A STRUCTURED APPROACH TO NETWORK SECURITY PROTOCOL IMPLEMENTATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Benjamin Tobler
November 2005

Supervised by
Prof. Andrew C.M. Hutchison

Abstract

The implementation of network security protocols has not received the same level of attention in the literature as their analysis. Security protocol analysis has successfully used inference logics, like GNY and BAN, and attack analysis, employing state space examination techniques such as model checking and strand spaces, to verify security protocols. Tools, such as the multi-dimensional analysis environment SPEAR II, exist to help automate security protocol specification and verification, however actual implementation of the specification in executable code is a task still largely left to human programmers. Many vulnerabilities have been found in implementations of security protocols such as SSL, PPTP and RADIUS that are incorporated into widely used operating system software, web servers and other network aware applications. While some of these vulnerabilities may be a result of flawed or unclear specifications, many are the result of the failure of programmers to correctly interpret and implement them.

The above indicates a gap between security protocol specifications and their concrete implementations, in that there are methodologies and tools that have been established for developing the former, but not the latter. This dissertation proposes an approach to bridging this gap, describes our implementation of that approach and attempts to evaluate its success.

The approach is three-fold, providing different measures to improve current ad-hoc implementation approaches:

1. **From Informal to Formal Specifications:** If a security protocol has been specified using informal *standard notation*, it can be converted, using automatic translation, to a *formal specification language* with well defined semantics. The formal protocol specification can then be analysed using formal techniques, to verify that the desired security properties hold. The precise specification of the protocol behaviour further serves to facilitate the concrete implementation of the protocol in code.

2. **Separate Implementation Concerns:** When implementing security protocols, the *what* and the *when* of protocol actions are abstracted from the *how*. That is, protocol logic implementation concerns, such as when and what actions should be performed on messages, should be clearly and cleanly separated from the cryptographic and network communication implementation details that implement how the actions are performed. Such high level modularity allows code implementing protocol logic to be re-used with different cryptographic algorithm implementations and network communication protocols. It also allows errors in the implementation of the cryptography to be addressed by swapping cryptographic implementations without changing the protocol logic code. The abstraction of cryptographic and network implementation is analogous to the adoption of the Dolev-Yao style models by many analysis techniques, where the cryptography itself is viewed as a black box and assumed perfect, allowing the analysis to focus on the protocol logic. Finally, this separation allows the correctness of the protocol logic implementation and cryptographic primitives implementation to be addressed separately.

3. **Automated Implementation Using Code Generation** We use code generation to automate the security protocol implementation process, avoiding the risk of human error in interpreting the sometimes subtle semantics of security protocol specifications. The precise nature of formal specification languages provides a base from which to specify and implement an automatic code generation tool. Our approach follows requirements identified for high integrity code generation - where feasible - to give a high level of confidence in the correctness of the generated code.

In implementing the approach, we adopt the *Spi Calculus* for the role of formal specification language. The Spi Calculus was developed by extending the π-calculus, a process algebra for describing concurrent communicating systems, to cater for the special case of network security protocols. Spi Calculus specifications can be analysed manually, by developing correctness proofs by hand, and automatically, by using model checkers such as MMC. As Spi Calculus specifications explicitly describe the actions of a security protocol, they are also particularly suitable for use as input for code generation. The implementation of the approach is split across three components that correspond to each of the parts of the approach:

1. **Sn2Spi** is a translator that converts an informal standard notation specification to a Spi Calculus specification, thus implementing part 1 of our approach. The converted specification can be analysed using any of the formal techniques applicable to the Spi Calculus. Once verified, the specification can be used to generate a concrete implementation using Spi2Java.

2. **The Security Protocol Primitives API** abstracts cryptographic and network communication operations, decoupling code that implements protocol logic from code that implements

cryptographic and network operations. It provides the basic cryptographic and network communications functionality required to implement a security protocol, including: symmetric and asymmetric encryption, message digest, nonce and timestamp generation, marshalling message component data and sending and receiving messages over a network. A provider model, much like that used in the Java Cryptography Extensions API, is employed to allow different implementations to be swapped without changing the SPP client code.

3. **Spi2Java** is a code generator, essentially implementing a compiler from the Spi Calculus to Java code. Spi2Java uses Prolog to implement a defined mapping from Spi Calculus constructs, i.e. terms and process actions, to Java code segments. These code segments call the SPP API to access cryptographic and network functionality where needed. The mapping was developed by refining Spi constructs to Java code segments that preserve the semantics of the Spi constructs. In addition, assertions are made in the code segments to ensure certain conditions are met before the implementation can continue running.

Part of evaluating the effectiveness of this automated approach to security protocol implementation, involved a case study where manual implementations of the CCITT Three Message X.509 Protocol, developed by 4th year Computer Science students, and a Spi2Java generated implementation are compared. The outcome of the study favoured the automatically generated implementation, indicating the potential of the approach.

Further to demonstrating the utility of code generation, we describe an SPP provider implementation developed to allow a security protocol run, including legitimate and attacker roles, to be simulated in a controlled environment. Spi2Java allows the protocol engineer to quickly and automatically generate code for protocol roles. The code can be executed using this implementation allowing the protocol engineer to step through execution of all roles, both legitimate and attacker, to gain insight into the behaviour of the protocol.

The approach is evaluated in terms of the class of attacks it prevents and how it meets the identified requirements for high integrity code generation. It is also compared to existing and current work in the field. Attack classes that exploit faulty protocol logic implementation, vulnerability to type flaws and buffer overflows are prevented. The Spi2Java code generator fully meets three of the five high integrity code generation requirements: formally defined source and target languages are used; the translation software is validated; and the generated code is well structured and documented and can be traced back to the specification. Spi2Java partially meets the requirement that the mapping from source to target language constructs be formally proven to preserve the specification semantics. However the arguments given are not strictly formal. The requirement related to rigorous testing are not met due to practical resource constraints. However, Spi2Java has been used to generate real world protocol implementations that have been verified by manual inspection.

Sprite, incorporating the Sn2Spi translator and Spi2Java code generator, provides a structured approach to network security protocol implementation by implementing automated translation from informal to formal security protocol specifications, and by being able to automatically generate Java implementations of network security protocols in which the security protocol engineer can have a high degree of confidence.

ACKNOWLEDGEMENTS

Firstly I would like to thank my supervisor Prof. Andrew Hutchison, for guidance, patience and constructive criticism. I also wish to thank the other staff of the Data Networks Architectures Lab for providing a constructive and enjoyable working environment. Thanks to my fellow students in the lab for their friendship, discussions and frequent coffee breaks. Finally, and most importantly, thanks to my family and friends whose support has made this possible in the first place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network security protocols provide a mechanism to communicate securely over public networks, such as the internet, facilitating electronic commerce, personal and business interactions and other communications and transactions that require some level of security. The aim of a security protocol is to provide the required combination of the general security properties: *Authentication*, *Confidentiality* and *Integrity* or their specialisations: *Authorisation* and *Non-repudiation*.

Security protocols specify rules to achieve secure communication. These rules describe a sequence of messages - the *message flow*, including content and format of the messages and cryptographic operations, that when followed successfully result in some security outcome, e.g. the authentication of a *principal* (a legitimate party in a security protocol) or the establishment of a channel allowing private communication between multiple principals. Because cryptography is used to achieve security, the terms *security protocol* and *cryptographic protocol* are often used interchangeably. We shall use the term "security protocol", or sometimes just "protocol" when the context is clear, throughout this dissertation.

Research in *security protocol engineering* has focused primarily on the analysis of security protocol specifications for correctness: that is determining whether or not the intended security properties of a protocol actually do hold. In the next section we elaborate on this, and develop the motivation for this thesis, which aims to address the fact that there has been little research into the correct implementation of security protocols.

1.1 Motivation

The purpose of security protocol analysis is to verify that the intended security properties of the protocol hold and cannot be subverted by either an active or passive attack on the protocol. Formal and semi-formal methods have been successfully used in security protocol analysis [41, 37, 39, 1, 61]:

- *Static analysis* using inference logics, such as BAN [41] and GNY [37], allow the protocol engineer to reason about the beliefs of the principals participating in a run of a security protocol. Tools such as SPEAR II [55], provide a user-friendly graphical environment for security protocol engineering and analysis. They allow the engineer to specify a security protocol and the intended possessions and beliefs that the principals should have at the end of a successful protocol run. The automatic analysis capability of SPEAR II uses the initial possessions and beliefs of the participants as given facts which, in conjunction with the inference rules of GNY, allow it to determine what valid beliefs the participants may hold at the end of the run.
- *Dynamic analysis*, using techniques such as model checking and strand space analysis to explore and reason about the state space of possible protocol runs, has revealed ways to actively subvert a protocol, such as man in the middle and parallel session style attacks. Tools like Casper/FDR [40], CAPSL/Maude [20] and MMC [54] implement model checking techniques by analyzing security protocols, specified in a suitable formal language, to determine whether certain properties hold. This approach has been particularly successful; revealing flaws, such as discovered by Lowe in the Needham Schroeder protocol [40], in protocols that for years were assumed to be secure.

The same emphasis has not, however, been placed on the correctness of security protocol *implementation*. Although methodologies and tools, such as those mentioned above, exist to help design and analyze security protocols, the concrete implementation in code is a task still largely left to human programmers. As with the implementation of concurrent communicating systems in general, there is significant scope for introduction of errors by human programmers. This is compounded by the fact that the semantics of security protocol specifications are often subtle, small deviations in the implementation from the specification can dramatically effect the actual security properties of the protocol.

The potential for implementation error is evident in the number of security alerts issued for implementations of various security protocols used by software such as web servers, web browsers and operating system components relied on by network aware applications. Flaws have been discovered in many software vendors' SSL implementations in the last year alone, including, but not limited

to, companies such as Apple, SCO, Microsoft, Cisco, and RSA, as well as open source organisations such as OpenSSL, KDE and Apache [11, 13, 36, 12].

It is clear then, that although security protocol engineering has been successful in producing correct specifications, it has been less so in producing correct implementations. This is arguably a general software development problem, but because it is of special significance to security software components, this discourse is limited to security protocols.

Our survey of the literature, when beginning this work, revealed few well defined methods, and - at the time - no available tools for implementing security protocols, indicating a gap between security protocol specification and implementation. This gap provides the motivation for this work.

1.2 Objective and Requirements

The aim of this work is to bridge the gap between security protocol specification and implementation. We define a structured approach to security protocol implementation, involving software tools for automated translation of specifications and code generation.

To achieve the objective, we define it in terms of the following requirements that our work must meet:

1. It must have the ability to automatically translate informal to formal specifications,
2. be able to automatically generate security protocol implementations from formal specifications and provide a high level of confidence in those implementations and
3. must realise a well defined methodology and tools for security protocol implementation, which are easily usable by the security protocol engineer.

1.3 Evaluation Criteria

To evaluate our approach, and specifically its software components, we need to determine to what extent it meets the requirements listed above.

Whether our work meets the first and third requirements can be judged directly. However to evaluate the work against the second, and key, requirement, an independently defined set of criteria is necessary. We will use the requirements for high integrity code generation, identified in [64], to

evaluate our work with respect to meeting the second requirement. We introduce, summarise and discuss these requirements for high integrity code generation in chapter 2.3.

1.4 Scope and Limitations

Research by Backes et al [44] has shown that concrete cryptographic libraries can correctly implement an idealised cryptographic model, such as that proposed by Dolev and Yao in [24]. We thus limit the scope of this work by not addressing the issues associated with correctly implementing cryptographic primitives. Instead we will focus on the correctness of the generated code that implements the protocol logic, i.e. the code that determines *when* and *what* actions are performed by the protocol implementation.

Similarly, we do not address the correctness of the implementation of network communication operations. We do, however, use widely adopted and tested cryptographic and network libraries to implement these functions.

1.5 Approach

To achieve the objective and meet the requirements, we propose a high level approach in which the following are advocated:

- **Formal Specification:** Use a suitable formal language for specifying security protocols. This language must be suitable for both protocol analysis techniques and as a basis for implementation, to avoid translation between multiple specification languages.
- **Automated Specification Translation:** Automatically translate any informal specifications in standard notation to the adopted formal language.
- **Separate Implementation Aspects:** Separate the code that implements the protocol actions from the code that implements the cryptographic algorithms and communications operations.
- **Automated Implementation:** Automatic code generation that generates an executable implementation from a formal specification.

In implementing this approach we use the *Spi Calculus* as the formal specification language; develop a translation utility *Sn2Spi* to convert informal standard notation specifications to Spi Calculus

specification; define and implement the *Security Protocol Primitives API* to allow cryptographic primitives and network communication operations to be de-coupled the protocol logic implementation; and implement *Spi2Java*, an automated code generation tool that generates Java code from Spi Calculus specifications.

1.6 Dissertation Outline

This layout of this disseration is as follows:

- In chapter 2 we introduce and provide some background on security protocol engineering, introduce and evaluate specification languages that can be used in that process, and look at requirements for high integrity code generation.
- Chapter 3 introduces the formal specification language we will use in this work, the Spi Calculus, and describes some modifications we have made to it to make it suitable for specifying input for a code generation tool.
- The Sn2Spi tool, developed for translating informal standard notation security protocol specifications to the Spi Calculus, is described in chapter 4.
- In chapter 5 the design and development of the Security Protocol Primitives API for abstracting cryptographic and network communications functionality is discussed and its interface defined.
- In chapter 6 we describe the core of this work, Spi2Java, a tool for generating Java code from security protocol specifications in the Spi Calculus. We discuss the development of the translation from Spi to Java, and the implementation of that translation in Prolog. We generate an implementation of the Needham-Schroeder-Lowe protocol to demonstrate Spi2Java.
- We briefly cover the Security Protocol Implementation Tool and Environment (Sprite) that provides a graphical user interface to Sn2Spi and Spi2Java, and a SPP API provider implementation that allows the simulation of security protocol runs using Spi2Java generated code, in chapter 7.
- In chapter 8, we described a case study comparing manual implementations of simplified version of the CCITT Three Message X.509 Protocol against an automatically generated implementation using Sn2Spi and Spi2Java. We also evaluate Spi2Java against the high integrity code generation requirements described in chapter 2.

- We conclude with chapter 9, by looking at how this work meets the objectives listed in this introduction chapter, discussing limitations and possible future work and describing the contributions of this work.

Chapter 2

Background

In the first section of this background chapter we discuss security protocol engineering and place this work in that context. The second section is a survey of security protocol specification languages. We look at the role of specification languages in security protocol engineering and develop a set of criteria to determine their suitability for this purpose. We summarise several key languages and evaluate them against these criteria. Finally we introduce high integrity code generation, as defined by the requirements formulated by Whalen and Heimdahl [64], and verifiable compilers.

2.1 Security Protocol Engineering

Security protocol engineering is a specialisation of software engineering in general, with its concept of a *system development life cycle* (SDLC), such as those described in [33], involving *requirements analysis*, *design*, *development* and *testing*. In this section we describe a *security protocol development process* similar to the traditional SDLC model, but with some modifications. We then place this work in the context of this process and relative to other research, analysis techniques and tools.

We define a security protocol development process, outlined in figure 1, consisting of *requirements analysis*, *design and specification*, *implementation* and *implementation verification* phases. The initial *software concept* phase of the traditional SDLC, that identifies the need for a new system, is skipped.

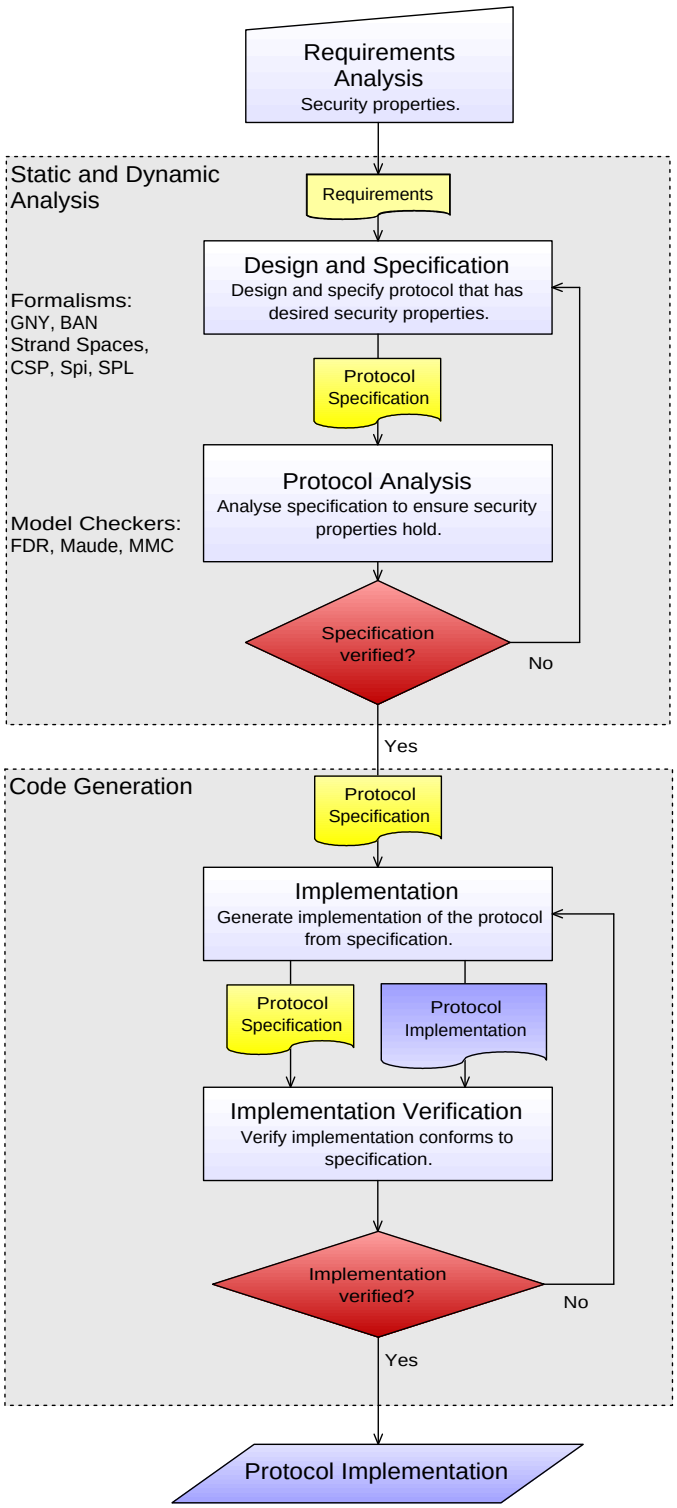

Figure 1: Overview of the security protocol development process.

2.1.1 Requirements Analysis

This first phase is analogous to the *requirements analysis* phase of the traditional SDLC model. In this process however, the requirements of interest are specifically those related to security, i.e. the desired security properties of the protocol. The requirements are specified in terms of the security properties: *authentication*, *confidentiality* and/or *integrity*. Note that restricting requirements to security properties in this discussion is not intended to deny that, in software development, a holistic approach should be taken to security. Security is a totally pervasive aspect of software systems, and cannot be addressed in isolation or (easily) implemented retro-actively.

2.1.2 Design and Specification

This phase, and the following *protocol analysis* phase, correspond to the design phase of the SDLC - in which a software system that meets the requirements, is designed both logically and physically. In this phase a security protocol specification, that attempts to meet the requirements identified in the first phase, is developed. The specification defines the protocol's

- message flow,
- message contents
- and the cryptographic mechanisms employed (not necessarily the specific algorithms to be used, but rather the type of operations, e.g. symmetric or asymmetric encryption and decryption, message digests etc.)

These are determined with the intention of endowing the security protocol specification with the desired security properties. Security protocol flaws are often subtle, and while there are guidelines for designing security protocols [2], there is no canonical list of rules that can be followed to guarantee a flawless protocol design. However, there are successful analysis techniques for verifying, or revealing flaws in, security protocol specifications. Thus this phase is coupled with the *protocol analysis* phase in a feedback loop; a security protocol specification is developed, then analysed, if flaws are found the engineer returns to the *design and specification* phase and repeats the process until a correct protocol specification is developed.

2.1.3 Protocol Analysis

The protocol specification is analysed to determine whether the security properties stated in the requirements hold. If they don't, the analysis results are fed back to the design and specification phase and the protocol is modified or redesigned. If they do then the process can progress to the implementation phase.

The output of this phase is a security protocol specification that has been verified to be correct w.r.t. the requirements formulated in the first phase.

2.1.4 Implementation

In this phase, which corresponds to the development phase of the SDLC, the protocol specification is implemented, either manually or by automatic code generation.

2.1.5 Implementation Verification

To avoid negating the work done verifying the protocol specification, the implementation must be verified to ensure it correctly implements the specification. The *implementation verification* phase corresponds to the testing phase of the SDLC. Unlike traditional "after the fact" testing of the SDLC, the use of automatic code generation allows the *implementation verification* to be integrated into the *implementation* phase. In section 2.3 we discuss how an automatic code generation tool can be developed to meet requirements for high integrity code generation, ensuring the correctness of the generated code. In this approach, the translation from the specification to code, and the implementation of that translation by the code generator, are verified, obviating the need to verify every protocol implementation generated by the automatic code generator.

The output of this phase is the output of the entire process: an executable program that conforms to the security protocol specification that meets the identified requirements.

2.1.6 This Work in Context

The work described in this dissertation focuses on the *implementation* and *implementation verification* phases and, to a lesser extent, relates to the *design and specification* and *protocol analysis* phases.

Our Spi2Java automatic code generation tool, introduced in chapter 6, forms the core of this work. It provides an automated mechanism for the *implementation* and *implementation verification* phases, as outlined in figure 1.

The Sn2Spi translation tool, discussed in chapter 4, converts informal specifications to formal specifications. It facilitates linking the *design and specification* phase to the *protocol analysis* phase by allowing informal specifications, that might be developed during the *design and specification* phase, to be translated to formal specifications that can be used more effectively in the *protocol analysis* phase.

2.2 Security Protocol Languages

The purpose of this section is to examine the role of specification languages in the security protocol development process described previously. We review informal security protocol specification using the *standard notation*, discuss why formal specification is important, show how a specification language can facilitate the analysis and implementation phases of security protocol engineering and then identify the properties that make a specification language suitable for these phases. With these properties in mind, we survey a number of existing languages for security protocols. Finally we discuss our selection of one these languages for use in this work.

2.2.1 Informal Specification Using the Standard Notation

Texts describing security protocols generally use an informal *standard notation*. The term "standard" may be misleading as there are many minor syntactic variations. The standard notation specifies security protocols at a high level of abstraction: indicating the order, direction of flow, and contents of the protocol messages, but not the format of the message components or the exact cryptographic algorithms used. A simple protocol, where principal A sends principal B a message consisting of her identifier and a nonce, all encrypted with B's public key, would be specified as:

1 $A \rightarrow B : \{n, A\}_{K_{PubB}}$

2.2.2 The Need for Formal Specification Languages

While intuitive, the notation above is informal. It does not explicitly state the actions necessary to verify messages received by the principals, nor the meaning of the messages' contents. The notation is not precise enough for basing a formal analysis on, developing implementations from, or verifying implementations against. Because of these shortcomings, formal languages have been developed for specifying and analysing security protocols.

