16) *and* (3.17) *hold, then, for any* $A \in Honest$*,*

$$(P_A^* \parallel INTRUDER(IK_0^*)) \setminus comm_A \sqsubseteq (P_A^{dec-} \parallel INTRUDER(IK'_0)) \setminus comm_A$$

Lemma 7 states that, under the same assumptions used by theorem 5, if communication channels are hidden, then one abstract protocol logic P_A^* acting with the intruder, is refined by its more detailed protocol logic P_A^{dec-}, that explicitly models the decoding process, acting with the intruder.

The proof idea for theorem 5 is to refine all abstract protocol logics P_A^* in $SYSTEM^*$ into their refined counterparts P_A^{dec-}, one at a time, thus refining the whole $SYSTEM^*$ to $SYSTEM'$. Note that it is not possible to trivially infer this result directly from lemma 7. Indeed, it is true that, for any processes A and R, and for any context $C[\,]$,

$$A \sqsubseteq R \Rightarrow C[A] \sqsubseteq C[R]$$

So, let us consider that lemma 7 holds for honest actor H. Then, setting the context to

$$C[X] = (|||_{A \in Honest \setminus \{H\}} P_A^* \parallel X) \setminus comm$$

leads to a process

$$(|||_{A \in Honest \setminus \{H\}} P_A^* \parallel ((P_H^* || INTRUDER(IK_0^*)) \setminus comm_H)) \setminus comm$$

which is not trivially proven to have all and the same traces of $SYSTEM^*$.

The proof steps of trace refinement reported here use two CSP operator properties that are introduced now.

The first property states that, for any processes P,Q,R, if $\alpha P \cap \alpha Q = \emptyset$, then

$$P \parallel (Q \parallel R) = (P \,|||\, Q) \parallel R$$

Remember that parallel operator without any subscripted event means synchronization on the intersection of the alphabets.

Informally, this property means that if P and Q cannot communicate directly (because, being the intersection of their alphabets empty, they cannot synchronize on any event), then, on one hand, P communicating with $Q \parallel R$, is actually only communicating with R;

on the other hand, Q is only communicating with R, and never with P. Thus, R acts as a proxy between P and Q, while the latter two processes can execute in interleaving.

The second property states that, for any processes P,Q,R, if $\alpha P \cap \alpha Q = \emptyset$, then

$$(P \parallel (Q \parallel R)) \setminus (\alpha Q \cap \alpha R) \cup (\alpha P \cap \alpha R) = (P \parallel ((Q \parallel R) \setminus (\alpha Q \cap \alpha R))) \setminus (\alpha P \cap \alpha R)$$

Informally, this property means that, since P and Q never communicate directly, and R is their proxy, from P's view it is irrelevant whether communication between Q and R is observable or not, thus allowing to put P in parallel either with $Q \parallel R$ or with $(Q \parallel R) \setminus (\alpha Q \cap \alpha R)$. However, from an observer point of view, communication between Q and R must always be hidden, so if it is not hidden with the $(Q \parallel R) \setminus (\alpha Q \cap \alpha R)$ process, it must be hidden at the top level process.

Indeed, in the used Dolev-Yao approach, any pair of protocol actors has disjoint alphabets, because the intruder is the only proxy between any pair of actors, and they can never communicate directly. Thus, these properties directly apply when P and Q are two processes representing interleaved actors and R is the intruder.

Trace refinement is proven by induction over the number of protocol logics that are step by step refined in $SYSTEM^*$.

base it will be proven that $SYSTEM^*$, where all protocol logics are abstract, is refined by a process where one protocol logic is refined, that is

$$\begin{aligned}
&\forall\ X \in Honest \\
&\quad (|||_{A \in Honest} P^*_A \parallel INTRUDER(IK^*_0)) \setminus comm \\
&\quad \sqsubseteq \left(\left(|||_{A \in Honest \setminus \{X\}} P^*_A ||| P^{dec-}_X \right) \parallel INTRUDER(IK'_0) \right) \setminus comm
\end{aligned}$$

The proof steps are:

$$
\begin{array}{ll}
 & (|||_{A\in Honest} P^*_A \parallel INTRUDER(IK^*_0)) \setminus comm \\
= & \big((|||_{A\in Honest\setminus\{X\}} P^*_A \;|||\; P^*_X\big) \parallel INTRUDER(IK^*_0)\big) \setminus comm \\
= & \big\langle \text{by letting } Others = |||_{A\in Honest\setminus\{X\}} P^*_A \big\rangle \\
 & ((Others \;|||\; P^*_X) \parallel INTRUDER(IK^*_0)) \setminus comm \\
= & \langle \text{by property of } ||| \text{ and } \parallel, \text{ and by } \alpha(Others) \cap \alpha P^*_X = \emptyset \rangle \\
 & (Others \parallel (P^*_X \parallel INTRUDER(IK^*_0))) \setminus comm \\
= & \langle \text{by hiding property; letting } comm^- = comm \setminus comm_X \rangle \\
 & (Others \parallel (P^*_X \parallel INTRUDER(IK^*_0) \setminus comm_X)) \setminus comm^- \\
\sqsubseteq & \langle \text{consider the context } Others \text{ surrounding the process refined in lemma 7} \rangle \\
 & \Big(Others \parallel \Big(\Big(P^{dec-}_X \parallel INTRUDER(IK'_0)\Big) \setminus comm_X\Big)\Big) \setminus comm^- \\
= & \langle \text{by hiding property} \rangle \\
 & \Big(Others \parallel \Big(P^{dec-}_X \parallel INTRUDER(IK'_0)\Big)\Big) \setminus comm \\
= & \langle \text{by property of } ||| \text{ and } \parallel, \text{ and by } \alpha(Others) \cap \alpha P^*_X = \emptyset \rangle \\
= & \Big(\Big(Others \;|||\; P^{dec-}_X\Big) \parallel INTRUDER(IK'_0)\Big) \setminus comm \\
= & \langle \text{by definition of } Others \rangle \\
= & \Big(\Big(|||_{A\in Honest\setminus\{X\}} P^*_A \;|||\; P^{dec-}_X\Big) \parallel INTRUDER(IK'_0)\Big) \setminus comm
\end{array}
$$

induction it will be shown that if the refinement relation holds when n actors have been refined, then it keeps holding when the $n+1^{th}$ actor is refined too. Let Ref be the

set of already refined actors, then $\forall\ X \in Honest \setminus Ref$

$$\Big(\Big(|||_{A \in Honest \setminus Ref} P_A^* \ ||| \ |||_{B \in Ref} P_B^{dec-}\Big) \parallel INTRUDER(IK_0')\Big) \setminus comm$$

$$= \Big(\Big(|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^* \ ||| \ |||_{B \in Ref} P_B^{dec-} \ ||| \ P_X^*\Big) \parallel INTRUDER(IK_0')\Big) \setminus comm$$

$$\sqsubseteq \Big\langle \text{by setting } Others = |||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^* \ ||| \ |||_{B \in Ref} P_B^{dec-}; \text{ by using the same steps as in base case} \Big\rangle$$

$$\Big(\Big(|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^* \ ||| \ |||_{B \in Ref} P_B^{dec-} \ ||| \ P_X^{dec-}\Big) \parallel INTRUDER(IK_0')\Big) \setminus comm$$

$$= \Big(\Big(|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^* \ ||| \ |||_{B \in Ref \cup \{X\}} P_B^{dec-}\Big) \parallel INTRUDER(IK_0')\Big) \setminus comm$$