2.2.3 Specification Languages in Security Protocol Engineering

For this work, the primary role of a specification language is to specify input for a code generation tool. However the language should have properties suitable for protocol analysis and verifying implementations against their specifications. More generally the specification language should be suitable for all the security protocol engineering activities to link them together, as shown in figure 2.

Design and Specification

For specification, a language needs to be intuitive as well as easily usable and understandable by the security protocol engineer. It must also be able to precisely and unambiguously specify the behaviour of a security protocol. Natural language definitions of protocol semantics may be intuitive, but due to the inherent ambiguities of natural languages they may not be sufficiently precise [64]. Formally defined semantics can provide the required precision.

Protocol Analysis

Formal languages have been used to reason about the security properties of protocols [16, 1, 40, 66, 61, 9]. A mathematically sound basis for analysis makes it possible to construct proofs showing that a desired security property always holds, or produce counter-examples to demonstrate that the desired security property does not hold, as in [40]. This can be done manually, or with model checkers and analysis tools such as FDR, Isabelle/HOL, the NRL Protocol Analyzer and Maude [40, 53, 20, 9].

Implementation

Correctly implementing a security protocol, either manually or automatically, requires an exact specification of the protocol behaviour. As mentioned, natural language definitions may not be sufficiently precise, so again formally defined semantics are required. To facilitate code generation, the language's concrete syntax should be defined to allow specification in plain text.

Implementation Verification

Even a small flaw in a security protocol implementation can result in the implementation being insecure. The problem is compounded by the fact that implementation is often an informal process and hence error prone. This makes it easier for inconsistencies between the specification and implementation to arise.

The formal definition of a language's semantics, provide a way to refine the specification to executable code and prove that it correctly implements that specification [64].

2.2.4 Specification Language Requirements

From the discussion above we have identified the following requirements for the selection of a specification language:

1. *Usable*: The language must be easily usable. It should have a concise and simple abstract and concrete syntax. An intuitive, natural language definition of the language semantics should be defined.

2. *Formal*: The language must have a formally defined syntax and semantics.

3. *Suitable for Analysis*: The semantics of the language should be suitable for a formal analysis of a protocol's security properties.

4. *Suitable for Verification*: The semantics of the language should be suitable for verifying implementations against specifications.

5. *Plain Text Specifications*: Specifications must be able to be written as ASCII text files for input into automated tools.

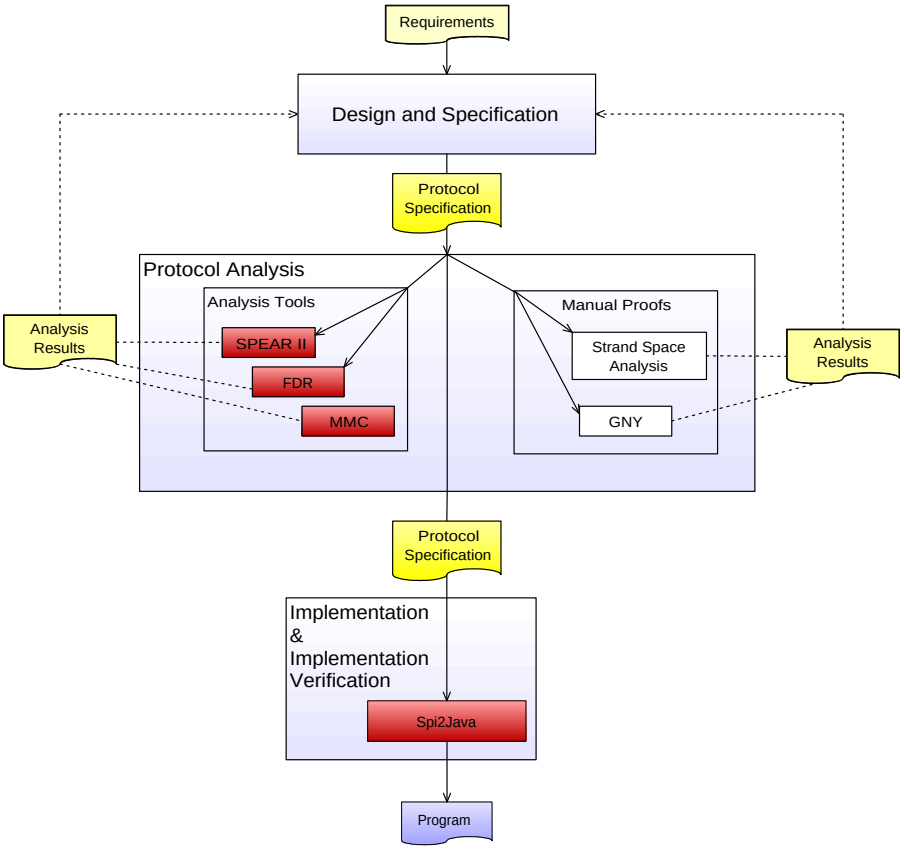

Figure 2: The use of a specification during analysis, implementation and verification.

2.2.5 Survey of Security Protocol Languages

In this section we review several potentially suitable security protocol specification languages. They are CAPSL/CIL, CPAL and the process algebras SPL and the Spi Calculus.

CAPSL and CIL

The Common Authentication Protocol Specification Language (CAPSL) was developed as a single language to specify input to a wide range of formal analysis methods and automated tools [22]. It is intuitive, yet formally defined. Its core syntax is similar to the standard notation and specifications are in plain ASCII text. CAPSL specifications have a preamble to define initial assumptions and protocol goals to be [21].

Its formal semantics are defined by its translation to the CAPSL Intermediate Language (CIL) [22], a rewrite logic that describes the state transitions of a protocol [21]. This semantics definition is similar to the internal representations of protocols used by many tools [20], facilitating translation to the other languages used by these tools.

CAPSL/CIL meets most of the identified requirements. However, there are some usability drawbacks of the combination, especially with regard to simplicity and conciseness. CAPSL has to be translated to CIL to provide a formal specification. Doing this manually is a long and tedious task - even for a simple protocol: the CIL definition of the Needham-Schroeder Public Key Authentication Protocol, which involves the exchange of three messages, runs to approximately 176 lines [48]. A preliminary automatic translator does exist, but is described as experimental and incomplete [49].

CPAL

The Cryptographic Protocol Analysis Language (CPAL) [66], makes protocol specification explicit by, for instance, allowing assertions to be made about received messages components, having explicit encryption and decryption operators and conditional statements such as “if-then-else” constructs.

The language semantics are defined by Dijkstra’s weakest precondition logic and is based on the pre-condition/post-condition approach to reasoning about programs developed by Hoare [66]. This gives CPAL the formality required to analyse security protocols, as demonstrated by the CPAL Evaluation System (CPAL-ES) described by Yasinsac in [66]. CPAL-ES provides a method for both static and dynamic analysis and has been used to evaluate a number of protocols, including the Woo-Lam, Andrews Secure RPC and Needham-Schroeder Public Key protocols [66]. The use of

CPAL appears to be quite limited.

Process Algebras: Spi Calculus and SPL

Process languages, or algebras, have been developed to facilitate the formal description and analysis of complex, communicating systems. As security protocols are a special case of communicating systems, process languages, e.g. CSP (Communicating Sequential Processes) and the π-calculus, can be used to reason about their properties [40] [1]. The discovery of a flaw in the Needham-Schroeder public key authentication protocol, cited earlier, is an example of the effectiveness of process languages in describing and analyzing security protocols [40].

The Spi Calculus, defined by Abadi and Gordon in [1], extends the π-calculus by adding encryption and decryption clauses to the syntax. SPL is similar to the Spi Calculus [16], but defines fewer, more specialised, processes. SPL uses a network communications model that is quite different from the channels used by the Spi Calculus.

The Spi Calculus, consists of terms, such as names, variables and pairs, and some simple processes, such as sending and receiving messages and parallel composition of processes. Each of the processes has a simple, well defined function. Despite its small size and relative simplicity, the Spi Calculus (and the π-calculus it is based on) is very powerful in its ability to describe concurrent systems.

We introduce the Spi Calculus by way of an example - in which we compare an informal to a formal specification. A full definition of the Spi Calculus is given in chapter 3 when we introduce it as our formalism of choice for specifying security protocols.

SPL is a process language similar to the Spi Calculus. SPL has a concise syntax, defining only three processes. These processes have more specialised and complex behaviours compared to those of the Spi Calculus. SPL uses a message space, as opposed to channels, to model network communication. Messages are sent by processes to the message space that represents the network. They can be read by any other process, as described in [15]. SPL is asynchronous, in that an SPL output process does not have to wait for a, or synchronise with, a corresponding input process to interact with in order to send a message [15].

Assuming a set of infinite names with elements n, m, A, ..., variables x, y, ..., X, Y ... over names, variables ψ, ψ', ψ_1, ... the syntax for SPL is defined, as it appears in [15], by the grammar:

Name expression	c	::=	n, A, ... | x, X, ...
Key expression	k	::=	$Pub(v) \mid Priv(v) \mid Key(\vec{v})$

Message expression	m	::=	$v \mid k \mid M_1,\, M_2 \mid \{M\}_k \mid \psi$
Process expression	p	::=	$out\, new \vec{x} M.p \mid in\, pat \vec{x} \vec{\psi} M.p \mid \; \|_{i \in I}\; p_i$

The vector notation, $\vec{x}$, is used to indicate a list of zero or more elements.

The behaviour of the processes is informally described, as in [15], as follows:

- $out\, new\, \vec{x} M.p$ selects fresh distinct names (generally nonces or keys), $\vec{n}$ and binds them to the element of $\vec{x}$. It then outputs the message $M[\vec{n}/\vec{x}]$ to the network. The process p is then run.
- $in\, pat\, \vec{x}\vec{\psi} M.p$ waits for a message to be input that matches the message pattern M for a valid binding to the pattern variables $\vec{x}\vec{\psi}$ that appear in M. The process p is then run.
- $\|_{i \in I}\; p_i$ is the parallel composition process and behaves as all the components, which are indexed by i, running in parallel. The empty parallel composition process is abbreviated as nil, and does nothing.

Formal vs. Informal Specifications

We use the canonical Needham-Schroeder Public Key Authentication Protocol as an example to contrast informal and formal specifications. The protocol is defined informally in the standard notation by message flow:

1 $A \rightarrow B : \{n, A\}_{K_{PubB}}$
2 $B \rightarrow A : \{n, m\}_{K_{PubA}}$
3 $A \rightarrow B : \{m\}_{K_{PubB}}$

To define the same protocol formally in the Spi Calculus, the initiator and responder roles are first defined as processes. If A and B are terms representing the principals participating in the protocol, N and M are nonces, K_{PubA} and K_{PrivA} are the public and private keys of A, K_{PubB} and K_{PrivB} are the public and private keys of B and C_{AB} is an insecure channel suitable for communication between A and B, then the initiator can be defined as:

$$
\begin{aligned}
Init(A,\, K_{PubB},\, K_{PrivA}) \quad = \quad & (N)\overline{C_{AB}}\langle \{N,\, A\}_{K_{PubB}} \rangle. \\
& C_{AB}(x_{cipher}).case\, x_{cipher}\, of\, \{x_1,\, x_2\}_{K_{PrivA}}.[x_1\, is\, N].
\end{aligned}
$$

$$\overline{C_{AB}}\langle\{x_2\}_{K_{PubB}}\rangle.$$
$$F(X)$$

and the responder as:

$$\begin{aligned} Resp(K_{PubA},\, K_{PrivB}) \quad &= \quad C_{AB}(x_{cipher}).case\, x_{cipher}\, of\, \{x_1,\, x_2\}_{K_{PrivB}}. \\ &\quad (M)\overline{C_{AB}}\langle\{y_1,\, M\}_{K_{PubA}}\rangle. \\ &\quad C_{AB}(y_{cipher}).case\, y_{cipher}\, of\, \{y\}_{K_{PrivB}}.[y\, is\, M]. \\ &\quad G(X) \end{aligned}$$

An instance of the protocol is defined as follows:

$$\begin{aligned} &NS(A,\, K_{PubA},\, K_{PrivA},\, K_{PubB},\, K_{PrivB}) = \\ &\quad Init(A,\, K_{PubB},\, K_{PrivA})\,|\,Resp(K_{PubA},\, K_{PrivB}) \end{aligned}$$

Although it is clear from the informal description what the contents of the messages should be, the actions of the protocol are not explicit. For example: the informal description does not explicitly state that the initiator, A, must check the nonce n received in message 2 against the nonce it sent in message 1. It is also not clear when the nonces should be generated and what action to take if the value of a received message component does not match the expected value. Though these actions may be regarded as intuitive, this may not be the case for a more complex protocol that has a greater number of messages and message components.

In contrast, the formal definition of the same protocol describes precisely the actions that should be taken when sending and receiving messages. Returning to the example, the formal definition specifies that the initiator must check that the nonce y_1 received in message 2 matches the nonce n sent in message 1. It also specifies when it should be checked and what to do (in this case halt) if it does not match.

So although the standard notation lends itself to a quick, intuitive understanding of a protocol, the Spi Calculus specification is far more precise. Where the former leaves many subtler aspects of the protocol to implication, the latter clearly specifies exactly how and when such aspects as nonce instantiation and message component matching and verification, should be performed.

2.2.6 Evaluation of Specification Languages

Table 1 summarises the specification languages discussed above, against the requirements we have identified. By these criteria the Spi Calculus, SPL and CAPSL all appear to be suitable candidates. We exclude CAPSL because it needs to be translated to CIL, which has the disadvantages discussed previously. CPAL does not seem to have been used widely outside of the projects involving its developers.

Our final choice, the Spi Calculus, was chosen over SPL, as its process have simpler, more explicit and contained behaviour. Like SPL the Spi Calculus can be used to manually reason about a security protocol, but it also has been used in conjunction with automatic model checkers [54]. Also, at the time this work began, there were no code generation projects for Spi, while there was such a project underway for SPL [42].

	CAPSL/CIL	CPAL	Spi Calculus	SPL	Standard Notation
Usable	Yes/No	Yes	Yes	Yes	Yes
Formal	No/Yes	Yes	Yes	Yes	No
Suitable for Analysis	No/Yes	Yes	Yes	Yes	No
Suitable for Code Generation	No/Yes	Yes	Yes	Yes	No
Plain Text Specification	Yes/Yes	Yes	Yes[1]	Yes[1]	Yes
Used in Multiple Projects	Yes	No	Yes	Yes	Yes

Table 1: Existing security protocol languages against identified requirements.

2.3 High Integrity Code Generation

Code generation automates the process of translation from a (generally) higher level source language to a (generally) lower level target language. A compiler is a specific instance of a code generator, translating what is essentially a high level specification in a programming language such as C, to a concrete implementation in executable machine code. In this work the type of code generation we are interested in is the translation from fairly abstract high level specifications, to implementation programs that can be compiled and executed.

Being an automated, mechanical process, code generation is more efficient, faster and less error prone than manual implementation by human programmers. There is a caveat: an error in the code generator will generate incorrect code more efficiently, faster and more regularly than those human programmers. However, if a code generator is correct, or has properties that instill a high level of confidence, that confidence can be carried over to all its output.

[1] With minor syntax modifications.

In this section we summarise and discuss the requirements for "high integrity code generation" identified by Whalen and Heimdahl in [64]. We then examine how adopting some of the aspects of the approach of Hoare et al in developing verifiable compiling specification and prototype compiler [10] meets some of these requirements.

2.3.1 Code Generation Requirements

Whalen and Heimdahl develop their requirements for code generation using a formal basis. In order to reason about the properties of the generated code, it is necessary for the semantics of both the source and target languages to be precisely (i.e. formally) defined. The translation process can then be viewed as transformation function, much like the compiling relation between a source program and its object code described by Hoare in [10]. Having this formal relation between the source and target languages allows us to reason about them with the goal of showing that the generated program in the target language is equivalent, or a refinements of, the specification in the source language. This is the rationale behind the first and second of their requirements listed in [64, pages 2, 3 and 4]:

1. "The source and target languages must have formally well-defined syntax and semantics."

2. "The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to maintain the meaning of the specification."

3. "Rigorous arguments must be provided to validate the translator and/or the generated code."

4. "The implementation of the translator must be rigorously tested and treated as high-assurance software."

5. "The generated code must be well structured, well documented, and easily traceable to the original specification."

Requirements 3 to 5 address the validation and correctness of the implementation of the translation process. The third requirement continues the formal approach to code generation, while last two are more practical requirements. These requirements are used later, both as a guide to developing our code generation tool - Spi2Java, and as criteria to evaluate the tool against.

2.3.2 Using Prolog to Implement Compilers

Hoare et al describe an approach to creating verifiable compilers in [10]. They develop a *compiling specification*, defined in terms of logical rules, that describes how source language constructs are translated into object code. Initially they define some axiomatic rules and develop a program refinement calculus for working with those rules, that allows them to show that the execution of the object code is equivalent or as good as the source program language segments that they are intended to implement.

To prototype their verifiable compiler, Hoare et al use Prolog. Prolog's declarative, rules based nature allows the compiling specification, which as mentioned is defined in terms of logical rules, to be very close to the implementation - the implementation is just a direct encoding of those rules in Prolog.

Such an approach addresses the first three requirements of Whalen and Heimdahl - provided you are confident in the Prolog implementation: formally defined source and target languages are used, the translation - in this case defined by a compiling specification - is proven to preserve the source language semantics and the translator or compiler is a direct implementation of the compiling specification which has been formally proven to be correct.

In a research project, meeting the fourth requirement may not be feasible, given the limited resources available, compared to the resources actually required to thoroughly test any system. The final requirement can be largely addressed by annotation of the generated code with comments indicating the source language construct that each segment was translated from.

2.4 Discussion

In this background chapter we have looked at the security protocol development process and placed the work described in this dissertation in the context of that process. We have introduced languages for specifying security protocols, discussed how they can be used to analyse security protocols and described how their formal nature is crucial for the correct implementation of security protocols. Requirements for high integrity code generation were introduced. These requirements are employed both as a guideline and as evaluation criteria for the code generation tool Spi2Java developed during the course of this work and described in this dissertation. Finally the use of Prolog for implementing compilers is briefly summarised, showing how Prolog language properties assist the implementation code generation, or compiler, software.

Chapter 3

Formal Specification with Spi Calculus

In this chapter we describe the Spi Calculus by first introducing process algebras in general and then a specific process algebra called the π-calculus. We then discuss how the Spi Calculus extends the π calculus and its suitability for specifying security protocols and verifying their properties. Finally we introduce our modifications to the Spi Calculus to make it suitable for specifying input for an automatic code generation tool.

3.1 Process Algebras

Process algebras are languages that have been developed to facilitate the formal description and analysis of complex, communicating systems. As network security protocols are a special case of communicating systems, popular process algebras such as CSP (Communicating Sequential Processes) and the π-calculus have been used to reason about their properties [40] [1], either proving them correct or revealing flaws. A successful example of this application of process algebras, is Lowe's discovery of a flaw in the Needham-Schroeder public key authentication protocol [39]. The flaw was revealed by specifying the protocol in CSP and checking the specification with the FDR model checker, which produced a trace of an attack on the protocol [40].

3.2 The π-Calculus

The π-calculus consists of terms, such as names, variables and pairs, and processes, such as sending and receiving messages and parallel composition of processes. Each of the processes has a simple, well defined behaviour. Despite its small size and relative simplicity, the π-calculus is powerful in its ability to describe concurrent systems.

We give a description of the syntax of the π-calculus and the behaviour of its processes below for convenience, summarising that in [1].

3.2.1 Grammar

To define the syntax an infinite set of *names* and an infinite set of *variables* over those names are assumed.

Terms

Letting m, n, p and r range over names and x, y and z over variables, the syntax of terms of the π-calculus is defined by the grammar:

$L, M, N ::=$

n	a name
(M, N)	a pair
0	zero
$suc(M)$	successor
x	a variable

Processes

The process syntax is defined as follows:

$P, Q, R ::=$

$\overline{M}\langle N\rangle.P$

$M(x).P$

$P \mid Q$
$(\nu n)P$
$!P$
$[M\, is\, N]P$
$\mathbf{0}$
$let\,(x,\, y)\, =\, M\, in\, P$
$case\, M\, of\, 0\, :\, P\, suc(x)\, :\, Q$

3.2.2 Informal Semantics

The behaviour of the π-calculus process actions can be described in an intuitive and informal manner:

- $\overline{m}\langle N\rangle.P$ will output N on channel m when an interaction with an input process occurs, and then run P.
- $m(x).P$ will input a term, say N, when an interaction occurs and then run $P[N/x]$, i.e. all occurrences of x substituted with N.
- $P \mid Q$ is the parallel composition process and behaves as P and Q running in parallel.
- $(\nu n)P$ creates a new, private name n and behaves like P. This process can be used to model the generation of nonces.
- $!P$ is replication, it behaves as an infinite number of processes P running in parallel.
- $[M\, is\, N]P$ behaves like P if the term M is the same as the term N or else it does nothing.
- $\mathbf{0}$ does nothing.
- $let\,(x,\, y)\, =\, M\, in\, P$ allows M to be split. If M is a pair $(N,\, L)$ then $P[N/x][L/y]$ is run, otherwise the process does nothing.
- $case\, M\, of\, 0\, :\, P\, suc(x)\, :\, Q$ will run P if M is 0, $Q[N/x]$ if M is $suc(N)$ or do nothing.
- $case\, L\, of\, \{x\}_N\, in\, P$ runs as $P[M/x]$ if L is an encryption of M with N, otherwise it does nothing.

3.2.3 Example

To illustrate the use of the π-calculus to specify a protocol, we give a trivial example: This (insecure) protocol involves an *initiator* that sends a request message to elicit a response from a specific remote principal, the *responder*. The initiator may then perform some kind of processing, not specified by the protocol, on the response.

The initiator, A, sends a request message (message 1), containing a term M to the specified responder B. M is a term suitable for uniquely identifying the current protocol run or session.

Upon receipt of the request, the responder replies with message 2, containing M concatenated with its own response message N. The initiator receives this message and confirms that it is the corresponding response from message 1, by checking that the first term of the response matches the term M it sent in the request. If the terms match initiator is assured that the response corresponds to the current request and not one from a previous run of the protocol that may have been delayed on the communications channel. The initiator can then do whatever processing is required on N.

Informally the protocol might be specified in standard notation as:

$$\begin{array}{ll} 1 & A \rightarrow B : M \\ 2 & B \rightarrow A : M, N \end{array}$$

This informal specification does not, however, explicitly indicate all the protocol requirements. The requirement that the initiator check that the first term of the response message matches M is implicit; leaving the exact actions to be performed to interpretation by the protocol implementor. Through oversight or inexperience, the programmer may misinterpret this specification, and introduce an error into the protocol implementation code.