It is worth noting that, in the inductive step, the intruder knowledge in the abstract system (the one having n refined actors) is set to IK_0', and not IK_0^*. Indeed, after the first refinement step made in the base case, the intruder knowledge is "restricted" to IK_0', the refined one. This is not an issue during the inductive step, because condition (3.17) in lemma 7 requires, in the abstract system, that the intruder knowledge is a *weak* superset of the intruder knowledge in the refined system. □

Proof of lemma 7. A weak simulation relation between the abstract process

$$P_A^* \parallel INTRUDER(IK_0^*) = f(P_A') \parallel INTRUDER(IK_0^*)$$

and the refined process

$$P_A^{dec-} \parallel INTRUDER(IK_0') = ((P_A' \parallel DEC_A) \setminus priv_A) \parallel INTRUDER(IK_0')$$

is first proven, which is then shown to imply the desired trace refinement. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a refined (also called *concrete* in other works) process, but each process is still allowed to perform any internal step in between two external states. More details about the weak refinement used here can be found, for example, in [65].

An external state of the refined protocol logic of actor A is defined as a generic state P_{A_i}' of the process P_A', such that P_{A_i}' does not begin with an action in the $priv_A$ set; that is, P_{A_i}' does not take the form $ev \to Q$, with $ev \in priv_A$ (note that, by construction of P_A', if an event $ev \in priv_A$ can occur, it is the only event that can occur). Accordingly, an external state of the refined process takes the form

$$SYSTEM_{A_i}' = ((P_{A_i}' \parallel DEC_A) \setminus priv_A) \parallel INTRUDER(IK_i')$$

because, by construction of P_A' and DEC_A, the latter process will always be in its initial state, which is DEC_A, when the former is in an external state.

In the same way, an external state $P^*_{A_i}$ of the abstract protocol logic is defined as a generic state of $f(P'_A)$ that is not ready to perform an event that is in $priv_A$. With this definition it turns out that *any* state of $f(P'_A)$ is an external state, since all actions in $priv_A$ have been removed by $f(\cdot)$. Then, any external state of the abstract process takes the form

$$SYSTEM^*_{A_i} = P^*_{A_i} \parallel INTRUDER(IK^*_i)$$

The relation R that binds external states of the abstract process to external states of the refined one is formally defined as

$$\begin{array}{c} R(SYSTEM^*_{A_i}, SYSTEM'_{A_i}) \\ \Leftrightarrow \\ P^*_{A_i} = f(P'_{A_i}) \ \wedge \ IK^*_i \supseteq IK'_i \end{array}$$

Note that relation R holds on the initial states of the abstract and refined processes. Indeed, P'_A must be an external state, because, by construction, it cannot begin with an action in $priv_A$. Moreover, by hypothesis (3.17), $IK^*_0 \supseteq IK'_0$ holds.

Let us denote with P/ev ("P after ev") the state of process P after it has engaged the event ev; if P/ev is not applicable, that is, P cannot engage the event ev, then $P/ev = P$. With this definition, the operator / ("after") is distributive with respect to the operator $\parallel$, which synchronizes on the intersection of the alphabets, that is

$$(P \parallel Q)/ev = P/ev \parallel Q/ev$$

The after operator is then overloaded to sequences of events in the obvious way. In this work, $\langle ev_1, ev_2 \rangle$ denotes the sequence of events ev_1 and ev_2.

In order to show that the weak simulation relation holds, it is enough to show that, for any external states $SYSTEM'_{A_i}, SYSTEM^*_{A_i}$, and event sequence ev_s:

$$\begin{array}{ccc} R(SYSTEM^*_{A_i}, SYSTEM'_{A_i}) & \wedge & SYSTEM'_{A_i} \stackrel{ev_s}{\rightarrow} SYSTEM'_{A_i}/ev_s \\ & \Longrightarrow & \\ SYSTEM^*_{A_i} \stackrel{ev^*_s}{\rightarrow} SYSTEM^*_{A_i}/ev^*_s & \wedge & R(SYSTEM^*_{A_i}/ev^*_s, SYSTEM'_{A_i}/ev_s,) \end{array} \tag{B.2}$$

where $\stackrel{ev_s}{\rightarrow}$ denotes the concatenation of state transitions for all events in ev_s, and ev^*_s is the sequence of events obtained by stripping all τ events from ev_s.

Note that, by the definition of intruder, $INTRUDER(IK^*_i)/ev^*_s = INTRUDER(IK^*_j)$, where IK^*_j is the new intruder knowledge reached after ev^*_s, and, by the distribution property of the after operator

$$\begin{aligned} SYSTEM^*_{A_i}/ev^*_s &= (f(P'_{A_i}) \parallel INTRUDER(IK^*_i))/ev^*_s \\ &= f(P'_{A_i})/ev^*_s \parallel INTRUDER(IK^*_i)/ev^*_s = f(P'_{A_i})/ev^*_s \parallel INTRUDER(IK^*_j) \end{aligned}$$

The same reasoning applies in the refined model.

Now all the possible event sequences ev_s that can lead to one of the next external states must be considered. Note that ev_s cannot start with a τ step, because it starts from an

external state. Also note that it is enough to prove (B.2) for event sequences ev_s such that no proper subsequence of ev_s leads to an external state because then the most general case descends by induction. Then, if ev_s starts with a $claimSecret$,$running$,$finished$,$leak$ or $send$ event, it is possible to consider only the case when it is composed of just one event, because after the first event an external state is reached.

Let us first consider these cases, where $ev_s = \langle ev \rangle = ev_s^*$ and let us show that (B.2) holds.

case $ev = claimSecret.A.B.M$ By hypothesis this event can happen in the refined system. If we let $P'_{A_j} = P'_{A_i}/ev$, we have

$$((P'_{A_i} \parallel DEC_A)\backslash priv_A)/ev = (P'_{A_i}/ev \parallel DEC_A/ev)\backslash priv_A = (P'_{A_j} \parallel DEC_A)\backslash priv_A$$

because $ev \notin priv_A$ and DEC_A is only engaged in the events in $priv_A$.

Since ev can occur in P'_{A_i}, and $P'_{A_i} \xrightarrow{ev} P'_{A_j}$, by the properties of $f(\cdot)$, it can be concluded that

$$f(P'_{A_i}) \xrightarrow{ev} f(P'_{A_j})$$

that is, ev can also happen in the abstract system, and the states of the abstract and refined protocol logics after ev are bound by $f(\cdot)$.

Moreover, after event ev, intruder knowledges remain unchanged in both systems, so $IK_j^* \supseteq IK'_j$ holds, and it can be concluded that R holds after ev.

The same reasoning also applies for the $running$ and $finished$ events.

case $ev = leak.M$ Since this event is engaged by the intruder process, and not by the protocol logic, if ev can happen in the refined system, then M must be in the intruder knowledge, that is $M \in IK'_i$. Consequently, by the assumption $IK_i^* \supseteq IK'_i$, it follows that $M \in IK_i^*$ too, and ev can happen in the abstract system too. Moreover, the state of the protocol logic and the intruder knowledge remain unchanged after this event in both refined and abstract processes, so relation R still holds after it.

case $ev = send.A.B.M$ If this event can happen in the refined system, it can also happen in the abstract system, because, as with the $claimSecret$ event,

$$P'_{A_i} \xrightarrow{ev} P'_{A_j}$$

implies

$$f(P'_{A_i}) \xrightarrow{ev} f(P'_{A_j})$$

While the reasoning about the protocol logic states is the same as explained in the $claimSecret.A.B.M$ case, the reasoning about intruder knowledges changes. After the event ev happens in the refined system, we have

$$IK'_j = IK'_i \cup \{M\}$$

and similarly, in the abstract system,

$$IK_j^* = IK_i^* \cup \{M\}$$

Since, by hypothesis, $IK_i^* \supseteq IK'_i$ holds, then $IK_j^* \supseteq IK'_j$ holds too.

The cases that remain to be considered are when ev_s starts with a *receive.B.A.M* event. This event is followed by the τ steps coming from the sequence of the pairs of $priv_send_A.(y,a) \rightarrow priv_receive_A.N$ actions that have been added during refinement after the *receive* action. Note that an external state is reached only after all these τ steps have been completed and, after a *receive* event, the protocol logic is obliged by construction to execute the sequence of private actions before any further external action. Then, there is a single external state that can be reached by the protocol logic after a *receive* event. Note also that, being the protocol logic synchronized with the intruder, the latter can execute only internal actions, i.e. *leak* actions, while the protocol logic is executing the internal steps that follow a *receive* event. These *leak* events do not change the global state. In conclusion, the only event sequences that start with a *receive* event and that lead to an external state are those that, after the *receive* event, have a sequence of τ events, corresponding to all the private actions that follow the *receive* action in the protocol logic, with interleaved *leak* events generated by the intruder. All these sequences lead to a single external state. If the refined protocol logic cannot execute one of the private actions that follow a *receive* action, the protocol logic gets stuck and no external state is ever reached. So, the only case that must be considered in order to check (B.2) is when all the private actions are executed.

By using the distributive property of the after operator, again we analyze protocol logics first, and then intruder knowledges.

In the refined system we have

$$((P'_{A_i} \parallel DEC_A) \setminus priv_A)/ev_s = (P'_{A_j} \parallel DEC_A) \setminus priv_A$$

Note that the state of DEC_A does not change because after each pair of $priv_send_A$ and $priv_receive_A$ events, it returns to its initial state, and we are under the hypothesis that all private events happen.

Let

$$\begin{aligned} receive.B.A.T \rightarrow priv_send_A.(y_1,a_1) \rightarrow priv_receive_A.T_1 \rightarrow \dots \\ \rightarrow priv_send_A.(y_n,a_n) \rightarrow priv_receive_A.T_n \rightarrow P'_{A_j} \end{aligned}$$

be the sequence of action prefixes that are executed in P'_{A_i} when ev_s occurs.

By the definition of DEC_A it follows that the data exchanged between the protocol logic and DEC_A in each pair of events $priv_send_A.(M,a), priv_receive_A.N$ must satisfy equations $N = d_A(a,M)$ and $N \neq \text{ATOM } \mathcal{E}$. Then, if all private events can occur in the refined system, the previous *receive.B.A.T* action must have bound variables in T in such a way that $T_i = d_A(a_i,y_i)$ for all $1 \leq i \leq n$. But, by (3.16), this also implies that

$$e_A(a_i,T_i) = e_A(a_i,d_A(a_i,y_i)) = y_i$$

By the definition of $f(\cdot)$, we have that the abstract protocol logic in state $f(P'_{A_i})$ is ready to execute

$$receive.B.A.T^* \rightarrow f(P'_{A_j})$$

where $T^* = T\left[{}^{e_A(a_1,T_1)}/_{y_1}\right]\dots\left[{}^{e_A(a_n,T_n)}/_{y_n}\right]$, i.e. where T^* is T with the same binding of variables that occurs in the concrete system when all private events occur.

Then, if in the refined system an event $receive.B.A.M$ followed by all the subsequent private events can occur, the same event can occur in the abstract system too, and

$$f(P'_{A_i}) \stackrel{receive.B.A.M}{\rightarrow} f(P'_{A_j})$$

With respect to the intruder, since the event $receive.B.A.M$ can happen in the refined system, it follows that $M \in IK'_i$. Since $IK^*_i \supseteq IK'_i$, it follows that the event can happen in the abstract system too, because $M \in IK^*_i$. Moreover, since after a *receive* event intruder knowledges remain unchanged, it also follows that $IK^*_j \supseteq IK'_j$.

In order to complete the proof of the simulation relation, we have to show that *leak* events interleaved in ev_s can happen in the abstract system too. This descends from the fact that neither the *receive* event nor the τ events change the intruder knowledge. Then, the occurrence of a $leak.N$ event in ev_s implies that $N \in IK'_i$, which, in turn, by the hypothesis $IK'_i \subseteq IK^*_i$, implies $N \in IK^*_i$, which finally means that $leak.N$ can be executed by the intruder in the abstract system too.

The weak simulation relation that has been proven implies that any trace of the refined system $P_A^{dec-} \parallel INTRUDER(IK'_0)$ that leads to an external state is also a trace of the abstract system $P^*_A \parallel INTRUDER(IK^*_0)$. However it is still possible that a trace that leads to an internal state in the refined system is not a trace of the abstract system. By the cases that have just been analyzed, it can be realized that a trace of the refined system can lead to an internal state only when the last action performed by the protocol logic is a *receive*. Moreover, in this trace, the last *receive* event can be followed only by *leak* events. The longest prefix of this trace that leads to an external state is the one that is obtained by removing the last *receive* event and the subsequent *leak* events, and the proof previously given ensures that this is also a trace of the abstract system. Moreover, since the receive event does not change the intruder knowledge, we can conclude that any *leak* event following the receive event in the refined system could also occur, both in the refined and in the abstract systems, before the *receive* event. Then, we have that even when a trace tr of the refined system is not a trace of the abstract system, the abstract system can still execute a trace tr^* that differs from tr only in the last *receive* event. This lets us conclude that lemma 7, which involves processes with hidden *send* and *receive* events, holds. □

Appendix C

Weaker Sufficient Conditions for Fault-Preserving Renaming Transformations for Security Protocols

Several formal models can be used in order to reason about the desired security properties of security protocols. Most of them stem from the Dolev and Yao [27] original idea of representing cryptographic primitives as operations of a term algebra. The more refined the model, the more accurate the results. However, highly refined models of real-world security protocols are so complex, that it may be difficult to get them verified by the currently known approaches, such as model checking or theorem proving. In order to tackle this issue, Hui and Lowe [36] proposed fault-preserving simplifying transformations, that is functions that transform a refined model into a simpler, more abstract one, while preserving the violations of some security properties. Then, if one of the preserved security properties can be verified on the abstract, simpler model, that security property is implied in the refined, complex model too. This result has been used in some later papers. For example, in [43] web-service security protocol models are simplified using the results presented in [36].