In contrast, the π-calculus specification of the same protocol, allows the protocol specifier to make the requirement explicit, albeit at the expense of a higher level of abstraction and ease of understanding. To specify this protocol in π-calculus, parameterised processes are defined for the initiator and responder roles:

$$\begin{array}{lll} Initiator(C_{AB}) & = & (\nu M)\overline{C_{AB}}\langle M\rangle. \\ & & C_{AB}(x). \\ & & let\,(y,\,N)\,=\,x\,in \\ & & [y\,is\,M]\,F(N) \\ \\ Responder(C_{AB}, N) = & & C_{AB}(M). \end{array}$$

$\overline{C_{AB}}\langle(M,\, N)\rangle.$
nil

A further process, consisting of the parallel composition of the initiator and responder processes, is defined to specify a run of the protocol:

$$\begin{aligned} Protocol \quad &= \quad (\nu N)(\nu C) \\ &\quad\quad (A(C) \,|\, B(C,\, M)) \end{aligned}$$

In π-calculus specification, the initiator is provided with a channel, C_{AB}, for communication with the responder. The initiator generates a random value, M, and sends it on the channel. It waits to receive a response message, x, which it attempts to split into a pair of terms, y and N. If the split is successful, the initiator proceeds to match the first term, y, against the term M it originally sent in the request message. Provided the match is successful, the initiator may assume that this is the correct response message and perform whatever processing it needs to on N, by executing the process F.

The responder process is simpler; it waits to receive a request message, and then sends a response back containing the request paired with its own response term N.

3.2.4 Transition Semantics

The behaviour of the processes of the π-calculus are formally defined by transition semantics in [51]. This formal definition provides the precise and unambiguous specification of behaviour required for protocol analysis and implementation. Rather than give a complete definition of the semantics of the π-calculus processes, we shall introduce the more relevant definitions later in the thesis when we develop a translation to executable code.

3.3 The Spi Calculus Extensions to π

The Spi Calculus was developed by Abadi and Gordon in [1] by extending the π-calculus to provide a formal framework for reasoning about the properties of security protocols. Instead of using existing π-calculus constructs to model security functions, Spi add terms and process to the π-calculus to provide primitives for encryption, decryption and message digests.

3.3.1 Grammar

In extending the π-calculus, three new constructs representing the cryptographic primitives are added: two terms for encryption and message digest creation and a process for decryption.

Terms

$\{M\}_N$	encryption
$hash(M)$	message digest

The term $\{M\}_N$ is the cipher text obtained by encrypting a term M with the N.

The hash function $hash(M)$, evaluates to the result of creating a message digest from a term M.

Processes

A single process

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

is added for decryption. This process behaves as P provided the term L can be successfully decrypted using the key term N. If it cannot the process halts.

3.3.2 Example

As an example of the use of Spi we specify a simple message exchange in the Spi Calculus:

$$\begin{array}{lcl} A & = & \overline{C_{AB}}\langle\{I_A\}_K\rangle. \\ B & = & C_{AB}(x_{cipher}). \\ & & x_{cipher}\, case\, of\, \{x\}_K\, in\, [x\, is\, I_A] \\ & & F \end{array}$$

A and B are process definitions representing the initiator and responder respectively. C_{AB} is a public, insecure channel for communication between the two processes. I_A is a value that uniquely identifies the initiator and K is a key that is shared by the initiator and the responder.

A encrypts the identifier I_A with the key K, and sends the result on the channel C_{AB} when an interaction occurs with another process.

B waits for an interaction on the channel C_{AB}, and receives a message that is bound to variable x_{cipher}. B then attempts to decrypt x_{cipher} with the key K. If the decryption is successful, the resulting plain text is bound to variable x. If it is not the process becomes stuck. If x is equal to the initiator's identifier, I_A, then the responder will perform some desired action specified by the process definition F.

As well as allowing the behaviour of the protocol participants to be specified, the Spi Calculus allows a run of the protocol to be defined by allowing A to B to be executed in parallel. An instance of the protocol can thus be specified as:

$$I(M) \quad = \quad (\nu K)(A|B)$$

The key K is created by the restriction process, and is only in the scope of the processes A and B. This accurately models the fact the K is a shared secret between the two protocol participants whose behaviour is specified by A and B.

3.3.3 Spi Semantics

As Spi is an extension of π, π's transition semantics are applicable. However, a further formal definition of the Spi Calculus, using reaction relations, is given in [1]. This definition is also suitable as a basis for reasoning about the properties of security protocols specified in the Spi Calculus as demonstrated in [1]. Again, rather than give a complete definition of the semantics of the π and Spi processes, we shall introduce the relevant definitions later in this dissertation, when we develop a mapping from Spi process actions to executable code.

3.4 Modification of Spi for Code Generation

To facilitate code generation, we define a variation of the standard Spi Calculus by making a few modifications. In this section we describe the modifications and the rationale for them.

3.4.1 Supporting an Executable Subset of Spi

A subset, albeit a comprehensive one, of the terms and processes defined by the original calculus are supported. The reasons for not supporting some of the constructs are given below:

- **Successor Function:** This function may be required by some protocols, specifically those that require the response to a challenge nonce to contain the value of the nonce incremented by a specific value. Currently we have no need for it, but it would be a fairly simple addition.
- **Integer Case Process:** This process would be required to verify the response nonce in the scenario described above.
- **Summation:** The summation process is non-deterministic. As it is not required for specifying the Spi process for an individual protocol role, it can be conveniently left out.
- **Replication:** Replication, which is defined to behave as infinitely many copies of a process running in parallel, is not supported for code generation as it is, by definition, unimplementable.

3.4.2 Spi Specification in Plain Text

Minor syntactic changes are defined to allow security protocols to be described in plain, ASCII encoded, text files. These include using the *!* and *?* characters to indicate output and input respectively - as in Occam [57], a programming language based on the CSP process algebra.

3.4.3 Support for Retrieving Public and Private Keys

We introduce the terms $pub(x)$ and $priv(x)$, which evaluate to the public and private keys of the principal x respectively, following an element of the syntax used by SPL [15].

3.4.4 Tuples

We adopt the standard abbreviations in [1] for constructing tuples from pairs and splitting tuples. $(N_1, ..., N_{i-1}, N_i)$ is written for $(N_1, (N_2, (..., (N_{i-1}, N_i))))$ and $let\,(x_1, ..., x_i) = M\,in\,P$ is written for $let\,(x_1, t_1) = M\,in\,let\,(x_2, t_2) = t_1\,in\,...\,let\,(x_{x-1}, x_i)\,in\,P$.

3.4.5 Timestamp Validation

A process that checks the validity of a timestamp is defined as: $case\, x\, valid\, in\, P$. This process behaves as P, provided the timestamp, x, is valid, i.e. it has not expired.

3.4.6 Typed Variables

To facilitate the code generation, we have added type indicators to the syntax of all the binding process actions, i.e. input, restriction, decryption or pair splitting. Instead of just declaring the variable, x, a type declaration is appended to get $x : Type$. For instance, the restriction process that creates a new nonce is $(x : Nonce)$. The syntax we have chosen follows that for the applied Spi Calculus described in [47].

Type declarations allow the code generator to specify the type of the corresponding variable in the implementing language. The supported types are:

- **Channel** An atomic type representing a channel for communication between multiple principals.
- **Encryption** An atomic value type that is an encryption of a term, or cipher-text. Although and Encryption is atomic it can be used, via the decryption process, to instantiate a term.
- **Hash** An atomic type representing a message digest.
- **Identifier** An atomic type for values that uniquely identify a principal.
- **Key** An atomic type for symmetric key values.
- **Nonce** An atomic type for nonce values.
- **Pair** A compound type for terms formed by pairing an atomic value, on the left, with a term on the right. For example, (n, M) where n is an atomic value such as a nonce or key, and M is an arbitrary term.
- **Term** A compound type for terms.
- **UserData** An atomic type for arbitrary user data that is not interpreted by the protocol itself.
- **Timestamp** An atomic type for timestamp values.

3.4.7 The Modified Spi Grammar

With these modifications, our Spi Calculus variant for code generation is defined by the grammar in figure 3

L, M, N ::=

n	a name
(M, N)	a pair
x	a variable
$\{M\}N$	encryption of M with N
$hash(M)$	hash of M
$pub(n)$	public key of n
$priv(n)$	private key of n

P, Q ::=

$c!\langle N\rangle.P$
$c?(x : \mathbf{Type}).P$
$(P \mid Q)$
$(n : \mathbf{Type})P$
$[M\, is\, N]P$
nil
$let\,(x : \mathbf{Type}, y : \mathbf{Type}) = M\, in\, P$
$let\,(x_1 : \mathbf{Type}, ..., x_i : \mathbf{Type}) = M\, in\, P$
$case\, L\, of\, \{x : \mathbf{Type}\}\, in\, P$
$case\, T\, valid\, in\, P$

Type ::=

$Channel$
$Encryption$
$Hash$
$Identifier$
$Identifier$
Key
$Nonce$
$Pair$
$Term$
$UserData$
$Timestamp$

Figure 3: The modified Spi Calculus grammar.

3.5 Discussion

The Spi Calculus trades off some of the higher level abstraction and clarity of the standard notation, for an explicit specification of the protocol logic, that define the logical steps required to achieve the security goals of a protocol. However it still abstracts some low-level implementation details: such as the format of messages and message components, the cryptographic algorithms used and the communication mechanisms. As we shall discuss later, this level of abstraction allows the implementation and verification of the protocol logic to be isolated from that of the cryptographic algorithms and network communication.

By making some minor modifications to the Spi Calculus and its syntax, we have defined a variant that is suitable for specifying input for a code generation tool, while retaining the Spi Calculus semantics that make it suitable for analysing and verifying security protocols.

Chapter 4

Sn2Spi: Translating Informal to Formal Specifications

This chapter covers the Sn2Spi translation tool that automates the translation of informal standard notation specifications to formal Spi Calculus specifications. Essentially the standard notation is defined as shorthand for a set of Spi processes, with each individual security protocol role specified by a Spi process without parallel components.

We begin by defining a grammar for a strict version of the standard notation, making it suitable for input into an automated software tool. We then define rules for translating specifications in this notation to Spi Calculus processes. Finally we elaborate on the implementation of these rules by describing the development of Sn2Spi.

Unlike the Spi2Java code generation tool (which forms the core of this work and is covered in chapter 6) a less rigorous approach was taken in developing Sn2Spi. Sn2Spi is not intended to meet the same requirements for high integrity code generation that Spi2Java is evaluated against. However, it does aim to automate the process of translating standard notation specifications to Spi Calculus processes. This is beneficial, as most existing security protocols are only specified in some form of the standard notation.

Once translated to Spi, the protocol specifications can be manually or automatically verified, to ensure that they achieve the intended security goals.

4.1 Standard Notation Syntax

Since many variations on the standard notation exist, we define an exact syntax for the notation in figure 4.

```
Protocol     ::= Roles Declarations ( Possession )* Messages
Roles        ::= ( <ROLES> <ID> ( <COMMA> <ID> )* )
Possession   ::= <POSSESSION> <ID> <COLON> <ID>
Declarations ::= ( ( <CHANNEL> | <IDENTIFIER> | <KEY> | <NONCE> |
                     <TIMESTAMP> | <USERDATA> ) <ID> ( <COMMA> <ID> )* )+
Messages     ::= ( Message )*
Message      ::= <ID> <ARROW> <ID> <COLON> Components
Components   ::= ( Component ( <COMMA> Component )* )
Component    ::= ( <ID> | CipherText | Hash )
Hash         ::= <HASH> <LPAR> Components <RPAR>
CipherText   ::= <LBRACE> Components <RBRACE> Key
Key          ::= ( <ID> | <PUB> <LPAR> <ID> <RPAR> | <PRIV> <LPAR> <ID> <RPAR> )
```

Figure 4: The BNF grammar for the standard notation.

We extend the standard notation to allow information about the principals and their possessions to be made explicit. A *preamble*, similar but simpler and less flexible than that used by CAPSL [21], is defined. The preamble contains a list of participant *roles* defined by the protocol, *type declarations* for the message components and a list of *initial possessions* for each of the roles.

1 $A \rightarrow B : \{n, A\}pub(B)$
2 $B \rightarrow A : \{n, m\}pub(A)$
3 $A \rightarrow B : \{m\}pub(B)$

Figure 5: The Needham-Schroeder-Lowe Public Key Authentication protocol.

The Needham-Schroeder-Lowe Public Key Authentication protocol in figure 5, is specified in figure 6 using the defined syntax. This specification defines roles for an initiator, A, and a responder, B.

In this specification the initiating principal is A, the responder is B and $pub(X)$ is a function that returns X's public key. Not shown in the example is the corresponding private key retrieval function, $priv(X)$, that returns X's private key. An extra possession, cAB, is declared for each role, indicating a communication channel shared by the Spi processes that specify the initiator and responder roles.

```
Principals A, B

Channel cAB
Identifier A, B
Nonce m, n

Possession A:cAB
Possession A:A
Possession A:B
Possession B:cAB
Possession B:B

A -> B: {n, A}pub(B)
B -> A: {n, m, B}pub(A)
A -> B: {m}pub(B)
```

Figure 6: The NSL protocol specified for input into Sn2Spi

4.2 Translation Rules

4.2.1 Roles

For each of the principals A_1, ..., A_n in the informal specification, corresponding Spi process definitions, $A_1 = P_1$, ..., $A_n = P_n$, are created specifying the principals' roles. The processes P_1, ..., P_n do not contain any parallel components.

4.2.2 Initial Possessions

A protocol may require that the principals fulfilling protocol roles have certain values in their possession, such as shared keys and principal identifiers, prior to a protocol run commencing. These are the *initial possessions* of the principal role. To specify this in Spi, a principal A's process definition $A = P$, is parameterised by the initial possessions. Thus if a principal fulfilling the role A is expected to have the possessions p_1, ..., p_n before commencing a run of the protocol, the parameter list for $A = P$ is updated to $A(p_1, ..., p_n) = P$.

4.2.3 Protocol Messages

Each message $A \rightarrow B : m$ in an informal specification indicates principal A sending principal B a message m. In Spi this corresponds to A's process performing an output action that is intended to interact with an input action performed by B's process, on a dedicated channel for communication between the two. Thus for each pair of communicating roles A and B, a Spi channel c_{AB} is defined and added to the parameter list of both A and B's Spi process definitions.

The rules below describe how to develop the Spi process definitions $A = P$ and $B = Q$ for each message in the informal specification. Initially P and Q are empty processes, but they are extended as the rules are applied.

The Sending Principal Role

For each message $A \rightarrow B : m$, the following actions are taken to develop the Spi process definition for A's role, $A = P$:

1. The parameter list of the process definition $A = P$ is extended to incorporate the channel c_{AB}, provided it does not already contain this parameter.

2. If any of the components of m are nonces or timestamps and this is their first occurrence in the informal specification, a Spi restriction action is appended to P to become P'. For example, if the first message in the informal specification is

$$A \rightarrow B : \{n, A\}pub(B)$$

   then this step transforms $A = P$ to $A = P(n)$.

3. Each component in m is paired with the component on the right to construct the corresponding Spi term M. This follows the convention adopted in [1] where a tuple (N_1, ..., N_i, N_{i+1}) is an abbreviation for ((N_1, ... , N_i), N_{i+1}) for $i \geq 2$. An output action $c_{AB}! < M >$ is appended to P' to become P''. Continuing the example, we get Spi process definition

$$A(C_{AB}) = (n)c_{AB}! < \{n, A))\}pub(B) > ...$$

   for A's role.

Once the rules have been applied, P is set to P'' and the process is repeated with the next message.

The Receiving Principal Role

The rules to develop the Spi process for B's role are more complex. This is due to actions that B's Spi process may need to perform to split the received message into its components, decrypt and verify components. Pair splitting is necessary as the message m is received by B as a single Spi term, say M, on the channel c_{AB}. B's Spi process then needs to recursively split M into its component parts. Component verification is required when m contains components that are supposed to be equal to values that B already has in its possession set or can calculate. Two specific examples are nonce validation (where a nonce returned by A in response to a challenge nonce sent by$B$ is compared to the original challenge nonce) and timestamp validation (where a received timestamp is checked to see if it has expired).

For each message $A \rightarrow B : m$, the following actions are taken to develop the current Spi process definition for B's role, $B = Q$:

1. The parameter list of process definition $B = Q$ is extended to incorporate the channel c_{AB}, provided it does not already contain this parameter.

2. Again we follow an abbreviation in [1] that allows us to write $let\,(x_1, ..., x_n) = M\,in\,P$ instead of $let\,(x_1, t_1) = M\,in\,let\,(x2, t_2)\,in\,...\,let(x_{n-1}, x_n) = t_{n-1}\,in\,P$. Note that atomic components include cipher texts, which can obviously be used to generate further Spi terms via the Spi decryption action. Q becomes Q' by having the necessary pair splitting actions appended to it. For example, the message:

   $$1)\,A \rightarrow B : n, A, B$$

   is translated to the Spi process for B's role as follows:

   $$B(C_{AB}) = c_{AB}?M.let\,(n, A, B) = M\,in\,...$$

3. Where a message, or message component, is of the form $\{m\}k$ in the informal specification, the Spi process for the receiving principal has a decryption action $case\,M\,of\,\{x\}K\,in$ where M is the Spi term corresponding to $\{m\}k$ and K is the Spi term corresponding to the key k. As x may be a compound term itself, pair splitting actions may need to be append to the process specification as described in step 2. Appending the required decryption actions change Q' to Q''.

4. When response nonces, or other components that correspond to values B already possesses, are in the received message m, they need to be compared against the corresponding value in

B's possession. To specify this, a match process for each such component of m is appended to B's Spi process definition taking Q'' to Q'''. The behaviour of the match is exactly that required: if the components match, the process implementing the protocol can continue, if not it halts.

5. Timestamp components can be validated by appending Spi timestamp validation actions to the process taking Q''' to Q''''.

After the rules have been applied for the current message, Q is set to Q'''' and the process is repeated for the next message.

4.3 Implementation of Translation Rules

The JavaCC compiler constructor, a tool combining some of the functionality of Lex and Yacc, was used to implement the translation rules in the Sn2Spi tool. In combination with some lexing (token parsing) rules, the grammar in figure 4 specifies parsing rules for JavaCC to use in creating Sn2Spi.

Along with the parsing rules we specified actions to generate the Spi processes for each message. These actions follow the rules defined above by creating and populating nested objects that represent Spi terms, processes and process definitions, during the parsing process. At the top level of these objects are ones that represent the Spi processes for each of the protocol roles. Once constructed they can be asked to return a string containing the Spi specification for role they represent. The Java code that implements the translation rules is thus split between the JavaCC specification and the Java classes for these objects. The JavaCC specification from which Sn2Spi was generated is available at [5], along with the source for the supporting classes.

4.4 Example

The specification of the Needham-Schroeder-Lowe protocol in figure 6 is saved as an ASCII encoded text file and input into Sn2Spi. The Sn2Spi generated output is listed in figure 7.

```
A(A:Identifier, cAB:Channel, B:Identifier) =
        (n:Nonce)                                                (Msg 1)
        cAB!<{n, A}pub(B)>.
        cAB?(tmp3:Encryption).                                   (Msg 2)
        case tmp3 of {tmp0:Pair}priv(A) in
        let (tmp1:Nonce, m:Nonce, tmp2:Identifier) = tmp0 in
        [tmp1 is n]
        [tmp2 is B]
        cAB!<{m}pub(B)>.                                         (Msg 3)
        nil

B(cAB:Channel, B:Identifier) =
        cAB?(tmp1:Encryption).                                   (Msg 1)
        case tmp1 of {tmp0:Pair}priv(B) in
        let (n:Nonce, A:Identifier) = tmp0 in
        (m:Nonce)
        cAB!<{n, m, B}pub(A)>.                                   (Msg 2)
        cAB?(tmp3:Encryption).                                   (Msg 3)
        case tmp3 of {tmp2:Nonce}priv(B) in
        [tmp2 is m]
        nil
```

Figure 7: Spi specification of the NSL protocol generated by Sn2Spi.

4.5 Discussion

Sn2Spi automates the process of translating informal specifications to formal ones. By translating an existing standard notation protocol specification into a formal Spi Calculus specification, the design and specification phase of the security protocol development process (described in figure 1) is by-passed and the formal Spi Calculus specification can be subjected to the protocol analysis phase: manual inspection and development of proofs and/or automatic verification using a model checker such as MMC [54]. This process ensures that the Spi Calculus translation of the protocol correctly describes the intended behaviour of the standard notation specification. The Spi Calculus translation is suitable for use by the Spi2Java code generator without any modification, allowing the implementation phase of the security protocol development to be automatically completed.

Sn2Spi is a useful aid in translating existing, informal protocol specifications to formal Spi Calculus specifications. It is not intended to encourage the development of new security protocols by using the standard notation to define an informal specification, and then converting that to Spi for analysis and implementation. Rather, the security protocol engineer should specify security protocols in a more formal manner, making the intended behaviour and security properties of the protocol clear, precise and unambiguous.

Chapter 5

Security Protocol Primitives API

The Security Protocol Primitives (SPP) API provides access to the primitive cryptographic and network communication functions required to implement a network security protocol. The API uses a provider model, similar to that employed by Sun's Java Cryptography Architecture [60], to expose this functionality to the user in a manner independent of the underlying implementation.

In this chapter we discuss our rationale for, and approach to, developing this API; we outline the functionality required of the API by the user; describe the design of the API, both in terms of the classes and methods exposed to the user, and those defined for compliant provider implementations; we describe some provider implementations we have developed and their use; justify the abstraction of cryptographic operations and discuss the benefits of the API to this work.

5.1 Approach

A security protocol implementation can be split into two discrete parts. The first part is the *security protocol logic* that determines *when* and *what* actions - such as decryption, comparison and instantiation of message components - to perform. The second part is the code that implements these actions - it determines *how* these actions are performed.

SPP defines a set of APIs that are exposed to the *user*. In this case the user is the code that implements the security protocol logic - and shall be referred to as such throughout the rest of this chapter. SPP also defines an interface for *providers*, allowing third parties to provide an implementation that conforms to this interface. The provider is free to use whatever asymmetric, symmetric and hashing

algorithms they choose, combined with their preferred mode of network communication.

An implementation of a security protocol that uses SPP for cryptographic and network communication functions, can change between different provider implementations without modifying its own code that implements the security protocol logic.

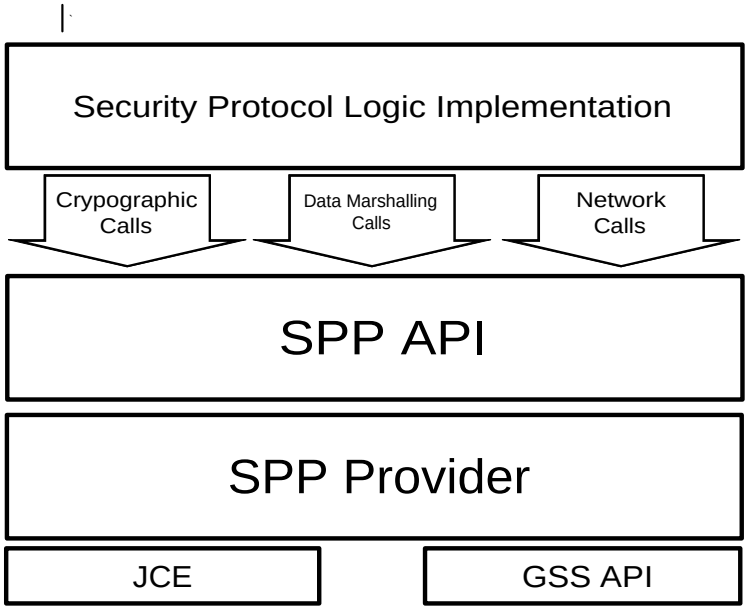

Figure 8: The SPP API.