A class of fault-preserving simplifying transformations introduced in [36] is the one of the so-called fault-preserving renaming transformations (FPRTs), that uniformly transform the protocol data without changing the protocol structure. In [36], sufficient conditions are given for FPRTs to preserve secrecy and authentication faults. The conditions for authentication faults preservation are based on the definition of one fundamental set, called *AgreementSet* (here *AgSet* for short), which is only informally given. The lack of a formal definition for *AgSet* allows different possible interpretations about the content of *AgSet* to be acceptable; in [36], in order to cope with all possible contents of *AgSet*, the given sufficient conditions for authentication faults preservation are so strong that they are difficult to be met. Indeed, some of the FPRTs introduced in [36] are claimed to satisfy these constraints, while in facts they do not, so that nothing can be concluded about their

soundness.

In this work we formalize one particular possible definition for $AgSet$, and we relate it with every other possible interpretation of $AgSet$; this leads us to expressing weaker sufficient conditions that replace the original ones given in [36] for a FPRT to preserve authentication faults, thus making it easier to prove authentication faults preservation. Moreover, by a counterexample we show that some FPRTs given in [36] do not meet the sufficient conditions required in that work. Furthermore, we show that the same functions meet the weaker sufficient conditions given here so that they can be formally proven sound for the first time.

C.1 Fault Preserving Renaming Transformations

In order to enable the simplest comparison between the enhancements proposed here and the original work in [36], the formalism used here to reason about security protocols is the same as the one used in [36], based on CSP [63]. Note that in section 3.2, a richer CSP datatype is used instead. Nevertheless, since the weakened condition and the theorem that comes along are independent of the datatype, the results obtained here still hold when richer datatypes than the one presented in [36] are used, as it is done, for example, in section 3.2 or in [43].

For completeness the formalism used in [36] is briefly recalled here. The $Atom$ set contains the atomic values; the $Key \subseteq Atom$ set contains the (atomic) keys, while K^{-1} represents the inverse of key K. The $HashFn$ set contains the identifiers of different cryptographic hash functions. Finally, the $Message$ set (i.e. the message space) is defined in [36] by the following datatype:

$$\begin{aligned} Message ::= \; & \textsc{Atom}\ Atom \mid \\ & \textsc{Pair}\ Message\ Message \mid \\ & \textsc{Encrypt}\ Message\ Key \mid \\ & \textsc{Hash}\ HashFn\ Message. \end{aligned}$$

As syntactic sugar, (M,M') stands for PAIR *M M'*, $\{M\}_K$ for ENCRYPT *M K* and $g(|M|)$ for HASH *g M*.

It is worth noting that this datatype distinguishes between different types of messages by tagging them. It follows that, during formal verification of CSP processes, this datatype does not allow to catch type flaws attacks, that are attacks based on type confusion. For example, if a process expects to receive an ATOM *a* message, it will never accept a PAIR *a a* message, because they have different tags. This feature is essential in the work in [36] and it is retained in this work. For example, it is used to claim authentication preservation of some FPRTs, by assuming that agreement only happens on specific types. The limitation of not catching type flaw attacks could be overcame by defining only one "universal" type, to which all messages belong. However, it is not trivial to show that the results in [36] still hold under this extension, and this is left for future work.

The datatype does not capture the associativity of pairing. For example, (ATOM *a*,(ATOM *b*,ATOM *c*)) and ((ATOM *a*,ATOM *b*),ATOM *c*) are different messages, but they should be considered

equal. For this reason the normal form is introduced, that lets all pairs associate to the right. In this work, like in [36], all messages are ensured to be in their normal form.

The whole system is defined as the interleaving of all honest agents acting with the intruder, formally:

$$SYSTEM \triangleq (|||_{A \in Honest} P_A) \parallel INTRUDER(IK_0)$$

where $Honest$ is the set of all honest agents, the CSP process P_A represents honest agent A's behavior, process $INTRUDER$ represents the intruder's behavior, and IK_0 the initial intruder knowledge. The intruder acts as the medium, thus being allowed to see, forge, modify or drop any message. A knowledge derivation relation $\vdash$ specifies how the intruder can generate new messages from previously learned messages. Formally

member $M \in U \Rightarrow U \vdash M$

pairing $U \vdash M \wedge U \vdash M' \wedge \neg pair(M) \Rightarrow U \vdash (M,M')$

splitting $U \vdash (M,M') \Rightarrow U \vdash M \wedge U \vdash M'$

encryption $U \vdash M \wedge U \vdash \text{ATOM}\ K \wedge K \in Key \Rightarrow U \vdash \{M\}_K$

decryption $U \vdash \{M\}_K \wedge U \vdash \text{ATOM}\ K^{-1} \Rightarrow U \vdash M$

hashing $U \vdash M \wedge g \in HashFn \Rightarrow U \vdash g(|M|)$

The $Agent$ set is defined as $Honest \cup \{INTRUDER\}$, and $Message$ is the set containing all messages that can be sent or received by the agents.

Honest agents and the intruder use the $send.A.B.M$ and $receive.A.B.M$ events to communicate. Actor A uses $send.A.B.M$ to send message M, apparently to actor B, and $receive.B.A.M$ to receive message M, apparently from B. Moreover, honest agents use the $running.A.B.M$ and $finished.A.B.M$ events to explicit agreement on data M, in order to express an authentication property. The $running.A.B.M$ event occurs when actor A thinks it is running the protocol with B, agreeing on some data M, that store some details about the run in question. The $finished.A.B.M$ event means that actor A thinks it has finished a protocol run apparently with B, agreeing on some data M.

A security property $SPEC$ is defined by means of a corresponding predicate on traces $SPEC(tr)$ that must hold for any trace tr of $SYSTEM$, formally

$$SYSTEM \textbf{ sat } SPEC \Leftrightarrow \forall\, tr \in traces(SYSTEM) \cdot\ SPEC(tr)$$

Injective authentication (or simply authentication) of agent B to agent A can be informally defined by the following statement.

> For each protocol run that A thinks it has finished with B, B must have started a protocol run with A, and both A and B must agree on some message M.

One way to formally capture this concept is to count the number of $finished.A.B.M$ and $running.B.A.M$ events that appear in a trace: if the number of $finished.A.B.M$ events is less than or equal to the number of $running.B.A.M$ events, then the trace is safe because A never finished a session that B never started. For this approach to work, the $running$ and $finished$ events must be properly put in honest agents' specifications. Specifically, A should emit the $finished$ events when a protocol run is completed. B should emit the $running$ events in the point of the protocol execution that B must reach in order to let A emit its $finished$ event; usually this will be just before the last *send* event from B that has a causal link with a message received by A. More details can be found in [47].

In order to formally define authentication, the concept of an "agreement set" $AgSet \subseteq Message$ is needed too. In [36], $AgSet$ is informally defined as the set

> "of sequences of messages upon which the agents should agree (for example, if it is required that the agents agree upon a key and a nonce, then the agreement set will be all sequences of the form $\langle K,N \rangle$ for keys K and nonces N)"

and authentication is formally defined by means of the following predicate on traces:

$$Agreement_{AgSet}(tr) \triangleq \forall\ A \in Agent; B \in Honest; M \in AgSet \cdot tr \downarrow finished.A.B.M \leq tr \downarrow running.B.A.M$$

where $tr \downarrow e$ is the number of events e occurring in trace tr. The predicate $Agreement_{AgSet}(tr)$ formally captures the given informal definition of authentication.

Note that in [36] $AgSet$ is defined as a set of message sequences rather than of messages, and agreement is expressed on message sequences. It turns out that this is unnecessary because it is possible to define a bijection between message sequences and (normal form) pairs. For example, the message sequence $\langle \text{ATOM } a, \text{ATOM } b, \text{ATOM } c \rangle$ can be univocally represented by the pair $(\text{ATOM } a, (\text{ATOM } b, \text{ATOM } c))$. Moreover, using message sequences would make the formalism of proofs more complex, because sequences would have to be handled specially, as they are not part of the datatype. For this reason in this work, differently from [36], $AgSet$ is defined as a set of messages, and any agreement made on a message sequence is expressed as an agreement on the corresponding normal form pair. This does not affect the scope of the result, it just makes the formalism simpler.

Finally, like for the other security properties, the process $SYSTEM$ satisfies authentication if and only if for all of its traces the authentication property is satisfied.

Note that $AgSet$ in this work refers to any of the possible interpretations of the agreement set, as informally defined in [36]. As it will be clear later on, knowing the formal definition of $AgSet$ is not needed for our purposes. All that is assumed is that $AgSet \subseteq Message$.

A Renaming Transformation is a function $f : Message \rightarrow Message$ that defines how messages in the original protocol are replaced by messages in the simplified protocol. The function $f(\cdot)$ is then overloaded to take events, traces and processes, such that all messages in the events, traces or processes are replaced.

Let $SYSTEM'$ be the simplified system, formally defined as

$$SYSTEM' \triangleq (|||_{A \in Honest} f(P_A)) \parallel INTRUDER(IK'_0)$$

and let U range over $2^{Message}$ and M over $Message$. In [36] the following theorems are proven.

Theorem 8.

$$U \cup IK_0 \vdash M \Rightarrow f(U) \cup IK'_0 \vdash f(M) \wedge \quad \text{(C.1)}$$
$$f(IK_0) \subseteq IK'_0 \quad \text{(C.2)}$$
$$\Rightarrow \forall\ tr \in traces(SYSTEM) \cdot f(tr) \in traces(SYSTEM')$$

That is, each trace $tr \in traces(SYSTEM)$ has a corresponding trace $f(tr) \in traces(SYSTEM')$, if condition (C.1), called the *distribution property*, holds and the initial intruder knowledges in the two systems are related by (C.2).

Theorem 9.

$$\forall\ M \in AgSet; M' \in Message \cdot M \neq M' \Rightarrow f(M) \neq f(M') \quad \text{(C.3)}$$
$$\Rightarrow\ (\ \neg Agreement_{AgSet}(tr) \Rightarrow \neg Agreement_{f(AgSet)}(f(tr))\ )$$

This theorem means that if condition (C.3) is satisfied, then if a particular trace tr constitutes a failure of authentication on the original protocol, then $f(tr)$ constitutes a failure of authentication on the simplified protocol.

Corollary 3. *If conditions* (C.1)*,* (C.2) *and* (C.3) *hold, then*

$$SYSTEM' \ \mathbf{sat}\ Agreement_{f(AgSet)} \Rightarrow SYSTEM\ \mathbf{sat}\ Agreement_{AgSet}$$

This corollary states that if conditions (C.1), (C.2) and (C.3) hold, then $f(\cdot)$ is a FPRT that preserves agreement on messages in $AgSet$.

C.2 Weakening the Original Sufficient Conditions

Note that condition (C.3) requires $f(\cdot)$ to be injective for all the messages belonging to $AgSet$, with respect to any other message. In this work, a new, weaker sufficient condition that replaces (C.3) is provided. One contribution is to show that there exist interesting functions that do not satisfy condition (C.3), while they satisfy the weakened condition. In practice, this means that the weakened condition can be used to prove soundness of some FPRTs, whereas in [36] nothing could be inferred about their soundness by using the former condition (C.3).

Let us introduce now the set $AgSet_F$, defined as the set that contains all and only the messages that appear in a $running$ or $finished$ event of any trace of the $SYSTEM$ process, upon which authentication is to be checked. Formally

$$M \in AgSet_F \ \Leftrightarrow\ \exists\ tr \in traces(SYSTEM); A \in Agent; B \in Honest \cdot \\ tr \downarrow finished.A.B.M > 0 \vee tr \downarrow running.B.A.M > 0 \quad \text{(C.4)}$$

In order to link the results presented here with those used in section 3.2, note that expression (C.4) is the same as the definition given in expression (3.4). Here however, $AgSet_F$

is formally distinguished from $AgSet$, whereas in section 3.2 this distinction is neglected, to keep the notation simpler and to not distract the reader with the subtle formal details being addressed here.

The new sufficient condition states that

$$\forall\ M \in (AgSet \cap AgSet_F), M' \in AgSet_F \cdot M \neq M' \Rightarrow f(M) \neq f(M') \qquad \text{(C.5)}$$

That is, $f(\cdot)$ must be *locally* injective, for all the messages in the intersection of $AgSet$ and $AgSet_F$, with respect to the messages in $AgSet_F$. The main weakening comes from $M' \in AgSet_F$, as opposite to the former $M' \in Message$, because, for example, unlike with condition (C.3), it is now possible that for some $M \neq M'$, such that $M \in AgSet \cap AgSet_F$ and $M' \in Message \setminus AgSet_F$, it holds $f(M) = f(M')$.

It may be argued that often $AgSet$ and $AgSet_F$ will be equal, and that in general it is not trivial to compute the messages belonging to $AgSet_F$. Moreover, if it happens that $AgSet_F = Message$, then conditions (C.3) and (C.5) coincide. Although this is true in principle, there are significant cases, like the ones that will be shown in this work, where it is actually possible to compute the messages belonging to $AgSet_F$, and they are a strict subset of $Message$. In these cases, appealing to the weaker condition (C.5) enables function soundness to be proven, where it was not possible before.