Providers may implement message components, such as principal identifiers, to conform to those defined by existing standards and infrastructures e.g. X.509, Microsoft Active Directory services or Internet Protocol addresses.

A provider may develop an implementation using elliptic curve asymmetric encryption, AES symmetric encryption and SHA message digests. For example: the provider may require 128 bits of random data for nonce message components; component data may be marshalled for transmission over networks by converting it to an XML document formatted according to a predefined Document Type Definition (DTD) or schema; the resulting message could then be sent to remote principals using SOAP over HTTP.

Another provider may prefer a more traditional approach: implementing RSA encryption, 3DES symmetric encryption and MD5 message digests; pseudo randomly generated 32 bit integers may be

deemed sufficient for nonce components; other data may be marshalled to a compact binary format, and written and read directly to and from a stream sockets interface to TCP/IP.

5.2 SPP User Functionality

This section serves as a high level functional specification by describing the cryptographic and network functions exposed to the SPP user.

5.2.1 Message Component Types

Network security protocols exchange messages consisting of one or more components. SPP defines abstraction for well defined components that can be operated on by SPP functions and for general user data:

- *Encrypted Component*: A message component containing encrypted data.
- *Message Digest*: A component containing a message digest or hash.
- *Nonce*: A data component generated in an unguessable or (pseudo) random way.
- *Principal Identifier*: A component that uniquely identifies a principal.
- *Public Key*: The public component of a public private key pair.
- *Symmetric Key*: A symmetric key component.
- *Timestamp*: A timestamp.
- *User Data*: A message component containing arbitrary user data.

A *Private Key* interface also exists, but is not exposed as a message component, preventing transmission over a network as part of a protocol message. Private keys generally require a higher level of security than symmetric keys - which often play the role of session keys. Revealing a private key can have severe consequences, and hence they are rarely transmitted over a network.

5.2.2 Cryptographic Functions

The SPP supports the following cryptographic operations:

- Symmetric encryption and decryption.
- Asymmetric encryption and decryption using public and private key pairs.
- Generating message digests components.
- Generating nonce message components.
- Generating and validating timestamp message components.
- Retrieving principals' public and private keys in the possession of the local principal.

5.2.3 Message Component Functions

The user, i.e. the code that implements the security protocol logic, dictates which message components to generate or verify and when. In the case of a nonce challenge, the user determines when a new nonce is generated and incorporated into a message to be sent to a remote principal. At a later stage the remote principal may return a copy of this nonce. The user will determine which is the original nonce it should be compared to, and what to do if the two aren't equal. How the nonces are compared and packed into messages is of no concern to the user, other than trusting that the operations are performed correctly, and are handled by SPP.

Some of the message component operations are:

- Compare two message components for equality.
- Pack (serialise) message components so that they can be sent as binary data.
- Unpack (deserialise) message components from a received binary data.

5.2.4 Network Communication Functions

The network communication functionality is summarised as follows:

- *Channels* abstract network protocol specific details by exposing an endpoint for communication with a remote principal.
- Arbitrary message data can be sent on a channel.
- Channels guarantee that messages are delivered in their entirety and in order.
- Functions for packing and unpacking of message components from data sent and received over channels.

5.3 SPP Design

5.3.1 Architecture

The design of the SPP API aims to provide high level, simple access to cryptographic functions by hiding implementation specifics from the user. To achieve this, established design patterns are used to decouple the implementation of cryptographic algorithms from the exposed user APIs. The implementation can be written by any provider as long as it conforms to the defined SPP provider interface. Multiple providers can be used, though obviously not concurrently, and the desired provider can be selected at runtime.

Network functions are abstracted. The SPP user only needs to deal with the *channel* interface to communicate with a remote principal. The provider is responsible for linking principal identifiers to real network addresses, such as IP hostnames or directory entries, and establishing connections to remote principals. Likewise, the provider must deliver user message data in its entirety, regardless of the underlying implementation of the channel abstraction; e.g. if the provider uses a datagram protocol, it must ensure that the user does not need to do any work to order the data and verify that it is complete when reading it from a channel.

The format of messages and message components - such as principal identifiers, nonces and timestamps - is determined by the provider. These specifics are hidden from the user; they will call SPP defined user functions (implemented by the provider) to operate on message components.

Principal identifiers, shared symmetric keys, and public and private keys in the possession of the local principal may be stored in a provider specific format on file, or retrieved by some end user interface such as a console.

Design patterns are solutions to commonly occurring problems and provide common terms of reference for developers. Figure 9 shows the bridge pattern that is used to separate the abstract interfaces for the message components and cryptographic and network operations, from the concrete SPP provider implementation. The factory method and builder patterns are used to allow SPP user code, to access the SPP provider implementation to instantiate, encrypt, decrypt, hash and validate message components. The factory pattern also allows a security protocol implementation to choose an SPP provider at runtime.

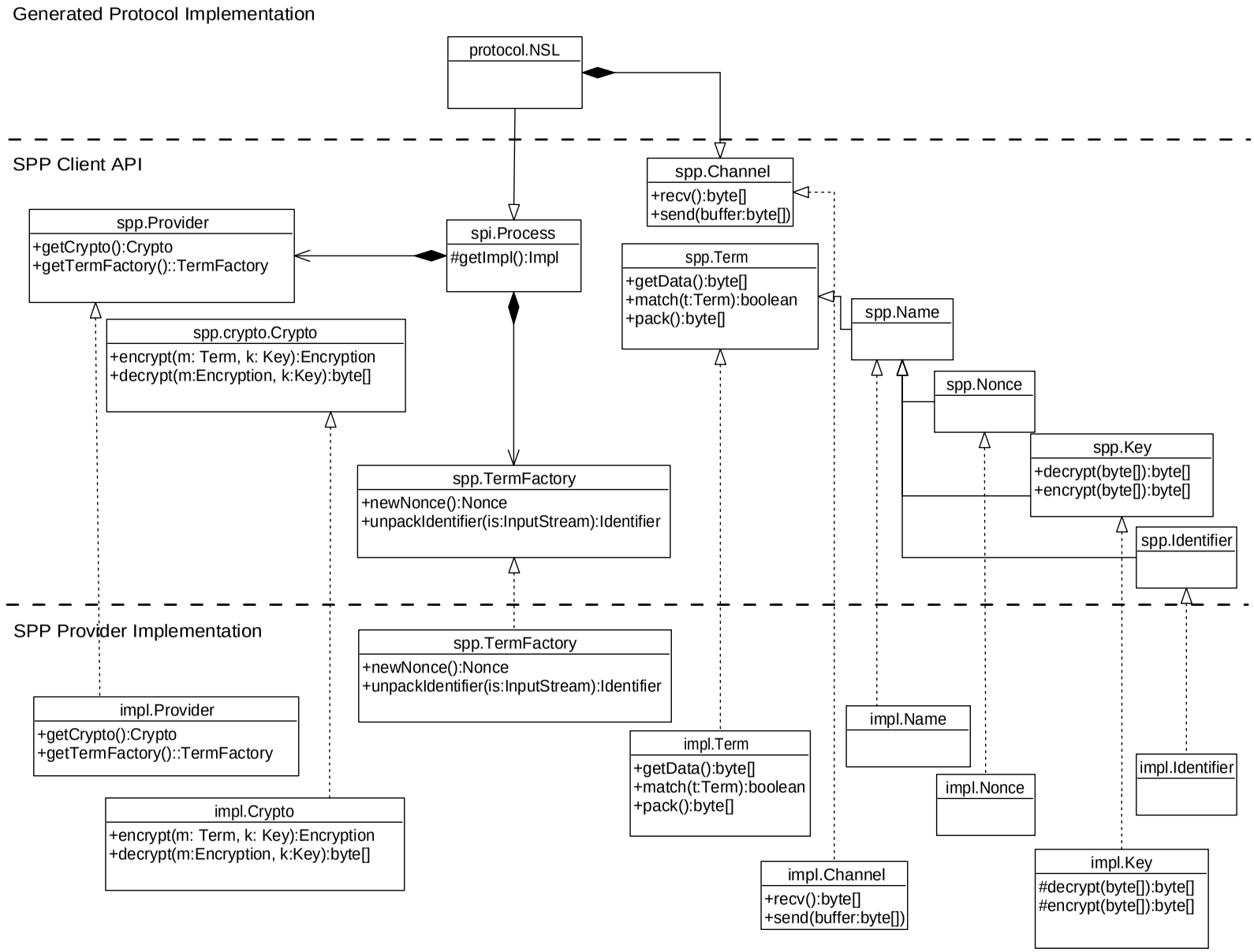

Figure 9: UML diagram of a subset of the SPP API.

5.3.2 SPP Client API

The client part of the SPP API defines the public interfaces that clients, such as Spi2Java generated security protocol implementations, can use to access network and cryptographic functionality. The client interface is separated into two main areas: abstract classes and interfaces that represent message components and channels for communication, and functions that perform cryptographic operations, construct messages and return objects that conform to the specified interfaces or extend the abstract classes.

The message component interfaces are declared in the `sprite.spp.term.*` package. They extend the `Term` base interface. Figure 9 shows how the `Name` interface inherits from and specialises this interface. The name interface is then extended `Nonce`, `Identifier` and other name message components. Compound message components such as `Encryption` and `Hash` also extend `Term`.

The `sprite.spp.Factory` interfaces provides access to instantiating these message components as well the implementations of the `sprite.spp.crypto.Crypto` and `sprite.spp.net.Channel` classes that expose methods for cryptographic operations and provide endpoints for communication, respectively.

5.3.3 SPP Provider Interface

The provider part of the SPP is also twofold: there are the client interfaces and abstract classes that provider class implementations must conform to, but there are also factory interfaces that must be implemented by a provider. These factory interfaces allow the internals of the SPP API to instantiate provider implementations of interfaces and pass them to the requesting client.

5.4 Abstracting Cryptographic Implementation Details

The scope of this work has been limited by not addressing the issue of the correctness of the cryptographic implementation. By de-coupling it from the other code that implements the security protocol, it can be addressed independently, as discussed in Section 9.2 on future work.

5.5 Implemented Providers

Our current SPP Providers all use the Bouncy Castle JCE Provider version 1.22 for Java 1.4. Any other JCE compliant provider can be substituted with a one line code change.

5.5.1 Standard Provider

The Standard Provider is a practical implementation for real world use. It the client code to communicate with remote principals over TCP/IP networks, such as the internet. It employs AES (Advanced Encryption Standard) for symmetric cryptography, RSA for asymmetric cryptography and SHA for generating message digests.

Message components can be packed (serialised) to a data field suitable for transport over a network. The data field for some message components may start with a length indicator, indicating the length of the field excluding the length indicator. The length indicator is a two byte unsigned integer, encoded in big endian (network) order. When such a length indicator is used, the maximum length of the field, without the indicator, is 65535 bytes.

- Encrypted Component: A length indicator followed by the encrypted data. The data may be encrypted by using either the AES symmetric encryption algorithm using a 128 bit key in CBC (Cipher Block Chaining) mode with PKCS5 padding, or RSA public or private encryption using the standard ECB (Electronic Code Book) mode.
- Message Digest: A 20 byte field containing a SHA message digest.
- Nonce: A 16 byte field containing pseudo randomly generated data. The TCP Provider uses the `java.security.SecureRandom` API. JCE Providers which implement this interface must conform to the randomness tests specified in FIPS 140-2, Security Requirements for Cryptographic Modules, section 4.9.1 [28] and RFC 1750: Randomness Recommendations for Security [17] [59].
- Principal Identifier: A 16 byte field containing a string of 16 ASCII encoded characters that uniquely identifies a principal.
- Public Key: A length indicator followed by an ASN.1 encoding of an RSA public key according to the X.509 standard.
- Symmetric Key: 16 bytes containing the raw data of a 128 bit AES key.
- Timestamp: An 8 byte signed long integer. The timestamp reflects the number of milliseconds elapsed between the current time and midnight on the 1st January 1970; this is as returned by a call to the Java `long java.lang.System.currentTimeMillis()` API.
- User Data: A length indicator followed by up to 65535 bytes of data in user specified format.

The Standard Provider ensures the delivery of messages in their entirety by prefixing them with a length, as described above. On the receiver's side, the provider automatically reads two bytes from its TCP/IP endpoint and determines the expected length of the entire message.

5.5.2 Typed Component Provider

The Typed Component SPP Provider extends the Standard Provider by using a tagging scheme, as proposed in [30]. In this implementation, a single byte is prepended to each message component. The byte contains a value that indicates the type of component. Should the type indicator of a received message component not match the expected type of the component, the provider implementation throws an exception, causing the execution of the protocol implementation to halt.

This prevents an attacker from successfully perpetrating a type flaw attack, by attempting fool a principal into interpreting a message component of one type, as one of another type. Heather et al show in [30] that a tagging scheme, such as the one implemented by this provider, is sufficient to prevent type flaw exploitations.

5.5.3 Simulation Provider

We have implemented an SPP provider for running simulation components. The details of this provider are discussed in chapter 7. The purpose of the Simulation SPP Provider is to provide an environment in which a protocol run, possibly including attempted attacker roles, can be executed and controlled by the user.

5.6 Discussion

By decoupling the security protocol logic implementation from the cryptographic and communication implementation, the SPP API allows the issue of the correctness of these two components to be addressed separately.

As SPP provides a comprehensive set of cryptographic and communications functionality, it can also be used by security protocol implementations that are manually coded or generated by tools other than Spi2Java.

Chapter 6

Spi2Java: Code Generation from Specifications

In this chapter we cover the design and implementation of the Spi2Java code generator. We give an overview of Spi2Java and how the code it generates works, discuss our choice of source language (Spi Calculus) and target language (Java). We also describe our approach to, and development of, a mapping from Spi Calculus constructs to Java code. The implementation of the mapping in Prolog is covered. Finally we evaluate Spi2Java against the criteria for high integrity code generation.

6.1 Overview

Spi2Java is a code generator that takes a network security protocol specified in the Spi Calculus as input, and generates Java code implementing the protocol as output. It is essentially a Spi Calculus to Java compiler.

Spi2Java generates code by applying a specified mapping from Spi language constructs to Java code segments. It is implemented in Prolog, so that its compiling specification is very close to the implementation.

The generated Java code defines a class that extends the predefined `sprite.spi.Process` class. The generated class accesses cryptographic and network communications functionality via calls to the SPP API, either directly or indirectly via methods inherited from its parent class. The SPP API abstracts implementation specifics from the generated code. It allows different pluggable providers,

that implement their own choice of network communications protocols and cryptographic algorithms, to be used without requiring changes to the generated code.

This architecture is outlined in figure 10: *Generated Class* represents the generated code that makes function calls to the *SPP API*, either directly or through its parent *Process Class*. The SPP API then routes these calls to the currently installed provider, that may use libraries such as the Java Cryptography Extensions (*JCE*) or General Security Services API (*GSS API*) to implement cryptographic services.

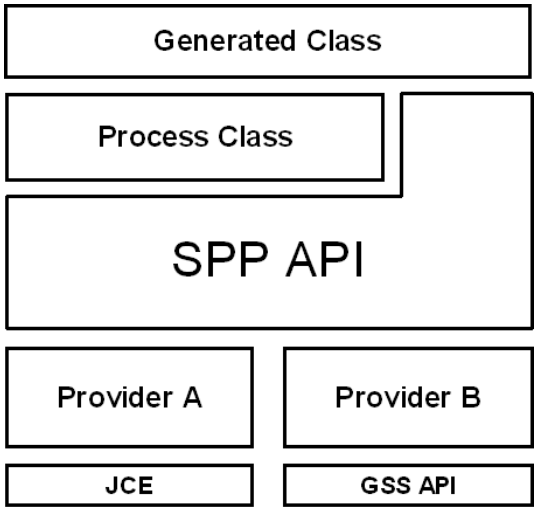

Figure 10: Generated class code relies on the Process class, the SPP API and SPP provider implementations.

6.2 Source and Target Languages

6.2.1 Source Language: Spi Calculus

The Spi Calculus, with the modifications we have defined, is the input language. Its properties, which make it a suitable formalism for security protocol specification and analysis, and its suitability for input to a code generation tool, are covered in detail in chapter 3.

6.2.2 Target Language: Java

Core Language Properties

The Java language, in combination with the Java Virtual Machine, has many properties that make programming in Java less error prone than some traditional languages. Although it has large well defined set of APIs associated with it, the actual Java language itself is very small and relatively simple.

Java has memory management in the form of garbage collection, freeing the programmer from having to explicitly free memory allocated on the heap. This prevents memory leaks in most, though not all, situations.

Bounds checking on array access avoids the possibility of a Java program accessing memory that is not in its address space or that does not belong to the array being traversed. Bounds checking negates the whole class of buffer overflow attacks that have plagued many network applications.

Java is a strongly typed language. Research has shown that Java's type safety holds: an object cannot be cast to type, or an expression cannot be evaluated to a type class to which it does not belong or does not inherit from without an exception being raised by the system [62, 25]. Static type checking detects many errors at compile time that would otherwise only be encountered at runtime.

Java APIs for Networking and Security

Java's large set of well defined APIs includes libraries for network communication and cryptography. The networking API exposes consistent and easy way to use TCP/IP and UDP/IP communication on all the supported platforms. Similarly, the Java Cryptography Extension (JCE) API allows multiple provider implementations to be used. This model is particularly beneficial as if the cryptographic provider used by a security application is found to be flawed, it can be swapped with a (hopefully correct) provider from another vendor without changing application code.

Portability and Platform Independence

The Java language is portable across a variety of hardware architectures and operating systems. Its associated APIs are largely platform independent, though behaviour of some of its interfaces may vary slightly.

Formal Semantics

The Java language is officially defined by the Java Language Specification. Although this does not include a formal definition of the semantics of the language, a number of research publications have defined a formal model of the language, or at least for significant subsets thereof. The formal approaches to specifying Java have included behavioural definitions using Gurevich's Abstract State Machines [8], process algebras such as π-calculus [35] and CSP [46] and a Hoare type calculus [62].

Widely Used

The Java language is widely used with implementations from a number of commercial and open source vendors. It has also been used as the target language for many code generation projects - including other tools that have been used to generate security protocol implementations [50, 23, 18].

6.3 Code Generation

6.3.1 Approach

Some approaches to implementing process algebras in Java [14, 38, 42] have encapsulated language constructs in individual classes. They provide the high level of abstraction necessary for implementing concurrent systems in general, where parallel communicating processes may be operating locally or on different machines, and synchronising and communicating over a network or a built in language feature (e.g. Java's synchronisation). As the Spi Calculus is used to specify network security protocols, the process behaviour is more restricted, allowing a simpler approach to implementation.

In our approach each Spi processes that specifies a role in a security protocol, is implemented as an instance method on a Java class. These methods are publicly exposed, providing other Java software applications access to an implementation of the security protocol. Spi variables are mapped directly to Java variables that can be assigned that value of Java expressions that correspond to closed Spi terms. Spi names - such as channels, nonces and keys - are implemented as classes from the SPP API, discussed previously. For example, a Spi Calculus nonce variable, $x : Nonce$, is implemented in Java by the declaration `sprite.spp.term.Nonce x`.

Spi processes that specify individual security protocol roles do not contain parallel components. However, in to order specify a run of a protocol, a process containing parallel components - which will be the Spi processes that specify the individual protocol roles - are required. We have implemented

parallelism in two ways: *explicitly* and *implicitly*. The explicit implementation spawns new threads to execute the Java code that implements each of the parallel components of a Spi process. This allows code to be generated that defines the entire system of a protocol - its participant roles and attacks - making a simulated run of the system possible in a single process space.

The implicit "implementation" relies on the fact that in practice, the participant role implementations will be run in separate process spaces on separate machines, and as such do not require any extra Java code to be emitted.

6.3.2 Generated Code Structure

Spi2Java implements a security protocol by generating a class, named after the protocol, with the Spi processes that specify each protocol role implemented as an instance method on this class. The generated class extends the `sprite.spi.Process` class that contains a set of protected instance of cryptographic and network functionality. These methods, listed in figure 11, are short and simple and serve to make the Spi2Java generated code cleaner and easier to read. None of these methods contain `try-catch-finally` blocks, ensuring that any exception thrown by the SPP API propagate up to their caller.

`protected final InputStream decrypt(Encryption encryption, Key k)`	Decryption.
`protected final Nonce newNonce()`	Instantiate a new nonce.
`protected final Timestamp newTimestamp()`	Instantiate a timestamp.
`protected final InputStream recv(Channel c)`	Receive a message.
`protected final void send(Channel c, Term m)`	Send a message.
`protected final boolean valid(Timestamp t)`	Validate a timestamp.
`protected final Encryption encrypt(Term term, Key k)`	Encryption.
`protected final Term hash(Term m)`	Generate a message digest..
`protected final Term pair(Term m, Term n)`	Construct a pair.
`protected final Key priv(Identifier i)`	Return private key.
`protected final Key pub(Identifier i)`	Return public key.
`protected final TermFactory getTermFactory()`	Return the provider's factory class.
`protected final Trace newTrace(String name)`	Instantiate a trace object.

Figure 11: The instance method exposed by the Process base class.

An example generated class for a protocol P, with initiator and responder roles defined by the Spi processes $Init(x1 : Type1, x2 : Type2, x3 : Type3, ...)$ and $Resp(x1 : Type1, x2 : Type2, x3 : Type3, ...)$ respectively, is listed in figure 12.

As each Spi process definition is emitted as an instance method on the generated class, the protocol roles are all accessible from a single class. Invocations of these methods are stateless, and hence

```
package protocol;

import java.io.*;
import sprite.spi.*;
import sprite.spp.*;
import sprite.spp.net.*;
import sprite.spp.term.*;

public class P extends sprite.spi.Process
{
   public P(Provider provider)
   {
      super(provider);
   }

   public void Init(Type1 x1, Type2 x2, Type3 x3, ...)
   {
      // Code implementing Init...
   }

   public void Resp(Type1 x1, Type2 x2, Type3 x3, ...)
   {
      // Code implementing Resp...
   }
}
```

Figure 12: Example generated class for protocol P with initiator and responder roles.

thread safe, allowing a single class instance to be shared by all threads executing protocol roles, in the case of a simulated protocol run in a single process space.

6.4 Mapping Spi to Java

6.4.1 Approach to Program Refinement

For Spi2Java, generating code to implement a security protocol, is essentially compiling a Spi Calculus process to Java code. In this section we develop a mapping from Spi Calculus process actions to Java code segments. Ideally we would like to formally prove that semantics of the Spi Calculus process actions is preserved in each Java code segment that they are mapped to.

The approach to this is generally to map the specification (Spi process action) data to the implementation (Java code segment) data by means of a data mapping function. It must then be shown that the implementation code leaves the implementation data in a state consistent with the corresponding specification data as defined by the semantics of the specification process action. This involves using the semantics of the specification to define pre and post-conditions on the implementation data. The post-condition must hold after the execution of the implementation, provided the pre-condition held

before.

In developing the mapping definition from Spi to Java, we have not provided rigorous formal proofs that Java code preserves the semantics of Spi. We have, however, made informal arguments where feasible, using both transition semantics for Spi and Borger et al's Abstract State Machine (ASM) semantics for Java [8], that each Java code segment preserves the semantics of its corresponding Spi process action.

We use the transition semantics for Spi to define pre and post-conditions on Spi data for some of the Spi process actions. These conditions are then applied to the corresponding Java data variables - which are determined by a trivial data mapping described below. We then show that, according to the ASM semantics for the Java statements that implement the process actions, the post-condition will hold after the Java code is executed.

6.4.2 Data Mapping Function

As in [10], we view compilation as a relation between the source program and the implementing code. In this case the source program is the Spi process s and the implementing code is the Java code j. A symbol table is required to map the elements of the data space operated on by s to those operated on by j.

We define the symbol table by the injective function f that maps elements of the Spi data space to those of the Java data space, and its inverse f^{-1} which does the reverse. For example, if x_{Spi}, y_{Spi} and z_{Spi} are Spi terms that correspond to the Java expressions x_{Java}, y_{Java} and z_{Java} respectively, we define f as:

$$f(x_{Spi}) = x_{Java}$$

$$f(y_{Spi}) = y_{Java}$$

$$f(z_{Spi}) = z_{Java}$$

and its inverse f^{-1} as:

$$f^{-1}(x_{Java}) = x_{Spi}$$

$$f^{-1}(y_{Java}) = y_{Spi}$$

$$f^{-1}(z_{Java}) = z_{Spi}$$

For convenience, we shall often denote $f(x)$ in the Java code listings that follow, by just referring to it as `x` using the font used for all Java code listings.

6.4.3 Type Mapping

Likewise we define a mapping from Spi types to Java types in table 2. The Java types have the same names as the Spi type bar being pre-fixed by a packaged name.

Spi Calculus Type	**Java Type**
Channel	sprite.spp.net.Channel
Encryption	sprite.spp.term.Encryption
Hash	sprite.spp.term.Hash
Identifer	sprite.spp.term.Identifier
Key	sprite.spp.term.Key
Nonce	sprite.spp.term.Nonce
Pair	sprite.spp.term.Pair
Timestamp	sprite.spp.term.Timestamp
UserData	sprite.spp.term.UserData

Table 2: The instance method exposed by the Process base class.

Again, for convenience, we shall use Spi and Java types interchangeably, using the different fonts for Spi and Java listing to indicate context.

6.4.4 Mapping Definitions

In this section we develop a mapping for each Spi Calculus process action to a Java code segment or *code template*. The code templates are parameterised by

- Java variables that correspond to Spi variables affected by the Spi process action,
- Java types that correspond to Spi types and
- Java expressions that correspond Spi terms.

The implementation of the mapping, is responsible for setting the parameters to instantiate the templated code.

We define code templates for the following Spi process actions:

- Restriction (nonce and timestamp generation) - $(n)P$

- The nil process - nil
- Input - $c?(M).P$
- Output - $c? < M > .P$
- Pair/tuple splitting - $let\,(x_1, ..., x_i) = M\,in\,P$
- Decryption - $case\,M\,of\,\{x\}\,in\,P$
- Timestamp validation - $case\,x\,valid\,in\,P$

To develop the Java implementations for these actions, we first define the Java implementations for two basic Spi Calculus behaviours: the behaviour of a process that is stuck and a variable binding or substitution. After defining implementations for these behaviours we move on to the process actions above.

Stuck Process Behaviour

A number of Spi processes are defined to terminate under certain circumstances. They differ from the *nil* process, which also terminates, in that they indicate the process has not terminated successfully. When such circumstances arise, the defined process behaviour is to do nothing, i.e. become *stuck*. In the context of a Spi process that specifies a security protocol, becoming stuck is generally an indication that something has gone wrong with the run of the protocol. For example: the match process $[MisN]P$ only behaves as P if M matches N, otherwise it is stuck. This may be because the value of a received message component was not equal to the expected value, possibly indicating an attempted attack on the protocol.

In our mapping, the behaviour of a stuck process is implemented by throwing an exception. Given that Spi2Java only emits `try-catch-finally` statements when it instantiates a new thread to implement a parallel component, exceptions thrown in a method the implements a Spi process that specifies a protocol role will bubble up to the caller of that method. This results in some desired properties:

1. No further statements, or method calls, in the current method are executed. Thus execution of the protocol role implementation terminates.
2. Control is passed to the calling method.
3. Information about the cause of the termination can be passed to the caller for logging.

Certain conditions that can cause a process implementation to become stuck - for instance, pair splitting where the underlying raw data has become corrupt - can only be detected by the SPP Provider code. This code it is allowed to throw an exception, which - because there are no `try-catch-finally` statements generated method that calls it - has the same effect as generated method throwing an exception: That is, it causes the process implementation to terminate.

This implementation behaviour is refinement of the Spi specification behaviour: it can - potentially - halt more often, but always halts under the same circumstances as the Spi specification

Variable Binding or Substitution

In the Spi Calculus, an action α, preceding a process P, may contain a *binding occurrence* of a variable, or perhaps multiple variables - as in the case of the pair splitting action. The syntax of such an action introduces a new variable that references a location containing a value in P. The variables act as a place holders and are essentially substituted with a value as a result of the execution of the action, causing all occurrences of the variables in the process to reference that value.

Formally the behaviour of binding actions can be generalised by the transition semantics:

$$\frac{-}{\alpha P \xrightarrow{\alpha} P[\vec{N}/\vec{x}]}\ \vec{x} \in bn(\alpha)$$

This is read as meaning that the process αP can progress to behave as process P, with all free occurrences of the variables in $\vec{x}$ substituted with their corresponding values in $\vec{N}$, after the action α has occurred. The input process $c?(x).P$ is an example of this: the input action reads the term N from channel c then the process can proceed as P with all free occurrences of x replaced with N, i.e. as $P[N/x]$.

Given the pre-condition that $f(x)$ is unassigned, i.e. `x` is `null`, the Java implementation of $P[N/x]$ must satisfy the post-condition $f(x) = f(N)$, i.e. the Java variable `x` must be assigned the result of evaluating the Java expression `N`, or the execution of the process must terminate.

The intuitive way to implement the substitution $P[N/x]$ in Java is by assignment. The ASM definition for Java assignment, from [8], is:

if $(task$ **is** $var = exp)$ **then**
    $loc(var) := val(exp)$
    $val(task) := val(exp)$

proceed

The dynamic function *loc* maps local variables to values while *var* maps an expression to a value. We derive the Java code template to implement the substitution $P[N/x]$ by replacing *var* with the Java variable `x`, which corresponds to the Spi variable x, i.e. it is $f(x)$, and *exp* with the Java expression `N` which corresponds to the Spi term N, i.e. it is $f(N)$. According the definition the assignment task updates *loc* at `x` to the value of *val* at `N`, i.e. $loc(\texttt{x}) := val(\texttt{N})$, which is exactly the post-condition. If an exception is thrown during the evaluation of `N`, we get the stuck process behaviour defined above, which terminates the process as desired.

Thus we define $P[N/x]$ to map to the Java code template in figure 13. The code template is parameterised by the type declaration `Type`, the variable name `x`, and the expression `N`. Spi type of x.

```
final Type x = N;
if (x == null) throw new AssignmentException();
// Code implementing P...
```

Figure 13: Implementation of substitution or variable binding.

Accurately implementing Spi substitution requires that the Java code template ensures `x` cannot be re-assigned by the Java code that implements P. Java's *final* variable declaration modifier is used (see line 1 of the code template) to ensure that the variable can only be assigned to once, as defined by the JLS [34]. Because most formal definitions of Java semantics describe its dynamic behaviour, they don't cover `final` variable declaration and assignment. We have to rely on the JLS's informal description that states that a `final` variable may only be assigned to once [34]. Violations of this constraint are intended to be caught by the compiler and hence are a static concern.

The evaluation of the Java expression `N`, will generally involve a call to an SPP provider function. Because provider code is relied on there is no guarantee `N` will not evaluate to `null`, instead of a valid `Type` instance. To mitigate this risk we add an assertion (line 2 of the code template) that will ensure `x` is a assigned a non-`null` value or else throw an exception.

Throwing an exception in such circumstances ensures the code is *fail fast*: the code fails immediately and non-deterministic behaviour is not introduced by a `NullPointerException` possibly being thrown at an unknown later point in the code, when `x` is accessed. As no subsequent method statements are executed, the Spi semantics are preserved: the code that implements P is not executed and again, the behaviour of the stuck process is implemented.

In the cases where an exception is thrown, either by the SPP provider or because `N` evaluated to `null`, the implementation of the Spi process αP will not progress to become $P[N/x]$, instead it will halt. Thus the process implemented by the code template is actually $P[N/x] + nil$, it behaves either as $P[N/x]$ or the nil process. Fortunately the process $P + nil$ is a refinement or simulation of the process P, as shown in [51] and by the fact that $P + nil$ must terminate as frequently or more than P.

Nil

The nil process action can be explicitly written as the final action of a Spi Calculus process, indicating that it has successfully finished. It can also be omitted, in which case it is implied. Its behaviour is to do nothing. The Java code that implements a Spi process definition is emitted by Spi2Java in a single *instance method* that has a `void` return type. Thus the behaviour of the nil process is exactly implemented by the `return` statement at the end of the method, which like nil can be explicitly listed at the end of the method, or omitted but still implicit.

Process Definition

The transition semantics for a π-calculus *process definition*, $A(\vec{y}) = P$, are given in [51]:

$$\frac{P[\vec{y}/\vec{x}] \xrightarrow{\alpha} P'}{A(\vec{y}) \xrightarrow{\alpha} P'} A(\vec{x}) = P$$

This says that provided $P[\vec{y}/\vec{x}]$, i.e. P with the elements of $\vec{y}$ substituted for the corresponding elements of $\vec{x}$, can progress to P', then $A(\vec{y})$ can to - with the obvious side condition that $A(\vec{x}) = P$.

As we intend to implement Spi Calculus process definitions with instance method calls, we cannot use the Java assignment implementation for substitution defined previously. Instead, the method call implements the substitution in much the same way. We thus implement Spi process definition of the form $A(x1 : Type1, x2 : Type2, x3 : Type3...) = P$ with the code template in figure 14. The Java instance method parameters and their types, are exactly the Java variables that correspond to the Spi variables that are A's parameters, hence the code template is parameterised by the types `Type1`, ..., `TypeN` and variable names `x1`, ..., `xN`.

```
public void A(Type1 x1, ..., TypeN xN)
{
       // Code implementing P...
}
```

Figure 14: Implementation of process definition $A(x1 : Type1, x2 : Type2, x3 : Type3, ...) = P$.

Input

The following transition rule defines the behaviour of the Spi input action:

$$\frac{-}{x?(y).P \xrightarrow{x?(M)} P[M/y]} M \notin fn((y)P)$$

Intuitively this means that the Java code template that implements the input action $x?(y)$, should block until it reads some data that corresponds to the Spi term M from a network communication endpoint that corresponds to the Spi channel x. This data should then be assigned to the Java variable that corresponds to y.

To implement the input action we employ our generalised definition for implementing substitution. The only specialisation required is to substitute the Java expression `N` from the implementation definition, with a method call that will return data corresponding to M, read from the communication endpoint corresponding to channel c. An instance method `InputStream recv(x)` on the `sprite.spi.Process` class handles calling the underlying provider implementation and returning the data corresponding to M in an `java.io.InputStream` instance. The resulting code template is shown in figure 15 and is parameterised by the variable name `y` and the `Channel` variable `x`.

```
final InputSream y = recv(x);
if (y == null) throw new AssignmentException();
```

Figure 15: Implementation of the input process $x?(y).P$.

As we don't have control of the provider code that is called by `recv(x)`, we assume that it returns the data or throws and exception if it encounters an error.

The pre and post-condition are as for the substitution code template: that is, `y` is `null`, and `y` equals the value returned by the evaluation of `recv(x)` or the process must terminate. As the same semantic

definition is used as for substitution, we are assured that the post-condition holds.

Restriction

The π-calculus defines restriction, (y)P, to have the following transition semantics:

$$\frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \, y \notin n(\alpha)$$

meaning that if the process P can evolve into the process P' after the execution of some action α, then the process P preceded by the restriction (y) can progress to P' given that y is not a name in α.

Again we specialise the defined implementation for substitution; this time replacing the Java expression `N` with a call to `Nonce Process.newNonce()` to get the code template in figure 16 which is parameterised by the variable name `y`.

```
final Nonce y = newNonce();
// Code implementing P...
```

Figure 16: Implementation of the restriction or nonce generation process $(y)P$.

The `Nonce newNonce()` instance method handles the call to the provider code and returns a `Nonce` instance that encapsulates (pseudo) randomly generated data that is suitable for use as a nonce.

The implementation of the `Nonce newNonce()` method is simply:

```
protected final Nonce newNonce()
{
       return getTermFactory().newNonce();
}
```

It exists only to make the Spi2Java generated code cleaner and easier to read.

Output

The output process does not result in the binding of any variables, the semantics for the process simply dictate that an output action must occur:

$$\frac{-}{x! < M > .P \xrightarrow{x!<M>} P}$$

Output is implemented by calling `void Process.send(Channel c, Term m, Trace t)`. The first two method parameters correspond to the Spi channel c and term M, i.e. they are $f(c)$ and $f(M)$ respectively. The last parameter is for tracing purposes. It allows a `Trace` instance to be provided so that the data being sent can be logged. The implementation of output action is just the single line code template in figure 17.

```
send(Channel x, Term m, Trace t);
// Code implementing P...
```

Figure 17: Implementation of the output process $x! < M > P$.

Pair Splitting

The Spi pair splitting process, $let\,(x,\, y) = M\, in\, P$, is implemented by extracting the raw data from the Java *spp.Term* instance corresponding to the Spi term M. The two terms, x and y, that M is to be split into are then unpacked from this data. To make it easier for the provider implementation to pack and unpack terms from raw data, we introduce a restriction on creating and splitting pairs that states that the first term must always be a name or name variable (i.e. it must be an atomic value). This restriction means that when the provider code packs and unpacks terms to send and receive over the communications network, it does not need to store extra information about the structure of the pairs - which may be nested to arbitrary depth, e.g. the message $\{A, B, C\}\, pub(A)$ would be specified $\{(A, (B, C))\}\, pub(A)$ given that A and B are names or name variables.

Apart from simplicity, this restriction has the advantage of making it possible to implement providers that are message compatible with existing security protocol implementations, as such implementations are unlikely to use pairing to structure their message data.

The Java code template for this process is shown in figure 18, where `TypeX` and `TypeY` are the types of the variables `x` and `y` respectively, and `_is` is a temporary variable that is only in the scope of the

block defined by the { and } characters on lines 4 and 10. Using a block allows the re-use of the same temporary variable name for multiple instances of the code template. The alternative would be for Spi2Java to generate a new temporary name each time it emits this code template, adding unnecessary complexity to its implementation.

```
1
2    final TypeX x;
3    final TypeY y;
4    {
5       InputStream _is = new ByteArrayInputStream(j.getData());
6       x = getTermFactory().unpackTypeX(_is);
7       if (x == null) throw new AssignmentException("x", null);
8       y = getTermFactory().unpackTypeY(_is);
9       if (y == null) throw new AssignmentException("m", null);
10   }
11  // Code implementing P...
12
```

Figure 18: Implementation of the pair/tuple splitting process $let\,(x,\,y)\,=\,M\,in\,P$.

Decryption

The decryption process, $case\,L\,of\,\{x\}N\,in\,P$, maps to a call to `InputStream spi.Protocol.decrypt(spp.Encryption, spp.Key)`. This call will propagate down to the `byte[] spp.Key.decrypt(byte[])` method that is implemented by the provider, according to encryption algorithm associated with the type of key i.e. either symmetric, public or private.

```
final TypeX x = getImpl().unpackTypeX( decrypt(L, N));
if (x == null) throw new AssignmentException();
// Code implementing P...
```

Figure 19: Implementation of decryption.

Timestamp Validation

The timestamp validation process, $case\,x\,valid\,in\,P$, is implemented by the code template in figure 20, which is parameterised by the variable name `x`.

In this case a `StuckException` is thrown, to differentiate the reason for the process becoming stuck from that of an assignment to a `null` value.

```
if (!valid(x))
{
   throw new StuckException();
}

// Code for P...
```

Figure 20: Implementation for timestamp validation *case x valid in P*

Match

The behaviour of the match process, $[M\,is\,N]P$, is to progress as P provided the value of the terms M and N is the same, otherwise the process is stuck. The implementation of the matching test, is implemented by the SPP provider, which knows how to compare the concrete implementation of message components. The `boolean match(Term t)` method on the `sprite.spp.term.Term` class, is defined to return a boolean indicating whether or the parameter `t` matches the instance the method is being called on. If the method returns `false`, i.e. the terms do not match, then the implementation must exhibit the stuck process behaviour. The process is thus implemented in Java by statement to throw an exception, implementing the stuck behaviour, guarded by an if statement that checks that the components don't match. Figure 21 lists the implementation of the process.

```
if (!x.match(n))
{
   throw new StuckException();
}

// Code for P
```

Figure 21: Implementation for match $[MisN]P$

6.5 Traceability

6.5.1 Compile Time Tracing

Spi2Java emits comments in the generated Java code, reflecting the Spi process actions that each Java code segment implements. Apart from making the generated code easier to read and understand, it allows each segment of generated code to be traced back to the original Spi specification. The trace comments are emitted by the parsing rule for each Spi process action and term, and are also concatenated to rebuild the original Spi process specification as a trace comment. This comment is

emitted as documentation for the Java method that implements it, providing not only documentation of the intended behaviour of the method, a but a compiler trace that can be validated against the original specification.

6.5.2 Runtime Tracing

In addition to the Java code that implements the Spi actions, Spi2Java also emits code to trace the protocol progress and state. This is done by making calls to a provider tracing implementation - which can be a simple log file or user interface displaying the state of the protocol run as messages are sent and received.

At the beginning of each generated method that implements a process definition $A(X_1, ..., X_i) = P$, a `sprite.spp.Trace` instance is retrieved from the provider and the name of the current thread is set:

```
final Trace _trace = newTrace("A(X1, ..., Xi");
Thread.currentThread().setName("A(X1, ...,Xi");
```

This trace instance is used to trace both the progression of the protocol run in relation to Spi specification, and the variable bindings that have occurred. For example, Spi2Java emits the following code for instantiating a new nonce n:

```
// (n)
_trace.traceSpi("(n)");
final Nonce n = newNonce();
if (n == null) throw new AssignmentException("n", null);
_trace.update("n", n);
```

In this example the first line is a Java comment indicating what Spi action is implemented by the following code segment. The second line makes a call to the trace object to specify that the restriction Spi action is about to executed. The code on the third line actually implements the action, and the an assertion is made on the 4th line, while the final line calls the trace object to update the protocol state with the value bound to the nonce n.

6.5.3 Stepping through a Process Run

The provider's implementation of the `sprite.spp.Trace` may, optionally, allow the user to step through a run of a protocol at the Spi process action granularity level.

The Java code segments that implement each action, always make a call to `sprite.spp.Trace.trace(String)`, prior to executing the code that implements the action. Because the trace instance is supplied by the provider, the provider implementation of the `sprite.spp.Trace.trace(String)` method can block until it has received input from the user. Once the user response has been received, the call can return and the run of the protocol can progress.

6.6 Implementation in Prolog

6.6.1 The Spi Lexer

Spi2Java lexer component defines a set of Prolog rules to implement a tokeniser that reads the plain text input file and breaks it into valid tokens. The tokens consist of keywords and valid variable identifiers and are compiled into a list that is passed to the parser component

6.6.2 Definite Clause Grammar

Spi2Java is implemented in SWI-Prolog using the Definite Clause Grammar (DCG) rules supported by most Prolog engines [65]. The DCG provides syntactic sugar, allowing grammars to be specified in a easier to read and more intuitive way. It allows parsing rules to specified without having to explicitly declare a list of tokens in the rules' parameters.

6.6.3 The Spi Parser

The Spi2Java parser component defines the Spi grammar using Prolog DCG rules in figure 22. This figure omits the code generation body of the rules. The complete rules are available at [5].

The rules are declared with extra uninitialised parameters to which output data, i.e. generated code and trace comments, can be bound to. The body of the rules defines a code segment template, into which the relevant variable names are inserted. Figure 23 shows the rule defining the template for restriction.

```
/* A(X, ..., Y)*/
process --> [var(A)], [left_par], variables, [right_par].

/*( P | Q )*/
process --> [left_par], process, [par_cmp], process, [right_par],

/* (A)P */
process --> [left_par], [var], [colon], [var], [right_par], process.

/* let (X, Y) = M in P */
process -->
[let], [left_par], [var], [colon], [var], [comma], [var], [colon], [var], [right_par], [equals], [var], [in], process.

/* let (x_1, ..., x_N) = M in P */
process --> [let], [left_par], [var], [colon], [var], [comma], let_tuple, [right_par], [equals], [var], [in], process.

let_tuple --> [var], [colon], [var], [comma], let_tuple.

let_tuple --> [var], [colon], [var].

/* [X is Y]P */
process --> [left_square], term, [is], term, [right_square], process.

/* case L of {X}K in P */
process --> [case], term, [of], [left_curly], [var], [colon], [var], [right_curly], term, [in], process.

/* case L is valid in P */
process --> [case], term, [is], [valid], [in], process.

/* nil */
process --> [nil].

/* C ? (A).P */
process --> [var], [in], [left_par], [var], [colon], [var], [right_par], [fullstop], process.

/* C ! <M>.P*/
process --> [var], [out], [left_angle], term, [right_angle], [fullstop], process.
```

Figure 22: Spi2Java's Prolog DCG rules defining the Spi Calculus process grammar.

```
/* (A)P */
process(InPre, OutCode, OutTrace) -->
[left_par], [var(A)], [colon], [var(Type)], [right_par],
   process(InPre, PCode, PTrace),
{
   type(Type),
   strings_concat(['(', A, ')'], Trace),
   strings_concat([Trace, '\n\t\t// ', PTrace], OutTrace),
   strings_concat([InPre, '_trace.trace("', Trace, '");\n'], PrintTrace),
   strings_concat([InPre, '_trace.update("', A, '", ', A, ');\n\n'], State),
   strings_concat([InPre, '// ', Trace, '\n'], Comment),
   strings_concat([
      Comment,
      PrintTrace,
      InPre, 'final ', Type, ' ', A, ' = new', Type, '();\n',
      InPre, 'if (', A,
         ' == null) throw new AssignmentException("', A, '", null);\n',
      State,
      PCode],
      OutCode)
}.
```

Figure 23: The DCG rule for parsing restriction and generating the implementing Java code.