Note that condition (C.5) is essentially the same as condition (3.11) in section 3.2; only in condition (3.11) the subtle difference between $AgSet$ and $AgSet_F$ is neglected for simplicity, as explained above.

Before arriving at the final result, it is worth making explicit by a lemma what events account for $f(tr) \downarrow finished.A.B.f(M)$ and $f(tr) \downarrow running.B.A.f(M)$ respectively.

Lemma 8. *If condition* (C.5) *holds, then for any trace tr, $A \in Agent$, $B \in Honest$ and $M \in AgSet \cap AgSet_F$*

$$\begin{aligned} f(tr) \downarrow finished.A.B.f(M) &= tr \downarrow finished.A.B.M \\ f(tr) \downarrow running.B.A.f(M) &= tr \downarrow running.B.A.M \end{aligned}$$

Proof. In general, $f(tr) \downarrow finished.A.B.f(M)$ and $f(tr) \downarrow running.B.A.f(M)$ can be computed as

$$\begin{aligned} f(tr) \downarrow finished.A.B.f(M) &= tr \downarrow finished.A.B.M + \textstyle\sum_{M' \in ZC(M)} tr \downarrow finished.A.B.M' + \\ &\quad \textstyle\sum_{M'' \in AC(M)} tr \downarrow finished.A.B.M'' \end{aligned}$$

$$\begin{aligned} f(tr) \downarrow running.B.A.f(M) &= tr \downarrow running.B.A.M + \textstyle\sum_{M' \in ZC(M)} tr \downarrow running.B.A.M' + \\ &\quad \textstyle\sum_{M'' \in AC(M)} tr \downarrow running.B.A.M'' \end{aligned}$$

where

$$ZC(M) = \{M' \in Message \setminus AgSet_F \mid M' \neq M \ \wedge \ f(M') = f(M)\}$$

$$AC(M) = \{M'' \in AgSet_F \mid M'' \neq M \ \wedge \ f(M'') = f(M)\}$$

represent the sets of messages that collide with M and that respectively do not belong and belong to $AgSet_F$.

Since by the definition (C.4) of $AgSet_F$ no message in $ZC(M)$ can appear in *running* or *finished* events, it follows that

$$\sum_{M' \in ZC(M)} tr \downarrow finished.A.B.M' = \sum_{M' \in ZC(M)} tr \downarrow running.B.A.M' = 0$$

whence the name ZC, which stands for “Zero Counting”.

Moreover, by hypotheses, $M \in AgSet \cap AgSet_F$ and, by definition of $AC(M)$, for any $M'' \in AC(M)$, $M'' \in AgSet_F$ holds too. So, by condition (C.5), $M \neq M'' \Rightarrow f(M) \neq f(M'')$, which makes $AC(M) = \emptyset$ (whence the name AC, standing for “Avoided Colliding”), finally leading to

$$\sum_{M'' \in AC(M)} tr \downarrow finished.A.B.M'' = \sum_{M'' \in AC(M)} tr \downarrow running.B.A.M'' = 0$$

This lets us conclude that the lemma statement holds. □

Theorem 10, that is the replacement for theorem 9, can now be defined and proven using lemma 8. Note that theorem 10 is exactly the same as theorem 4 in section 3.2; it is just reported here and re-numbered in this section for completeness. Thus, the proof of theorem 10 also serves as the proof for theorem 4.

Theorem 10. *If condition* (C.5) *holds, then*

$$\neg Agreement_{AgSet}(tr) \Rightarrow \neg Agreement_{f(AgSet)}(f(tr))$$

Proof. Suppose tr constitutes a failure of agreement authentication on the original protocol, i.e. $\neg Agreement_{AgSet}(tr)$. Then, for some $A \in Agent$, $B \in Honest$, $M \in AgSet$, we have:

$$tr \downarrow finished.A.B.M > tr \downarrow running.B.A.M$$

By definition (C.4), this implies $M \in AgSet_F$ too, and then $M \in AgSet \cap AgSet_F$.

By condition (C.5) and $M \in AgSet \cap AgSet_F$, lemma 8 can be used to conclude that

$$\begin{aligned} f(tr) \downarrow finished.A.B.f(M) &= tr \downarrow finished.A.B.M \\ f(tr) \downarrow running.B.A.f(M) &= tr \downarrow running.B.A.M \end{aligned}$$

From this it follows that

$$f(tr) \downarrow finished.A.B.f(M) > f(tr) \downarrow running.B.A.f(M)$$

that is a failure of agreement in the simplified trace. □

It can be noted that the proof of theorem 10 is essentially the same as the proof of theorem 9 given in [36]. Indeed, the proof of theorem 9 happens to hold also when the weakened condition (C.5) replaces condition (C.3), as justified by lemma 8.

It can also be noted that $AgSet_F$ is a boundary set: all the messages outside it are irrelevant w.r.t. authentication, so they can safely collide with the messages upon which agreement is required.

Theorem 10 is a drop in replacement for theorem 9, because it requires a weaker sufficient condition in order to get to the same result. For this reason the following

Corollary 4. *If conditions* (C.1)*,* (C.2) *and* (C.5) *hold, then*

$$SYSTEM' \textbf{ sat } Agreement_{f(AgSet)} \Rightarrow SYSTEM \textbf{ sat } Agreement_{AgSet}$$

can be proven with the same proof given in [36] for corollary 3. Again, note that corollary 4 and corollary 1 of section 3.2 are exactly the same.

C.3 Using the Weakened Condition

In [36], a FPRT that removes all the encryptions contained in a set $Encs \subseteq Message$, by replacing each $\{M\}_K \in Encs$ with M, is defined.

$$\begin{array}{rcl} f(\text{ATOM } A) & = & \text{ATOM } A \\ f(M,M') & = & NF(f(M),f(M')) \\ f(\{M\}_K) & = & \begin{cases} f(M) & \text{if } \{M\}_K \in Encs \\ \{f(M)\}_K & \text{otherwise} \end{cases} \\ f(g(|M|)) & = & g(|f(M)|) \end{array}$$

In [36] it is claimed that this function is a sound FPRT because it satisfies condition (C.3). However, a counter-example is shown now where condition (C.3) is not met. This does not imply that the function is not sound; it simply means that the function does not meet the strong sufficient condition (C.3), so, according to the material presented in [36], nothing about its soundness could actually be concluded. Fortunately, it can be shown that the same function meets the weaker condition (C.5). Therefore, using theorem 10 it is possible to prove for the first time that the FPRT is sound. In practice, this means that all previous applications of the function are still fully valid. Only, there was no proof showing that they were sound. Here that proof is provided, by appealing to the weaker condition (C.5).