6.7 Implementation Correctness

Presuming the correctness of the Spi to Java mapping, the next concern is that the mappings are correctly implemented by the Spi2Java code generator. This concern corresponds to the third requirement identified by Whalen and Heimdahl for high integrity code generation is that "Rigorous arguments must be provided to validate the translator and/or the generated code" [64, Page 4].

Using Prolog does not in and of itself provide a proof of correctness of the translator software (Spi2Java) and hence meet this goal. However, given that in the development of Spi2Java the specification of the mapping from Spi to Java was essentially defined using Prolog rules, we can be confident (at least as much as our faith in the Prolog engine allows), that Spi2Java preserves those mappings.

6.8 Example Implementation Using Spi2Java

In this section Spi2Java's code generation abilities are demonstrated by way of example. Implementations of the Needham-Schroeder-Lowe protocol's initiator and responder roles are generated. The roles are specified as Spi processes and the specifications input into Spi2Java. The provider implementation used is the type flaw detection one, that uses TCP/IP for network communications and RSA, AES and SHA algorithms for asymmetric, symmetric and message digest cryptography respectively. The resulting Java programs are run on separate machines, connected via a local area network, and the output examined.

6.8.1 Implementing The Needham-Schroeder-Lowe Protocol

The legitimate roles of Needham-Schroeder-Lowe Public Authentication Protocol are specified as the Spi processes $Init$,and $Resp$ in figure 24.

Both the roles have process definition parameterised by a channel c. This channel represents an endpoint for communication on a public network and thus may not be viewed as secure by the principals. There is no guarantee that they are both instantiated with the same channel c when they are executed. They could each be passed different channels, which may possessed by an attacker, who would thus have control over their communications. We will elaborate further on modelling an attacker in this manner in the next chapter, where we discuss the implementation of the flawed Needham-Schroeder protocol and Lowe's attack on it.

```
Init(c:Channel, A:Identifier, B:Identifier) =
        (n:Nonce)
        c!<{(n, A)}pub(B)>.
        c?(l:Encryption).
        case l of {j:Pair}priv(A) in
        let (x:Nonce, p:Pair) = j in
        let (m:Nonce, y:Identifier) = p in
        [x is n]
        [y is B]
        c!<{m}pub(B)>.
        InitF(c, n, m)

Resp(c:Channel, B:Identifier) =
        c?(l:Encryption).
        case l of {j:Pair}priv(B) in
        let (n:Nonce, A:Identifier) = j in
        (m:Nonce)c!<{(n, (m, B))}pub(A)>.
        c?(p:Encryption).
        case p of {x:Nonce}priv(B) in
        [x is m]
        RespF(c, n, m)

abstract InitF(c:Channel, n:Nonce, m:Nonce)

abstract RespF(c:Channel, n:Nonce, m:Nonce)
```

Figure 24: Spi specification of the Needham-Schroeder-Lowe protocol initiator and responder roles.

The specification demonstrates explicit pair splitting, as opposed to using the shorthand for tuples, and also lists $InitF$ and $RespF$, the user implementable abstract process definitions. These abstract definitions provide a mechanism for the user integrate their code with the generated code. For example, the user may wish to use the Spi2Java generated NSL implementation to establish a secure session. Once the session has been established, the $InitF$ and $Resp$ implementations are run. They are called with three parameters, the two secret nonces generated during the session - potentially for use as a session key, and a handle to the shared channel for further communication.

6.9 Discussion

The evaluation of Spi2Java against the high integrity code generation requirements in section 2.3.1 is deferred to the results chapter. Briefly, requirements 1, 3 and 5 are largely met. Requirement 2 is not completely met as the arguments we provide for the semantic consistency of the translation from Spi constructs to Java code are not strictly formal. Requirement 5 is not met in any meaningful sense, due to the limited resources available for exhaustive testing.

To put this in perspective, it is notable that formally verifying translator software is not, at least currently, a completely attainable goal. Doing so would require a verified programming language in which to implement the translator software, a verified compiler to compile the software to verifiable machine code, making calls to verified libraries, with a verified operating system, all running on a verified hardware architecture implementation [64, 10]

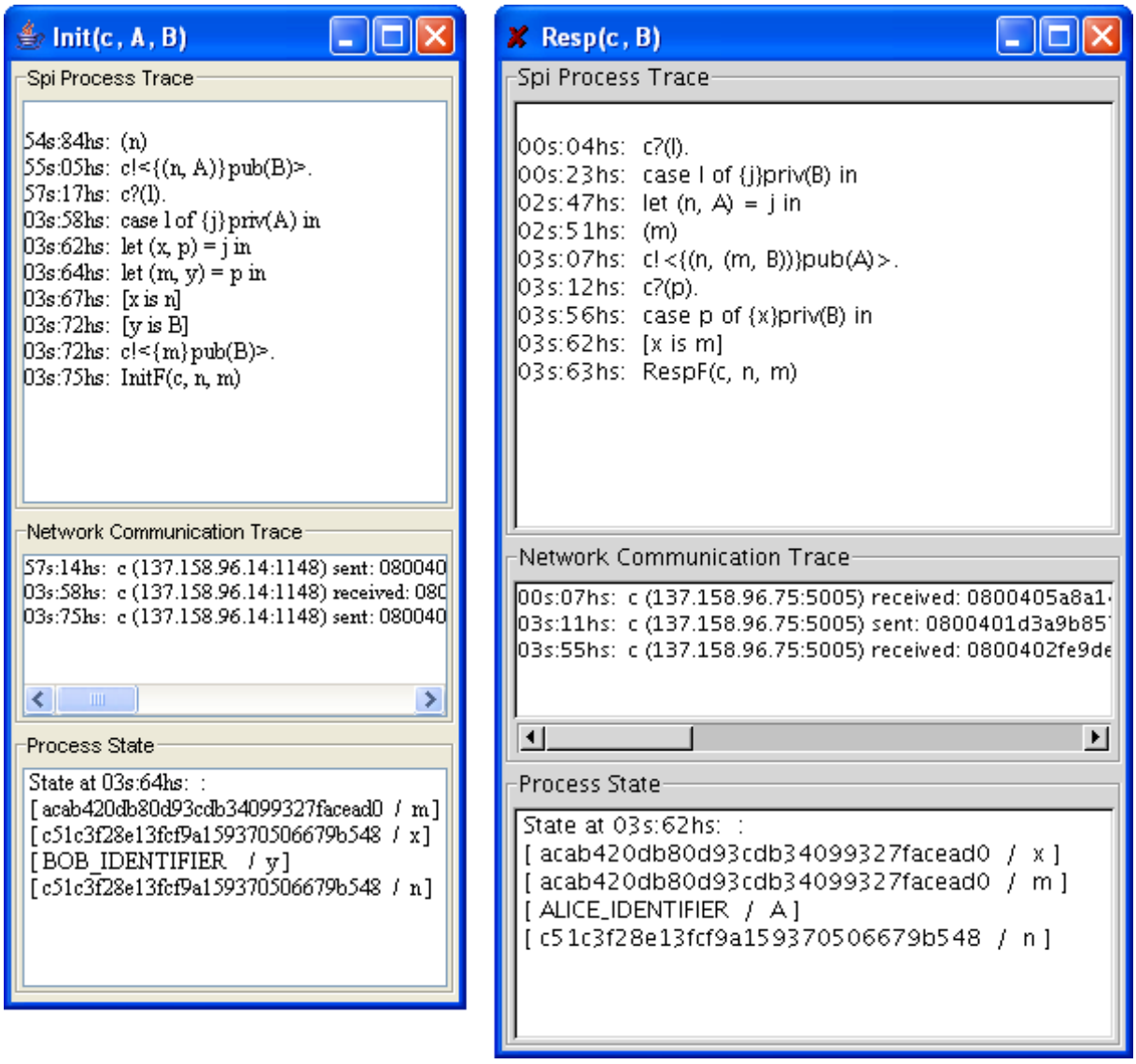

Figure 25: Run of Spi2Java generated implementations of the Needham-Schroeder-Lowe initiator and responder roles on separate machines.

Chapter 7

Sprite

In this chapter we briefly describe our Security Protocol Implementation Tool and Environment (SPrITE). The *tool* component is a simple graphical user interface for editing specifications and running the Sn2Spi translator and Spi2Java code generator. The *environment* component consists of an SPP provider that we have developed that, when used by Spi2Java generated security protocol implementations, allows the user to simulate a run of a security protocol in a controlled environment. This SPP provider exposes a graphical user interface that can be used to control the run of the protocol simulation.

To illustrate the use of the Sprite environment, we simulate a run of the Needham-Schroeder Protocol. The simulated run involves both the usual initiator and responder roles, as well as an attacker role that executes Lowe's attack on the protocol. The run is specified as a Spi Calculus process composed of three parallel components, each of which specify one of the protocol roles. We describe how in a protocol run specified in this manner, and simulated using Sprite, the attacker can have capabilities of similar power as those of a Dolev- Yao style attacker.

7.1 Tool

The Sprite user interface provides a graphical menu to invoke either an editor, the Sn2Spi translator or the Spi2Java code generator. The editor component provides text editing abilities to create and edit standard notation and Spi specifications in plain text. Dialogue windows are provided when invoking either Sn2Spi or Spi2Java, to select the input specification file and the output file, as shown in figure 26.

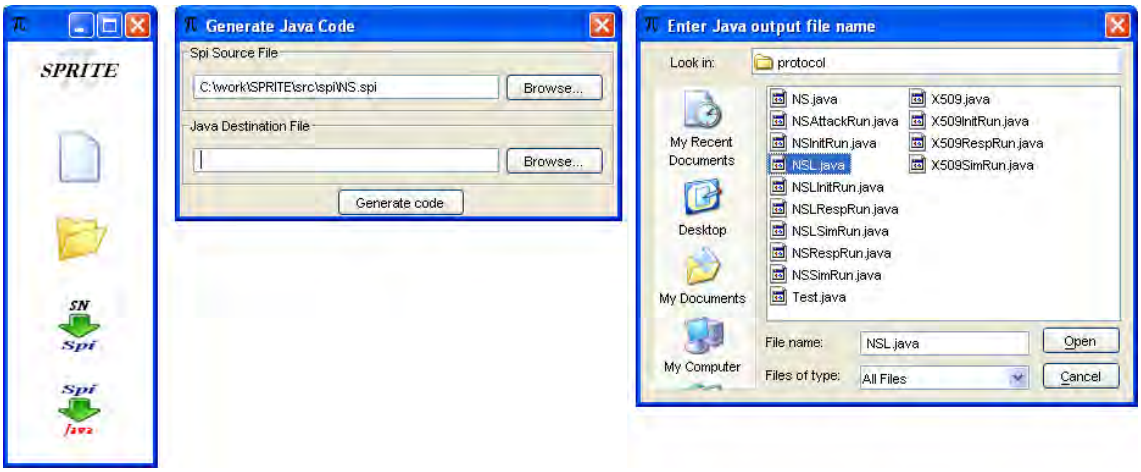

Figure 26: Using the Sprite UI to invoke Spi2Java.

7.1.1 Usability of Sprite

Sprite does not require the user to possess strong programming skills, nor experience implementing security protocols. The user may employ Spi2Java generated code in their own program, and as such should understand the security protocol being implemented, but they do not need to understand the generated implementation code.

7.1.2 Implementation

The user interface is implemented in Java using the AWT and Swing APIs. It can be invoked from command-line using the command `java SpriteUI`, provided the necessary Java classes are specified by the `CLASSPATH` environment variable.

7.2 Environment

The Sprite environment employs an SPP provider developed to allow the user to run a controlled simulation of a protocol. The provider displays a graphical user console for each Spi process that specifies a protocol role, and each channel. The consoles allow the execution of the Spi processes and the messages communicated over a channel to be controlled by user interaction.

7.2.1 The Simulation Provider

The simulation SPP provider was briefly introduced in section 5.5.3; we describe it in more detail in this section. Unlike the Standard and Typed Component SPP Providers, the Simulation SPP Provider is not intended for real world deployment, but rather for simulating protocol runs in a controlled environment.

Single Process, Multi-Threaded Execution

A simulated protocol run, using this provider, is executed as a single process on a single host. The implementation of each protocol role - generated from a Spi process specification - is run in its own thread within that process.

Spi2Java's explicit implementation of the parallel composition process $(P|Q)$ - as described in a previous chapter - makes this possible. Spi2Java will emit Java code that spawns a new thread to execute the implementation of P, while Q's implementation will be executed by the existing thread. Thus, if P and Q specify protocol roles they are executed by separate threads in the same process.

In contrast with this approach, when a "real world" SPP provider is used each protocol role's implementation is run in its own process and, probably, on a separate machines connected by a network. In such cases Spi2Java's explicit support for parallel composition is unnecessary: Parallel composition is implemented implicitly, by virtue of each protocol role's implementation running in its own process.

Message Components

Because the message component values need to be easily recognizable by the user during a simulation run, their sizes are reduced: Principal identifiers are a single byte - a single ASCII encoded character - and nonces are also a single byte - a numeric value in the range 0 to 255.

Communication by Queue

Network communication is simulated by a simple queue data structure. Each pair of communicating principals share an underlying queue. The queue is indirectly exposed to the principals via a concrete instance of the `sprite.spp.net.Channel` interface. Protocol role implementations can thus send and receive messages via the queue, using the channel abstraction.

The queue implementation has thread safe methods to enqueue and dequeue messages, which are accessed by the `sprite.spp.net.Channel` instance shared by the communicating principals. The queue mechanism allows messages to be controlled, at runtime, by user interaction.

Cryptographic Implementation

The Simulation SPP Provider employs the same cryptographic algorithms used by the Standard SPP Provider. Performance, in terms of user responsiveness, is increased by using short keys for public key encryption.

Spi Process Control via User Interface

The Simulation Provider provides a *console* for each Spi process that specifies a given protocol. A *Spi Process Console* allows the user to step through a Spi process at the granularity of individual process actions by clicking on the *step* button. Alternatively, the user can run the process through by clicking the *run* once.

Displaying Spi Process Information in the User Interface

The user interface console for each Spi process also contains two panes to display information about the process. The first pane displays the progression of the Spi process. As each Spi action is executed by the Spi2Java generated implementation, it is listed in this pane, along with the time of its execution. The second pane displays state information pertaining to the Spi variables. Whenever a binding action occurs, the name of the bound variable, and the value it was substituted with, is displayed.

Controlling Messages via Channel User Interface

A *Channel Console* is also provided for each channel in the Spi process specification of the protocol. The current message on the queue is displayed to the user as hexadecimal digits representing the binary value of the message. The value can be modified by the user and then sent to the intended recipient by pressing the *Send Message* button.

Like the Spi Process Console, the Channel Console has a *run* button that can be pressed once to allow all messages to be delivered unmodified to their intended recipient.

7.3 Run of the NS Protocol and Attack

In this section we demonstrate the Sprite environment by running a simulation of of a run of the Needham Schroeder Public Key Authentication Protocol. In the run, shown in figure 27, we also simulate an attacker using the attack discovered by Lowe in [39]. The inclusion of the attacker role in the simulation - and the fact that it is successful in the simulated run - demonstrates that Spi2Java does not, as expected, solve the problems inherent in a flawed security protocol specification. It also show's that Spi2Java can be used to implement Spi specifications of legitimate and attacker roles, for the purposes of gaining insight into protocol behaviour, via simulation, and to demonstrate an actual real world implementation of an attack.

The simulation is prepared by first using Spi2Java to generate code implementing the three Spi processes that specify the initiator, responder roles and attacker roles. These three process are composed in parallel, to form a single Spi process that specifies the protocol run. An implementation of this parallel Spi process is also generated using Spi2Java.

The Spi2Java generated code is then used with the Simulation SPP Provider. The simulation can be initiated by the user who can step through it action by action, allowing them to examine the state of the simulation (as displayed in the user interface), at any given time.

Obviously the generated code can be used with any other SPP providers - such as the Typed Component SPP Provider - to implement a real world attack. Such an implementation works against implementations of the legitimate roles running on separate machines communicating over a network.

1 $A \rightarrow B : \{n, A\}pub(B)$
2 $B \rightarrow A : \{n, m\}pub(A)$
3 $A \rightarrow B : \{m\}pub(B)$

Figure 27: The Needham-Schroeder protocol.

7.3.1 Protocol and Attack Specification

Legitimate Roles

The legitimate roles of Needham-Schroeder protocol are specified as the Spi processes $Init$,and $Resp$ in figures 28 and 29 respectively.

```
Init(c:Channel, A:Identifier, B:Identifier) =
   (n:Nonce)
   c!<{(n, A)}pub(B)>.                                  (Msg 1)
   c?(l:Encryption).                                    (Msg 2)
   case l of {j:Pair}priv(A) in
   let (x:Nonce, m:Nonce) = j in
   [x is n]
   c!<{m}pub(B)>.                                       (Msg 3)
   nil
```

Figure 28: Spi specification of the initiator role of the Needham-Schroeder protocol.

Both the legitimate roles have process definitions parameterised by a channel c. This channel represents an endpoint for communication on a public network and thus may not be viewed as secure by the principals. There is no guarantee that messages sent on the channel will get to the intended recipient.

```
Resp(c:Channel, B:Identifier) =
   c?(l:Encryption).                                    (Msg 1)
   case l of {j:Pair}priv(B) in
   let (n:Nonce, A:Identifier) = j in
   (m:Nonce)
   c!<{(n, m)}pub(A)>.                                  (Msg 2)
   c?(p:Encryption).                                    (Msg 3)
   case p of {x:Nonce}priv(B) in
   [x is m]
   nil
```

Figure 29: Spi specification of the responder role of the Needham-Schroeder protocol.

Lowe's Attack

Lowe's attack on the protocol is specified by the Spi process in figure 30. The attacker role's Spi process definition has two channel parameters, cA and cB. The former represents the channel for communication with the initiator A, the latter with the responder B.

The attacker's control of the network is modelled by defining a run of the attack as the Spi process *AttackRun* in figure 31.

B can - unbeknownst to it - only communicate with A via C.

```
Attack(cA:Channel, cB:Channel, B:Identifier, C:Identifier) =
   cA?(l:Encryption).                                    (Init's Msg 1)
   case l of {j:Pair}priv(C) in
   let (n:Nonce, A:Identifier) = j in
   cB!<{(n, A)}pub(B)>.                                  (Resp's Msg 1)
   cB?(p:Encryption).                                    (Resp's Msg 2)
   cA!<p>.                                               (Init's Msg 2)
   cA?(k:Encryption).                                    (Init's Msg 3)
   case k of {m:Pair}priv(C) in
   cB!<{m}pub(B)>.                                       (Resp's Msg 3)
   nil
```

Figure 30: Spi specification of Lowe's attack role on the Needham-Schroeder protocol.

```
AttackRun(A:Identifier, B:Identifier, C:Identifier) =
   (cAC)(cBC)(Init(cAC, A, B) | Resp(cBC, B) | Attack(cAC, cBC, A, B, C))
```

Figure 31: Spi specification for a run of the attack on the Needham-Schroeder protocol.

Dolev-Yao Attacker Capabilities

An attacker, specified in the manner of figure 31 - where well-intentioned principals A and B share channels cAC and cCB with a potential attacker C, but not with each other - has the power to:

1. Pass on messages unmodified, for example: Messages received from A on cAC can be passed on to B by sending them, as is, on cCB,
2. Remove messages: a message received on from A on cAC can just not be sent on cCB.
3. Introduce new messages by constructing it and sending it on either cAC or cCB.
4. Modify a message: a message received on cAC can be changed and then send on cCB.
5. Be a legitimate participant in a protocol run.

Given these powers, the *Attack* process has the capabilities of a Dolev-Yao style attacker [24].The first, third and last abilities are demonstrated in this example. C participates as itself in the responder role, with A as the initiator, in legitimate run of the Needham-Schroeder protocol. However, C also masquerades as A, in the initiator role in a second subverted run. In the subverted run, C reuses a message from the first run with A, and sends it to B and visa-versa to achieve its goal.

7.3.2 Successful Attack Run

The results of a successful run of this attack are shown in figure 32. This screen-shot shows output from a completed protocol run by Spi2Java generated implementations of the initiator, attacker and responder roles. The three console windows display the state of the run from the perspectives of the initiator (ALICE), attacker (EVE) and responder (BOB) respectively.

The bottom pane of the each output window lists the variable bindings for the role that have occurred during the run. A comparison of the values of the variables m and n in the initiator, attacker and responder's output, reveals that the attacker has the same values for these nonces as the two legitimate principals. Furthermore, as the runs for initiator and responder have completed successfully, the responder BOB is, according to the protocol, entitled to believe that he has just completed a successful run of the protocol with ALICE, and hence that m and n are secrets shared between the two. Clearly this is not the case: EVE also possesses the values.

7.3.3 Lowe's Fix

Lowe suggested an amended version of the protocol that corrects this problem [40]. By including the identifier of the responder in the second message, an attacker can no longer send the message unmodified to the initiator. Modifying the message to replace the real responder's ID is not possible, as the attacker does not possess the initiator's private key, and can thus not decrypt the message. The updated protocol is specified, informally, in figure 5 in a previous chapter.

The Spi specification for the initiator role is updated so that when the second message is received, the extra component - the responder's identifier - is unpacked and compared against the principal identifier sent in the first message. The responder role is simply updated by appending the responder's identifier to the second message. The Spi specification for the initiator and responder roles of this protocol, and a screen-shot of a run, appear in figures 7 and 25 respectively.

The Spi process specifying the ammended initiator role will halt when it compares the identifier B in message two, with the expected value (EVE), and see's that they do not match. Thus the attempted attack is thwarted.

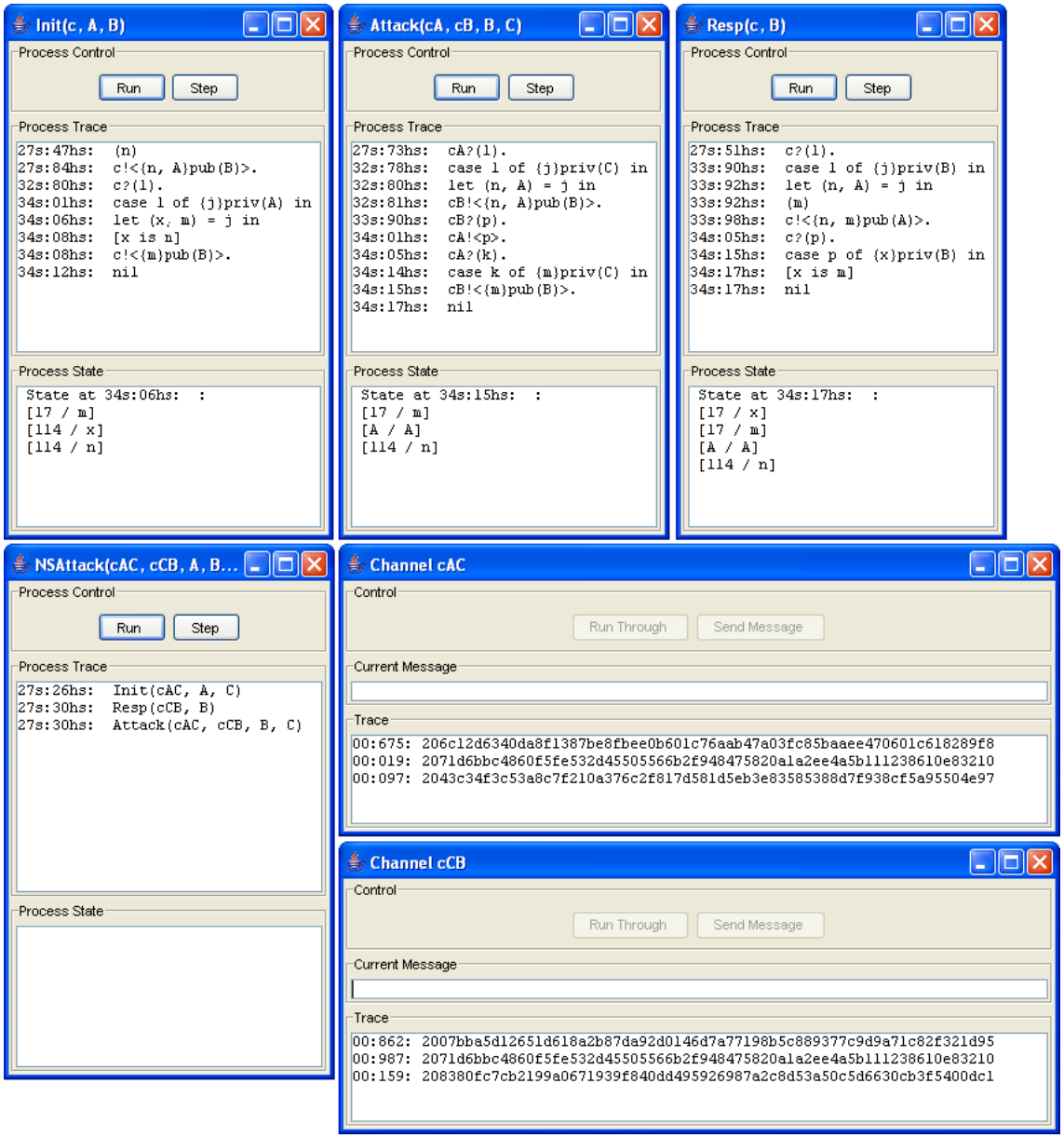

Figure 32: Run of Lowe's attack on the Needham-Schroeder protocol using the Sprite simulation environment.

7.4 Discussion

Sprite allows protocol runs to be simulated and controlled and their state observed, assisting in the understanding of a given security protocol. By stepping through simulation, and possibly modifying messages during the simulation, the Sprite user can gain insight into the protocol and possibly its weaknesses.

In describing the Spi specification for our chosen example simulation, we have shown that an attacker specified in this manner, and simulated using Sprite, has the idealised Dolev-Yao attacker capabilities. This is useful because as this model, or one's of similar power, are used by many security protocol analysis methodologies, theoretical attacks discovered during analysis can be demonstrated in a Sprite simulation.

Chapter 8

Results

In this chapter we describe a *case study* comparing manual security implementation against automated implementation using Sprite, *evaluate* Spi2Java against the requirements for high-integrity code generation, discuss the *classes of attacks* address by Spi2Java and *compare* Spi2Java to existing and current work in security protocol code generation.

8.1 Case Study: Manual vs. Automated Implementation

This section describes a case study in which we compared manually coded implementations of a real world network security protocol, against an implementation automatically generated by Sprite. The purpose of the case study was to evaluate how using automatic code generation mitigates the introduction of errors during implementation.

The three message protocol in the CCITT recommendations for the X.509 standard was chosen. Twenty-two pairs of fourth year computer science students were set a practical exercise to implement it. Their resulting implementations were evaluated, along with the automatically generated Sprite implementation, against a set of criteria and the results compared.

8.1.1 The Three Message CCITT X.509 Protocol

This protocol was selected because although its security properties are not very different from some simpler, theoretical protocols such as NSL, the increased complexity of its message format and

the number of checks and verifications that need to be performed, more accurately reflect security protocols deployed in the real world. The protocol is defined informally by the message flow in figure 33.

1 $A \rightarrow B : A, \{Ta, Na, B, Xa, \{Ya\}pub(B)\}priv(A)$
2 $B \rightarrow A : B, \{Tb, Nb, A, Na, Xb, \{Yb\}pub(A)\}priv(B)$
3 $A \rightarrow B : A\{Nb\}priv(A)$

Figure 33: The Three Message CCITT X.509 Protocol

A practical version, where a only hash of the message components is signed in preference to the components themselves, has also been specified as shown in figure 34. The second version was used in this study, as - depending on the key length - implementations of public key algorithms such as RSA can only encrypt blocks of data of limited length. By using a hash, only 20 bytes of data (in the case of the SHA hash algorithm) need to be encrypted to create the signature.

1 $A \rightarrow B : A, Ta, Na, B, Xa, \{Ya\}pub(B), \{h(Ta, Na, B, Xa, \{Ya\}pub(B))\}priv(A)$
2 $B \rightarrow A : B, Tb, Nb, A, Na, Xb, \{Yb\}pub(A), \{h(Tb, Nb, A, Na, Xb, \{Yb\}pub(A))\}priv(B)$
3 $A \rightarrow B : A\{Nb\}priv(A)$

Figure 34: A more practical specification of the Three Message CCITT X.509 Protocol

This three message protocol has the security properties necessary to meet the following criteria as they appear in [58]:

> "The protocol must ensure the confidentiality of Ya and Yb: if A and B follow the protocol, then an attacker should not be able to obtain Ya or Yb." [58]
>
> "The protocol must ensure the recipient B of the message 1 that the data Xa and Ya originate from A" [58]
>
> "The protocol must ensure the recipient A of the message 2 that the data Xb and Yb originate from B" [58]

Currently there are no known attacks against a correct implementation. However, there is a potential attack, described in [45], that is effective if the responder (i.e. principal B) does not verify the timestamp Ta in message 1.

8.1.2 Manual Implementations

The manual implementations were hand-ins from a practical exercise that was set for fourth year computer science students enrolled in a network and internetwork security course in the Department of Computer Science at the University of Cape Town. Given that this was the first course on network security they had attended, the students fulfilled the role of being programmers with little or no experience in implementing network security protocols - though they otherwise may have a fair amount of programming experience. The exercise required that the students implement both the initiator and responder roles of this three message protocol, from the standard notation specification. The exact format of the message fields was also specified: fixed length fields were defined for atomic message components, such as nonces, principal identifiers and timestamps; variable length fields, preceded by length indicators, were defined for compound components such as cipher texts and complete messages.

Students were encouraged to implement the protocol using the Java language: to benefit from its simplicity, memory management and well defined libraries for network communication and cryptography. However, they did not have the benefit of an API such as SPP, that provides functionality to instantiate, serialise and deserialise messages and message components, hides the details of cryptographic operations and abstracts network communication.

The main object of the practical was not to test the students general programming skills - although this was a side effect, but rather to test their interpretation of the protocol specification and understanding of the checks and verification steps required to implement the protocol correctly - maintaining its intended security properties.

8.1.3 Evaluation Criteria

We drew up a table of the actions the protocol implementations must perform in order to correctly implement the protocol behaviour. These actions included:

- Comparing received message components against corresponding values already in the principal's possession - i.e. checking nonces and identifiers,
- validating timestamps,
- generating fresh values for nonces and timestamps,
- correctly encrypting and decrypting the relevant message components and
- verifying signatures and message digests.

8.1.4 Evaluation of Manual Implementations

Only 9 of the 22 implementations of the initiator role did not miss any of the actions necessary to achieve the desired security properties of the protocol. Of the 13 flawed implementations: 3 did not verify that the nonce Na received in message 2 matched the nonce Na sent in message 1; 3 did not verify both the identifiers A and B in message 2; 5 did not verify the timestamp received in message 2 at all and 3 implementations didn't verify the timestamp correctly. The implementations of the responder role had comparable error rates and characteristics.

8.1.5 Automated Implementation Using Sprite

To automatically generate an implementation using Sprite, the protocol was first specified in a standard notation format suitable for input to Sn2Spi (see figure 35). Sn2Spi was run with this input to generate a Spi specification for the initiator, A and responder, B, roles of the protocol as shown in figure 36. This Spi specification was used as input to Spi2Java to produce the protocol implementation listed in appendix A.

```
Principals A, B

Channel cAB
Identifier A, B
Nonce Na, Nb, Ya, Yb
Timestamp Ta, Tb
UserData Ua, Ub

Possession A:cAB
Possession A:A
Possession A:B
Possession A:Ua
Possession B:cAB
Possession B:B
Possession B:Ub

A -> B: A, Ta, Na, B, Ua, {Ya}pub(B), {hash(Ta, Na, B, Ua, {Ya}pub(B))}priv(A)
B -> A: B, Tb, Nb, A, Na, Ub, {Yb}pub(A), {hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))}priv(B)
A -> B: A, {Nb}priv(A)
```

Figure 35: The Three Message CCITT X509 Protocol specified as input for Sn2Spi.

The Spi2Java generated X509 implementation was used with the Standard SPP Provider, that uses the same message format as the manual implementations. A run of the initiator and responder roles

```
A(A:Identifier, Ua:UserData, cAB:Channel, B:Identifier) =
        (Ta:Timestamp)
        (Na:Nonce)
        (Ya:Nonce)
        cAB!<A, Ta, Na, B, Ua, {Ya}pub(B), {hash(Ta, Na, B, Ua, {Ya}pub(B))}priv(A)>.
        cAB?(tmp0:Pair).
        let (tmp1:Identifier, Tb:Timestamp, Nb:Nonce, tmp2:Identifier, tmp3:Nonce,
                Ub:UserData, tmp4:Encryption, tmp6:Encryption) = tmp0 in
        case tmp4 of {Yb:Nonce}priv(A) in
        case tmp6 of {tmp5:Hash}pub(B) in
        [tmp1 is B]
        case Tb is valid in
        [tmp2 is A]
        [tmp3 is Na]
        [tmp5 is hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))]
        cAB!<A, {Nb}priv(A)>.
        nil

B(Ub:UserData, cAB:Channel, B:Identifier) =
        cAB?(tmp0:Pair).
        let (A:Identifier, Ta:Timestamp, Na:Nonce, tmp1:Identifier,
                Ua:UserData, tmp2:Encryption, tmp4:Encryption) = tmp0 in
        case tmp2 of {Ya:Nonce}priv(B) in
        case tmp4 of {tmp3:Hash}pub(A) in
        case Ta is valid in
        [tmp1 is B]
        [tmp3 is hash(Ta, Na, B, Ua, {Ya}pub(B))]
        (Tb:Timestamp)
        (Nb:Nonce)
        (Yb:Nonce)
        cAB!<B, Tb, Nb, A, Na, Ub, {Yb}pub(A),
                {hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))}priv(B)>.
        cAB?(tmp5:Pair).
        let (tmp6:Identifier, tmp8:Encryption) = tmp5 in
        case tmp8 of {tmp7:Nonce}pub(A) in
        [tmp6 is A]
        [tmp7 is Nb]
        nil
```

Figure 36: Sn2Spi translation of the Three Message CCITT X.509 Protocol.

of the implementation is shown in figure 37.

8.1.6 Evaluation of Sprite Generated Implementation

The Sprite generated implementation was evaluated in the same manner as the manual implementations. It was found to be correct with regard to the evaluation criteria, i.e. Sn2Spi translated the standard notation specification to Spi processes that performed all the necessary actions, and Spi2Java emitted code to implement all those actions.

8.1.7 Discussion

The Three Message CCITT X.509 Authentication Protocol, is an example of a real world protocol that may be implemented and deployed in a production environment. In this case, the Sn2Spi and Spi2Java components of Sprite, demonstrate that, used together, they can generate a correct implementation of the protocol from an informal standard notation specification. This implies that, at least for this protocol, Sn2Spi applies the correct rules for formalising the standard notation specification in terms of Spi. Furthermore, Spi2Java correctly implements the protocol actions. The only caveat, which applies more broadly to the use of Sprite, is the correctness of the cryptographic operations as implemented by the SPP provider used. In this study the Standard SPP Provider, which uses the JCE to implement RSA and AES encryption and SHA for message digest, was used. Given the wide use of the JCE and its open nature, we can be reasonably confident that these operations are implemented correctly.

The results of the comparison of manual and Sprite implementations, suggests that an inexperienced protocol developer, can take a security protocol specification and use Sprite to generate an implementation in which we can have more confidence than one coded by themselves. There are obvious limitations to drawing conclusions from this case study: experienced professional programmers are arguably more thorough and have the time and resources to develop suitable test cases which may catch many of the errors seen in the manual implementations. Nevertheless, as discussed in the introduction to this work the discovery of flaws in widely deployed security protocol implementations, both commercial and open source, is a regular occurrence.

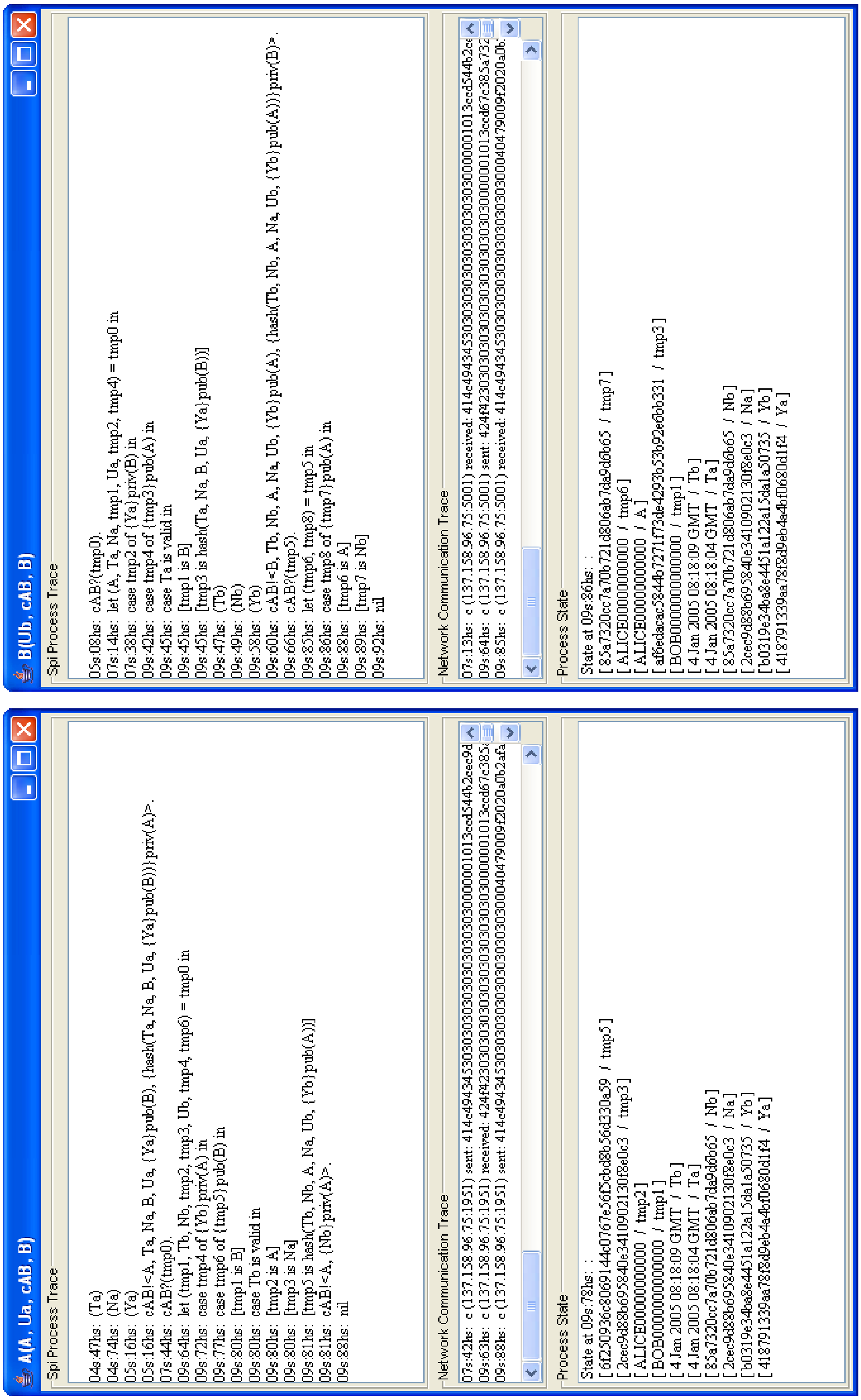

Figure 37: Run of Spi2Java generated implementations of the Three Message CCITT X.509 Protocol initiator and responder roles.

8.2 Evaluation of Spi2Java

In this section we evaluate Spi2Java against the Whalen et al's [64] requirements for integrity code generation introduced in the background chapter.

8.2.1 Requirement 1: Formally Defined Source and Target Languages

Both the source language, Spi, and the target language, Java, have formally defined semantics.

8.2.2 Requirement 2: Semantic Preserving Translation

We have defined a translation from Spi constructs to Java code segments and have provided detailed arguments for the translations, though they are not strictly formal or rigorous as would be ideal. Thus Spi2Java does not entirely meet this requirement. However, given that each Spi Calculus language construct is translated to a simple and concise Java code segment, the user can inspect them to satisfy themselves of their correctness, or review the mapping definitions and our supporting arguments, provided in section 6.4.4.

8.2.3 Requirement 3: Validated Translator

The third requirement states that either the translation implementation, in this case Spi2Java, is validated or the generated code is verified. We focus on the former: It is more efficient to verify the translation definition and the implementation of the translation once, than to validate the code generated by every run of the translator. We have used Prolog to implement the translation rules. The Prolog rules that implement the translation are very close to the logical rules that specify it, providing a high level of confidence in the implementation. As discussed previously, this is not an absolute guarantee of correctness - given the distance of Prolog code from the physical machine - but is as close as can be reasonably expected.

8.2.4 Requirement 4: Rigorously Tested Translator

This requirement has arguably not been met to a degree sufficient for production software. Spi2Java has been used to implement a number of security protocols (some of theoretical interest, such as the Needham-Schroeder protocol, and some of that reflect real world protocols, such as the Three

Message X.509 protocol). Although these implementations have been manually inspected them for correctness, this does not qualify as rigorous testing, nor does it provide complete coverage of the code that implements Spi2Java

8.2.5 Requirement 5: Structured, Documented and Traceable Generated Code

Determining whether or not code is well structured is a somewhat subjective exercise. Spi2Java emits code that is structured for clarity and ease of understanding. The Spi process for each protocol role is implemented in a single instance method. These methods are preceded by a comment documenting the Spi process that they implement.

The code within the methods is clearly partitioned into segments implementing each process action. Each such code segment is preceded by a comment, indicating corresponding Spi action in the specification that it implements. This allows the generated code to be traced back to the Spi specification at a very fine level of granularity. Conversely, the code segment that implements a Spi action in the specification can easily located in the generated code. This traceability facilitates the manual verification of Spi2Java implementations by inspection.

Further to meeting the traceability requirement, the comments emitted in the Spi2Java generated code are mirrored in trace calls in the generated code. This allows the progress of the Spi processes the specify the implementation, to be tracked by the user as the implementation runs. It also allows the state of the Spi variables to displayed to the user. Finally it provides mechanism for the user to control the progress of the implementation run, a feature used by the Simulation SPP Provider to allow the user to step through a run of a protocol as described in section 7.2.1.

8.3 Classes of Attack Addressed

This section discusses classes of attacks and the properties of Spi2Java and SPP that serve to minimise the vulnerability of Spi2Java generated implementations to these attack classes.

8.3.1 Protocol Logic Attacks

Attacks that exploit flaws in protocol specifications, where the logic of the protocol does not achieve the intended security goals under certain conditions, are largely mitigated by the fact that protocols

must be specified in Spi for input into Spi2Java. As discussed previously, the Spi formalism can be used with a number of techniques and tools to verify the correctness of the protocol specification. While this does not prevent Spi2Java from being used to generate implementations from flawed protocol specifications, as demonstrated in section 7.3, it does provide the diligent security protocol developer the opportunity verify the protocol specification prior to implementation.

8.3.2 Protocol Logic Implementation Attacks

Given that the protocol specification is correct, there is still the possibility of the implementation failing to follow the specification faithfully. With Spi2Java, individual Spi actions are mapped to Java code segments that preserve their semantics. Spi2Java is implemented by simple Prolog rules that specify exactly that mapping. During the code generation process, Spi2Java emits trace comments that allow each emitted Java code segments to be traced back to the corresponding Spi action that it implements in the originating specification. We can thus have a high level of confidence in the correctness of the mapping from Spi to Java, the implementation of that mapping, and hence the protocol implementations generated using Spi2Java.

8.3.3 Type Flaw Attacks

The Typed Component SPP provider implements a tagging scheme similar to that proposed in [30] - which has been shown to negate the risk of type flaw attacks that may be perpetrated by fooling the protocol implementation into interpreting a message component of one type, as a component of another type.

8.3.4 Buffer Overflow Attacks

Buffer overflow attacks are a specific instance of the general class of attacks that allow an attacker to execute arbitrary code on a machine by getting the instruction pointer to point to an area of memory that has been suitably modified. Buffer overflow exploits are, however, by far the most common form of this class of attack. They are generally implemented by exploiting code written in C, or C++, that uses unsafe string manipulation routines operating on null terminated byte array in memory. Because these functions do not allow the length of the arrays being operated to be specified up front, they allow memory that does not below to the array to be modified.

Newer, managed, languages, such as Java and C#, avoid this problem by using arrays with built in bounds checking. These data structures contain information about their length, as well as code to

ensure that memory that does not belong to the array is not accessed when the array is referenced. By using Java as the target language, Spi2Java generated protocol implementations are not generally vulnerable to this class of attack. There is always the possibility that the implementation of the Java Virtual Machine the protocol implementation is running on, may have vulnerabilities of this type. However, exploiting them would be a very complex task, and successful exploits are only likely to work on a specific JVM implementation.

Another potential, though very impractical attack on managed languages such as Java, relies on exploiting randomly flipped bits - a result of cosmic rays or hardware defects, in physical system memory [29].This attack involves the creation of elaborate, inter-referencing data structures, designed in such a way as to maximise the chance that if any bit is flipped, a variable will end up pointing to an object of an incompatible type - thus subverting the Java type system.

8.3.5 Summary of Sprite's Attack Class Coverage

Sprite and its component parts - Sn2Spi, SPP and Spi2Java - advances security protocol implementation through a structured approach that directly addresses three (i.e. *Protocol Logic Implementation*, *Type Flaw* and *Buffer Overflow* attacks) of the four attack classes and, in conjunction with security protocol analysis, addresses the class of attacks based on flawed security protocol specifications.