Technically, in [36] it is stated that, since this function is clearly injective on atoms, theorem 9 can be used to show that $f(\cdot)$ preserves authentication faults, provided agreement is only made on sequences of atoms, that is, when *AgSet* is a set of sequences of atoms. However, condition (C.3) does not hold because this function is injective on atoms, and hence on sequences of atoms, but it is not injective for the atoms in the *AgSet*, with respect to any other message, as required by condition (C.3). The following counterexample proves our claim. Suppose $\text{ATOM } A \in AgSet$, and $\{\text{ATOM } A\}_K \in Encs$. It follows that $\text{ATOM } A \neq \{\text{ATOM } A\}_K$ and $f(\text{ATOM } A) = f(\{\text{ATOM } A\}_K) = \text{ATOM } A$, which contradicts condition (C.3).

Fortunately, the weaker condition (C.5) proposed here can be easily shown to hold for this function. Since agreement on sequences of atoms is going to be proven, it follows that $AgSet_F$ contains only sequences of atoms (here represented as normal form pairs) because no honest agent would rise a *running* or $finished$ event on different data types. This can be easily obtained because the datatype used in this work tags data with their type. For example, it is possible to ensure that $AgSet_F$ only contains pairs of atoms, by letting honest agents check the type of any data M just before emitting the *running* or $finished$ events on M. So, the function is *locally* injective on all the messages (that is, pairs of

atoms) belonging to $AgSet_F$, thus letting condition (C.5) hold. From this, by theorem 10, it is possible to prove that this function preserves authentication faults when agreement is made on sequences of atoms, which was not correctly proven previously.

Moreover, preservation of authentication faults with agreement on any message, and not only on sequences of atoms, can be proven too for the first time, provided all the encryptions included in data that are in $AgSet_F$ are not removed in the transformation, that is, provided that

$$\forall\ M \in AgSet_F; M' \in subterms(M)\ \cdot\ M' \notin Encs \tag{C.6}$$

where $subterms(M)$ is the set containing M and all its subterms. If this condition holds, then constraint (C.5) is satisfied because the function can be proven to be *locally* injective on $AgSet_F$ (actually, it maps each $M \in AgSet_F$ onto itself) by plain structural induction over messages. Note, however, that differently from what happens with agreement on sequences of atoms, in this case it may not be trivial to ensure that condition (C.6) holds.

The same reasoning exposed in this section applies to some other similar FPRTs defined in [36], such as the one removing hashings and the one renaming atoms.

A FPRT that removes hashings is defined in [36] in a very similar way to the FPRT that removes encryptions. Given a set $Encs$ of hashings to be removed, the function definition is as follows.

$$\begin{array}{rcl}
f(\textsc{Atom}\ A) & = & \textsc{Atom}\ A \\
f(M{,}M') & = & NF(f(M){,}f(M')) \\
f(\{M\}_K) & = & \{f(M)\}_K \\
f(g(|M|)) & = & \left\{\begin{array}{ll} f(M) & \text{if } g(|M|) \in Encs \\ g(|f(M)|) & \text{otherwise} \end{array}\right.
\end{array}$$

Again, this function is clearly injective on atoms. However, if authentication is made on atoms, condition (C.3) does not hold, and this can be shown by a counter-example very similar to the one given above. Nevertheless, if $running$ and $finished$ events are issued only on atoms, condition (C.5) is satisfied, leading to the conclusion that this function preserves authentication faults when agreement is made on atoms. This result can be extended to agreement on arbitrary messages too, with the same reasoning and caveats shown above.

Another interesting FPRT that was introduced in [36] is the one removing some atomic or hashed fields. Given the set Ms of applications of hash functions or atoms that are not keys, the FPRT is defined as

$$\begin{array}{rcl}
f(M) & = & \textsc{Atom}\ \texttt{nil} \quad \text{if } M \in Ms \\
f(\textsc{Atom}\ A) & = & \textsc{Atom}\ A \\
f(M{,}M') & = & \left\{\begin{array}{ll} f(M') & \text{if } f(M) = \textsc{Atom}\ \texttt{nil} \\ f(M) & \text{if } f(M') = \textsc{Atom}\ \texttt{nil} \\ (f(M){,}f(M')) & \text{otherwise} \end{array}\right. \\
f(\{M\}_K) & = & \left\{\begin{array}{ll} \textsc{Atom}\ \texttt{nil} & \text{if } f(M) = \textsc{Atom}\ \texttt{nil} \\ \{f(M)\}_K & \text{otherwise} \end{array}\right. \\
f(g(|M|)) & = & \left\{\begin{array}{ll} \textsc{Atom}\ \texttt{nil} & \text{if } f(M) = \textsc{Atom}\ \texttt{nil} \\ g(|f(M)|) & \text{otherwise} \end{array}\right.
\end{array}$$

where ATOM `nil` is a distinguished atom representing the empty message.

In [36] it is claimed that

> "if all the values upon which agreement is intended are atoms, and none are removed, that is:
>
> $$\forall\, M \in AgSet \cdot\ \exists\, A \in Atom \cdot\ M = \text{ATOM } A \wedge M \notin Ms \qquad \text{(C.7)}$$
>
> then the transformation satisfies condition (C.3), and so we deduce that this is a fault-preserving transformation with respect to agreement on such values."

Unfortunately, it is not true that the function satisfies condition (C.3), so nothing can be deduced about its fault-preserving behavior. The following counter example proves our claim. Suppose $M = \text{ATOM } A \in AgSet$ and $\text{ATOM } B \in Ms$. Then, by (C.7), $\text{ATOM } A \notin Ms$ and $f(\text{ATOM } A) = \text{ATOM } A$. If we now consider message $M' = (\text{ATOM } A, \text{ATOM } B) \in Message$, we have $f(M') = \text{ATOM } A = f(M)$, which contradicts condition (C.3).

Fortunately, condition (C.5) can be satisfied. In order to show this, let us define a weaker constraint

$$\forall\, M \in AgSet \cap AgSet_F \cdot\ \exists\, A \in Atom \cdot\ M = \text{ATOM } A \wedge M \notin Ms \qquad \text{(C.8)}$$

that is implied by (C.7). Any result obtained here by assuming (C.8) can thus be obtained when (C.7) is assumed. If $AgSet_F$ is enforced to contain only atoms (which is reasonable and easily viable since agreement is intended on atoms), condition (C.5) is satisfied because for any $M \in AgSet \cap AgSet_F$ constraint (C.8) ensures $f(M) = M$, and for any atom $M' \in AgSet_F$ such that $M' \neq M$, either $f(M') = M' \neq f(M)$ or $f(M') = \text{ATOM } \texttt{nil} \neq f(M)$. Note that constraint (C.8) can be easily enforced, for example by letting any honest agent emit *running* or *finished* events on M, only if M is an atom and it is not in Ms, which is easy to check because Ms is defined by the user. These checks actually let any message in $AgSet_F$ not being in Ms, thus implying (C.8).

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM Special Interest Group on Programming Languages Notices*, 36(3):104–115, 2001.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[4] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22:122–136, 1996.

[5] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[6] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In *International Symposium on Formal Methods for Components and Objects*, pages 70–101, 2007.

[7] M.R. Paterson K.G. Watson G.J. Albrecht. Plaintext recovery attacks against SSH. In *IEEE Symposium on Security and Privacy*, pages 16–26, 2009.

[8] Tuomas Aura. Strategies against replay attacks. In *IEEE Computer Security Foundations Workshop*, pages 59–68, 1997.

[9] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *ACM Conference on Computer and Communications Security*, pages 220–230, 2003.

[10] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008.

[11] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, Ümit Yalçinalp, and David Orchard. Web services policy 1.2 - framework (WS-policy). W3C Recommendation, 2006.

[12] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Hiroshi Maruyama, Anthony Nadalin, David Orchard, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey

Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, and Ümit Yalçinalp. Web services policy 1.2 - attachment (WS-policyattachment). W3C Recommendation, 2006.

[13] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, pages 363–366, 2000.

[14] Endre Bangerter, Jan Camenisch, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008.

[15] Andreas Bauer and Jan Jürjens. Security protocols, properties, and their monitoring. In *International Workshop on Software Engineering for Secure Systems*, pages 33–40, 2008.

[16] Giampaolo Bella. *Formal Correctness of Security Protocols*. Springer, 2007.

[17] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Computer Security Foundations Symposium*, pages 17–32, 2008.

[18] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verified reference implementations of WS-security protocols. In *Web Services and Formal Methods*, pages 88–106, 2006.

[19] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Greg O'Shea. An advisor for web services security policies. In *Secure Web Services*, pages 1–9, 2005.

[20] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop*, pages 139–152, 2006.

[21] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, 2003.

[22] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop*, pages 82–96, 2001.

[23] Bruno Blanchet. From secrecy to authenticity in security protocols. In *International Static Analysis Symposium*, pages 342–359, 2002.

[24] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

[25] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *International Cryptology Conference*, pages 537–554, 2006.

[26] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, and Christopher Walstad. Breaking and fixing public-key kerberos. *Information and Computation*, 206(2-4):402–424, 2008.

[27] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[28] Luca Durante, Riccardo Sisto, and Adriano Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology*, 12(2):222–284, 2003.

[29] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Selected Areas in Cryptography*, pages 1–24, 2001.

[30] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[31] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation*, pages 363–379, 2005.

[32] Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. Developing provable secure M-commerce applications. In *Emerging Trends in Information and Communication Security*, pages 115–129, 2006.

[33] Holger Grandy, Kurt Stenzel, and Wolfgang Reif. Refinement of security protocol data types to Java. In *Program Analysis for Security and Safety Workshop Discussion*, 2006.

[34] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[35] Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing*, pages 213–226, 2003.

[36] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

[37] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 311(1-3):121–163, 2004.

[38] Chul-Wuk Jeon, Il-Gon Kim, and Jin-Young Choi. Automatic generation of the C# code for security protocols verified with casper/FDR. In *International Conference on Advanced Information Networking and Applications*, pages 507–510, 2005.

[39] S. P. Joglekar and S. R. Tate. ProtoMon: Embedded monitors for cryptographic protocol intrusion detection and prevention. *Journal of Universal Computer Science*, 11(1):83–103, 2005.

[40] Jan Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *International Conference on Formal Methods and Models for Co-Design*, pages 89–98, 2005.

[41] Jan Jürjens and Mark Yampolskiy. Code security analysis with assertions. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 392–395, 2005.

[42] Shinsaku Kiyomoto, Haruki Ota, and Toshiaki Tanaka. A security protocol compiler generating C source codes. In *International Conference on Information Security and Assurance*, pages 20–25, 2008.

[43] E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. *Electronic Notes in Theoretical Computer Science*, 155:583–603, 2006.

[44] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM Special Interest Group on Software Engineering Notes*, 31(3):1–38, 2006.

[45] V. B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, pages 18–18, 2005.

[46] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[47] Gavin Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop*, pages 31–43, 1997.

[48] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[49] Catherine A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[50] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[51] Anthony Nadalin, Chris Kaler, P. Hallam-Baker, and R. Monzillo. OASIS web services security: SOAP message security 1.1 (WS-security 2004), 2006.

[52] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[53] Alfredo Pironti and Jan Jürjens. Formally-based black-box monitoring of security protocols. In *International Symposium on Engineering Secure Software and Systems*, pages 79–95, 2010.

[54] Alfredo Pironti and Jan Jürjens. Online resources about black-box monitoring of legacy implementations. http://alfredo.pironti.eu/research/projects/monitoring, 2010.

[55] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. Spi2Java home page. http://spi2java.polito.it/.

[56] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *IEEE Symposium on Computers and Communications*, pages 839–844, 2007.

[57] Alfredo Pironti and Riccardo Sisto. Formally sound refinement of Spi Calculus protocol specifications into Java code. In *IEEE High Assurance Systems Engineering Symposium*, pages 241–250, 2008.

[58] Alfredo Pironti and Riccardo Sisto. Soundness conditions for cryptographic algorithms and parameters abstractions in formal security protocol models. In *International Conference on Dependability of Computer Systems*, pages 31–38, 2008.

[59] Alfredo Pironti and Riccardo Sisto. Soundness conditions for message encoding abstractions in formal security protocol models. In *Availability, Reliability and Security*, pages 72–79, 2008.

[60] Alfredo Pironti and Riccardo Sisto. Provably correct Java implementations of Spi Calculus security protocols specifications. *Computers & Security – Elsevier*, 29(3):302–314, 2010.

[61] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol Java code generation from Spi Calculus. In *International Conference on Advanced Information Networking and Applications*, pages 400–405, 2004.

[62] Fulvio Risso and Mario Baldi. NetPDL: an extensible XML-based language for packet header description. *Computer Networks*, 50(5):688–706, 2006.

[63] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[64] Usa Sammapun, Raman Sharykin, Margaret DeLap, Myong Kim, and Steve Zdancewic. Formalizing Java-MaC. *Electronic Notes in Theoretical Computer Science*, 89(2):171–190, 2003.

[65] Gerhard Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–435, 2005.

[66] Steve Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

[67] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Transactions on Information and System Security*, 7(2):319–332, 2004.

[68] Martin Süßkraut and Christof Fetzer. Robustness and security hardening of COTS software libraries. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 61–71, 2007.

[69] D. Syme. F#. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/default.aspx.

[70] Benjamin Tobler and Andrew Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Services*, 2004.

[71] Luca Viganò. Automated security protocol analysis with the AVISPA tool. *Electronic Notes on Theoretical Computer Science*, 155:61–86, 2006.

[72] Victor L. Voydock and Stephen T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, 1983.

[73] Song Dawn Xiaodong, Wagner David, and Tian Xuqing. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, pages 25–25, 2001.

[74] Song Dawn Xiaodong, Adrian Perrig, and Doantam Phan. AGVI - automatic generation, verification, and implementation of security protocols. In *International Conference on Computer Aided Verification*, pages 241–245, 2001.

[75] Li Yafen, Yang Wuu, and Huang Ching-Wei. Preventing type flaw attacks on security protocols with a simplified tagging scheme. In *International Symposium on Information and Communication Technologies*, pages 244–249, 2004.

[76] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[77] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.