8.4 Comparison with Existing and Current Work

A number of difficulties are apparent when attempting to compare, Sprite (specifically the Spi2Java component) to other code generation tools for security protocols. With the exception of COSP-J none the other tools are actually available for download and use. Few have any associated publications detailing the definition of the translation from specification language to implementation language and the implementation of that translation definition. None of them have documentation relating the semantics of the specification language constructs to those of the generated code.

8.4.1 COSP-J

COSP-J is the best documented with regard to its implementation. It is partly based on code from Casper[40] - Lowe's translation tool that has been very successfully used to translate informal specifications to CSP for automated analysis [23]. Casper has produced successful results [39], which should inspire confidence in the implementation of the translation.

8.4.2 Another Spi2Java

After the completion of the implementation of our Spi2Java tool and submission for publication of a paper summarising this work [6], we came across another tool [18], with the same name, and purpose. A short publication describing this tool is available, but doesn't describe the translation implementation nor define the mapping from specification language to implementation code.

8.4.3 CAPSL

The approach to code generation from CAPSL involves generating Java classes that represent CIL (the CAPSL intermediate language) constructs in a parse tree data structure that represents the protocol [50]. The Java objects in the data structure are able to generate code, implementing the CIL construct that they represent. While there is no emphasis on a formal methods for code generation, this approach clearly links the source specification in CIL to the generated Java code that implements the protocol. However, the CIL specification has to be generated via translation from CAPSL, as CIL is not easily usable by the security protocol engineer. Thus there are two translation or code generation processes that have to be undertaken to get from the source specification in CAPSL to the implementation in Java.

The CAPSL code generation approach's two phases allow Java code that represents protocol logic to be independent of the code that implements some of the actions and practicalities, like message component type interpretation. This aspect is mirrored by Spi2Java generated code's use of the SPP API that allows protocol logic implementation to be separated from lower level implementation concerns.

The CAPSL approach has some limitations:

1. The generated code is not suitable for application use; it is intended to run in a demonstration environment that is part of the work described in [50].
2. Public key encryption is not implemented as it was not available in the JCE provider employed.
3. The use of specific encryption algorithms is entrenched in the key objects, making it difficult to change these algorithms.

The first and second limitations are reasonably easily addressed by firstly modifying the generated code to make it suitable for application use, and secondly using an updated JCE provider that implements RSA public key encryption. The final limitation is obviously not as easily rectified.

This limitation is not encountered by Sprite, as Spi2Java generated code uses the SPP API that abstracts the details of the encryption algorithms as well message component type interpretation and network operations.

Some of the suggestions that the authors of [50] make for future extensions to their work are:

1. providing a choice of cryptographic algorithms,
2. generating code for application use and
3. adding functionality to the demonstration environment to allow the protocol messages to be modified.

These suggested extensions have been largely implemented by Sprite. The first via the use of the SPP API. The second is implemented by default by Spi2Java which generates code for application use, but also allows a simulation SPP provider to be used to provide a simulation environment that parallels CAPSL's demonstration environment. The third extension is implemented to some degree by Sprite's simulation environment that allows the user to inspect and modify protocol messages, though not in as user friendly a fashion as suggested by the authors of[50], who suggest that individual message components could be saved by the user and used to synthesise new messages. Sprite only provides access to a binary representation of the message, which the user can manually deconstruct into message components, modify and use to construct new messages.

8.4.4 χ-Spaces

χ-Spaces is a programming language that is a concrete version of SPL [6]. As such, the specification is the implementation. However, the publications describing χ-Spaces do not cover the compilation process from χ-Spaces to the implementation language - Java. It is thus currently not possible to evaluate this work.

Chapter 9

Conclusion

9.1 Meeting the Objectives

In the introduction we defined the objective of our work in terms of the following requirements:

1. It must have the ability to automatically translate informal to formal specifications,
2. be able to automatically generate security protocol implementations from formal specifications and provide a high level of confidence in those implementations and
3. must realise a well defined methodology and tools for security protocol implementation, which are easily usable by the security protocol engineer.

The first requirement is met by Sn2Spi, which converts informal specifications, in our well defined version of the standard notation, to formal Spi Calculus specifications. The second, and key requirement, is achieved by Spi2Java's ability to automatically generate security protocol implementations from Spi Calculus specifications. Thus Sprite can generate implementations from both informal and formal security protocol specifications.

We evaluated Spi2Java against the requirements for high integrity code generation. Spi2Java fully meets three of these requirements, partially meets one of the requirements but does not meet the rigorous testing requirement due to resource constraints. Although a complete formal proof of the correctness of the Spi2Java mapping from Spi Calculus to Java code is not provided, as we have followed the requirements for high integrity code generation where feasible, the Spi2Java user should

be confident in the correctness of the generated code. Thus the second objective is largely, though not entirely met.

Sn2Spi and Spi2Java both provide simple command line interfaces. The SPP API provides a clean, simple interface to network and cryptographic operations. Sprite meets the objective of being easily usable by the security protocol engineer by providing a graphical user interface to the Sn2Spi and Spi2Java tools.

Though the implementation aspect of the development process is well defined in Sprite, through the use of Sn2Spi and Spi2Java, the security protocol analysis process is not. As mentioned from the outset, security protocol analysis is outside the scope of this work, but is nevertheless a complementary activity. The example presented in section 7.3, explicitly demonstrates that if the protocol specification is flawed, the errors will be propagated to the generated code.

An automated interface to a model checker, such as MMC, or multi-dimensional analysis environment such as SPEAR II should be considered as part of future work to fully meet this requirement.

9.2 Limitations and Future Work

The primary limitations, and areas for future extension, of this work are:

1. the lack of seamless integration with an analysis tool, such as SPEAR II or the MMC model checker, and
2. the correctness of the cryptographic operations, and the interface to them, implemented by the SPP API and conforming providers.

The first limitation can be addressed by future implementation work. Integration with the MMC model checker would be the most desirable approach. By making the necessary modifications to the syntax of the Spi Calculus generated by Sn2Spi, informal specification could be automatically translated and then partly automatically verified by forwarding the Sn2Spi output to MMC.

Tackling the second limitation, which was discussed initially when describing the scope of the work, requires further research as well as implementation. A possible approach would be to follow the work of Backes et al in [44]. They develop an idealised cryptographic library - that can be used like that Dolev-Yao cryptographic model - and a real library that conforms to the idealised one. However, there may still be a way to go before being able to show, in an acceptably formal manner, that a concrete Java implementation conforms to a formal cryptographic model such as Dolev-Yao.

Other limitations of this work are covered in Sections 8.2 and 9.1, where Spi2Java is evaluated, and we discuss how Sprite meets the objectives set out in the introduction, respectively.

9.3 Contributions

Sn2Spi provides an automated tool to convert informal security protocol specifications to Spi Calculus processes. The resulting Spi Calculus processes can then be subjected to all the formal methodologies that have been developed for analysing, verifying and reasoning about security protocols specified in Spi. To our knowledge no other tool that fulfills this role exists.

Spi2Java is one of a handful of code generators for security protocols. Although we have since discovered that it is no longer unique in function, or name, [18], when we started developing it there was no other published research covering such work. However, unlike other code generators for security protocols, with the exception of COSP-J, we have described in detail not only the mapping from the formal specification language to implementation code, but also the implementation in Prolog of that translation and published a summary of our approach in [6].

Some degree of isolation of the security protocol logic from the cryptographic algorithm implementations is evident in other code generators [23, 18]. Our SPP API clearly defines this separation and extends it to network communications, by defining a distinct interface for these operations. The API is also suitable for use by other implementation methods.

The process of implementation, from either informal or formal specifications is facilitated by Sprite's user interface for invoking the Sn2Spi and Spi2Java tools. The generated code is suitably packaged and its functionality clearly exposed and documented, for use by, and incorporation in, other Java applications.

Finally, three distinct areas of work that have been addressed plus a fourth unifying dimension, for which software artefacts have been produced, namely Sn2Spi, SPP API, Spi2Java and Sprite. While each is makes an individual contribution, we contend that the combination of these tools, when applied in the context of security protocol engineering (as described in section 2.1) realises the structured approach to network security protocol implementation as sought by this work.

Appendix A

Spi2Java Generated X.509 Implementation

```
package protocol;

import java.io.*;
import sprite.spi.*;
import sprite.spp.*;
import sprite.spp.net.*;
import sprite.spp.term.*;

/**
 * Generated by Sprite::Spi2Java on Thu Dec 30 16:56:14 2004.
 * See http://people.cs.uct.ac.za/~btobler/sprite/ for more information.
 */
public class X509Gen extends sprite.spi.Process
{
   public X509Gen(Provider provider)
   {
      super(provider);
   }

   // A(A, Ua, cAB, B) =
      // (Ta)
      // (Na)
      // (Ya)
      // cAB!<A, Ta, Na, B, Ua, {Ya}pub(B), {hash(Ta, Na, B, Ua, {Ya}pub(B))}priv(A)>.
      // cAB?(tmp0).
      // let (tmp1, Tb, Nb, tmp2, tmp3, Ub, tmp4, tmp6) = tmp0 in
      // case tmp4 of {Yb}priv(A) in
      // case tmp6 of {tmp5}pub(B) in
      // [tmp1 is B]
      // case Tb is valid in
      // [tmp2 is A]
      // [tmp3 is Na]
      // [tmp5 is hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))]
      // cAB!<A, {Nb}priv(A)>.
      // nil
   public void A(
      final Identifier A,
      final UserData Ua,
      final Channel cAB,
      final Identifier B)
   {
      final Trace _trace = newTrace("A(A, Ua, cAB, B)");
      Thread.currentThread().setName("A(A, Ua, cAB, B)");

      // (Ta)
      _trace.trace("(Ta)");
      final Timestamp Ta = newTimestamp();
      if (Ta == null) throw new AssignmentException("Ta", null);
      _trace.update("Ta", Ta);
```

```
// (Na)
_trace.trace("(Na)");
final Nonce Na = newNonce();
if (Na == null) throw new AssignmentException("Na", null);
_trace.update("Na", Na);

// (Ya)
_trace.trace("(Ya)");
final Nonce Ya = newNonce();
if (Ya == null) throw new AssignmentException("Ya", null);
_trace.update("Ya", Ya);

// cAB!<A, Ta, Na, B, Ua, {Ya}pub(B), {hash(Ta, Na, B, Ua, {Ya}pub(B))}priv(A)>.
_trace.trace("cAB!<A, Ta, Na, B, Ua, {Ya}pub(B), {hash(Ta, Na, B, Ua, {Ya}pub(B))}priv(A)>.");

send(cAB, pair(A, pair(Ta, pair(Na, pair(B, pair(Ua, pair(encrypt(Ya,pub(B)),
        encrypt(hash(pair(Ta,pair(Na,
                pair(B, pair(Ua, encrypt(Ya, pub(B))))))), priv(A))))))), _trace);

// cAB?(tmp0).
_trace.trace("cAB?(tmp0).");
final Pair tmp0 = getTermFactory().unpackPair(recv(cAB, _trace));
if (tmp0 == null) throw new AssignmentException("tmp0", null);
_trace.update("tmp0", tmp0);

// let (tmp1, Tb, Nb, tmp2, tmp3, Ub, tmp4, tmp6) = tmp0 in
_trace.trace("let (tmp1, Tb, Nb, tmp2, tmp3, Ub, tmp4, tmp6) = tmp0 in ");
final Identifier tmp1;
final Timestamp Tb;
final Nonce Nb;
final Identifier tmp2;
final Nonce tmp3;
final UserData Ub;
final Encryption tmp4;
final Encryption tmp6;
{
   InputStream _is = new ByteArrayInputStream(tmp0.getData());
   tmp1 = getTermFactory().unpackIdentifier(_is);
   if (tmp1 == null) throw new AssignmentException("tmp1", null);
   _trace.update("tmp1", tmp1);

   {
      Pair _p = getTermFactory().unpackPair(_is);
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      Tb = getTermFactory().unpackTimestamp(_is);
      if (Tb == null) throw new AssignmentException("Tb", null);
      _trace.update("Tb", Tb);
   }

   {
      Pair _p = getTermFactory().unpackPair(_is);
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      Nb = getTermFactory().unpackNonce(_is);
      if (Nb == null) throw new AssignmentException("Nb", null);
      _trace.update("Nb", Nb);
   }

   {
      Pair _p = getTermFactory().unpackPair(_is);
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      tmp2 = getTermFactory().unpackIdentifier(_is);
      if (tmp2 == null) throw new AssignmentException("tmp2", null);
      _trace.update("tmp2", tmp2);
   }

   {
      Pair _p = getTermFactory().unpackPair(_is);
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      tmp3 = getTermFactory().unpackNonce(_is);
      if (tmp3 == null) throw new AssignmentException("tmp3", null);
      _trace.update("tmp3", tmp3);
   }

   {
      Pair _p = getTermFactory().unpackPair(_is);
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      Ub = getTermFactory().unpackUserData(_is);
      if (Ub == null) throw new AssignmentException("Ub", null);
      _trace.update("Ub", Ub);
   }

   {
      Pair _p = getTermFactory().unpackPair(_is);
```

```
      _is = new ByteArrayInputStream(_p.getData());
      if (_p == null) throw new AssignmentException("_p", null);
      tmp4 = getTermFactory().unpackEncryption(_is);
      if (tmp4 == null) throw new AssignmentException("tmp4", null);
      _trace.update("tmp4", tmp4);
   }

   tmp6 = getTermFactory().unpackEncryption(_is);
   if (tmp6 == null) throw new AssignmentException("tmp6", null);
   _trace.update("tmp6", tmp6);

}


// case tmp4 of {Yb}priv(A) in
_trace.trace("case tmp4 of {Yb}priv(A) in ");
final Nonce Yb = getTermFactory().unpackNonce(decrypt(tmp4, priv(A)));
if (Yb == null) throw new AssignmentException("Yb", null);
_trace.update("Yb", Yb);

// case tmp6 of {tmp5}pub(B) in
_trace.trace("case tmp6 of {tmp5}pub(B) in ");
final Hash tmp5 = getTermFactory().unpackHash(decrypt(tmp6, pub(B)));
if (tmp5 == null) throw new AssignmentException("tmp5", null);
_trace.update("tmp5", tmp5);

// [tmp1 is B]
_trace.trace("[tmp1 is B]");
if (!tmp1.match(B))
{
   _trace.trace("MATCH FAILED: tmp1 != B.");
   _trace.trace("Variable tmp1's value is " + tmp1.toString());
   _trace.trace("Variable B's value is " + B.toString());
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Match failed: tmp1 is not equal to B.");
}

// case Tb is valid in
_trace.trace("case Tb is valid in ");
if (!valid(Tb))
{
   _trace.trace("TIMESTAMP EXPIRED: " + Tb);
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Timestamp expired: Tb");    }

// [tmp2 is A]
_trace.trace("[tmp2 is A]");
if (!tmp2.match(A))
{
   _trace.trace("MATCH FAILED: tmp2 != A.");
   _trace.trace("Variable tmp2's value is " + tmp2.toString());
   _trace.trace("Variable A's value is " + A.toString());
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Match failed: tmp2 is not equal to A.");
}

// [tmp3 is Na]
_trace.trace("[tmp3 is Na]");
if (!tmp3.match(Na))
{
   _trace.trace("MATCH FAILED: tmp3 != Na.");
   _trace.trace("Variable tmp3's value is " + tmp3.toString());
   _trace.trace("Variable Na's value is " + Na.toString());
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Match failed: tmp3 is not equal to Na.");
}

// [tmp5 is hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))]
_trace.trace("[tmp5 is hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))]");
if (!tmp5.match(hash(pair(Tb, pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))))))
{
   _trace.trace("MATCH FAILED: tmp5 != hash(pair(Tb, pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))))).");
   _trace.trace("Variable tmp5's value is " + tmp5.toString());
   _trace.trace(
           "Variable hash(pair(Tb, pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))))'s value is "
           + hash(pair(Tb, pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))))).toString());
   _trace.trace("PROCESS STUCK");
   throw new StuckException(
           "Match failed: tmp5 is not equal to hash(pair(Tb, pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))))).");
}

// cAB!<A, {Nb}priv(A)>.
_trace.trace("cAB!<A, {Nb}priv(A)>.");
send(cAB, pair(A, encrypt(Nb, priv(A))), _trace);

// nil
_trace.trace("nil");
return;
```

```
}

// B(Ub, cAB, B) =
   // cAB?(tmp0).
   // let (A, Ta, Na, tmp1, Ua, tmp2, tmp4) = tmp0 in
   // case tmp2 of {Ya}priv(B) in
   // case tmp4 of {tmp3}pub(A) in
   // case Ta is valid in
   // [tmp1 is B]
   // [tmp3 is hash(Ta, Na, B, Ua, {Ya}pub(B))]
   // (Tb)
   // (Nb)
   // (Yb)
   // cAB!<B, Tb, Nb, A, Na, Ub, {Yb}pub(A), {hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))}priv(B)>.
   // cAB?(tmp5).
   // let (tmp6, tmp8) = tmp5 in
   // case tmp8 of {tmp7}pub(A) in
   // [tmp6 is A]
   // [tmp7 is Nb]
   // nil
public void B(
   final UserData Ub,
   final Channel cAB,
   final Identifier B)
{
   final Trace _trace = newTrace("B(Ub, cAB, B)");
   Thread.currentThread().setName("B(Ub, cAB, B)");

   // cAB?(tmp0).
   _trace.trace("cAB?(tmp0).");
   final Pair tmp0 = getTermFactory().unpackPair(recv(cAB, _trace));
   if (tmp0 == null) throw new AssignmentException("tmp0", null);
   _trace.update("tmp0", tmp0);

   // let (A, Ta, Na, tmp1, Ua, tmp2, tmp4) = tmp0 in
   _trace.trace("let (A, Ta, Na, tmp1, Ua, tmp2, tmp4) = tmp0 in ");
   final Identifier A;
   final Timestamp Ta;
   final Nonce Na;
   final Identifier tmp1;
   final UserData Ua;
   final Encryption tmp2;
   final Encryption tmp4;
   {
      InputStream _is = new ByteArrayInputStream(tmp0.getData());
      A = getTermFactory().unpackIdentifier(_is);
      if (A == null) throw new AssignmentException("A", null);
      _trace.update("A", A);

      {
         Pair _p = getTermFactory().unpackPair(_is);
         _is = new ByteArrayInputStream(_p.getData());
         if (_p == null) throw new AssignmentException("_p", null);
         Ta = getTermFactory().unpackTimestamp(_is);
         if (Ta == null) throw new AssignmentException("Ta", null);
         _trace.update("Ta", Ta);
      }

      {
         Pair _p = getTermFactory().unpackPair(_is);
         _is = new ByteArrayInputStream(_p.getData());
         if (_p == null) throw new AssignmentException("_p", null);
         Na = getTermFactory().unpackNonce(_is);
         if (Na == null) throw new AssignmentException("Na", null);
         _trace.update("Na", Na);
      }

      {
         Pair _p = getTermFactory().unpackPair(_is);
         _is = new ByteArrayInputStream(_p.getData());
         if (_p == null) throw new AssignmentException("_p", null);
         tmp1 = getTermFactory().unpackIdentifier(_is);
         if (tmp1 == null) throw new AssignmentException("tmp1", null);
         _trace.update("tmp1", tmp1);
      }

      {
         Pair _p = getTermFactory().unpackPair(_is);
         _is = new ByteArrayInputStream(_p.getData());
         if (_p == null) throw new AssignmentException("_p", null);
         Ua = getTermFactory().unpackUserData(_is);
         if (Ua == null) throw new AssignmentException("Ua", null);
         _trace.update("Ua", Ua);
      }

      {
         Pair _p = getTermFactory().unpackPair(_is);
         _is = new ByteArrayInputStream(_p.getData());
         if (_p == null) throw new AssignmentException("_p", null);
```

```
      tmp2 = getTermFactory().unpackEncryption(_is);
      if (tmp2 == null) throw new AssignmentException("tmp2", null);
      _trace.update("tmp2", tmp2);
   }

   tmp4 = getTermFactory().unpackEncryption(_is);
   if (tmp4 == null) throw new AssignmentException("tmp4", null);
   _trace.update("tmp4", tmp4);

}


// case tmp2 of {Ya}priv(B) in
_trace.trace("case tmp2 of {Ya}priv(B) in ");
final Nonce Ya = getTermFactory().unpackNonce(decrypt(tmp2, priv(B)));
if (Ya == null) throw new AssignmentException("Ya", null);
_trace.update("Ya", Ya);

// case tmp4 of {tmp3}pub(A) in
_trace.trace("case tmp4 of {tmp3}pub(A) in ");
final Hash tmp3 = getTermFactory().unpackHash(decrypt(tmp4, pub(A)));
if (tmp3 == null) throw new AssignmentException("tmp3", null);
_trace.update("tmp3", tmp3);

// case Ta is valid in
_trace.trace("case Ta is valid in ");
if (!valid(Ta))
{
   _trace.trace("TIMESTAMP EXPIRED: " + Ta);
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Timestamp expired: Ta");    }

// [tmp1 is B]
_trace.trace("[tmp1 is B]");
if (!tmp1.match(B))
{
   _trace.trace("MATCH FAILED: tmp1 != B.");
   _trace.trace("Variable tmp1's value is " + tmp1.toString());
   _trace.trace("Variable B's value is " + B.toString());
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Match failed: tmp1 is not equal to B.");
}

// [tmp3 is hash(Ta, Na, B, Ua, {Ya}pub(B))]
_trace.trace("[tmp3 is hash(Ta, Na, B, Ua, {Ya}pub(B))]");
if (!tmp3.match(hash(pair(Ta, pair(Na, pair(B, pair(Ua, encrypt(Ya, pub(B)))))))))
{
   _trace.trace("MATCH FAILED: tmp3 != hash(pair(Ta, pair(Na, pair(B, pair(Ua, encrypt(Ya, pub(B)))))).");
   _trace.trace("Variable tmp3's value is " + tmp3.toString());
   _trace.trace("Variable hash(pair(Ta, pair(Na, pair(B, pair(Ua, encrypt(Ya, pub(B)))))))'s value is " + hash(pair(Ta, pair(Na, pair(B, pair(Ua, encrypt(Ya, pub(B)))))).toStrin
   _trace.trace("PROCESS STUCK");
   throw new StuckException("Match failed: tmp3 is not equal to hash(pair(Ta, pair(Na, pair(B, pair(Ua, encrypt(Ya, pub(B)))))).");
}

// (Tb)
_trace.trace("(Tb)");
final Timestamp Tb = newTimestamp();
if (Tb == null) throw new AssignmentException("Tb", null);
_trace.update("Tb", Tb);

// (Nb)
_trace.trace("(Nb)");
final Nonce Nb = newNonce();
if (Nb == null) throw new AssignmentException("Nb", null);
_trace.update("Nb", Nb);

// (Yb)
_trace.trace("(Yb)");
final Nonce Yb = newNonce();
if (Yb == null) throw new AssignmentException("Yb", null);
_trace.update("Yb", Yb);

// cAB!<B, Tb, Nb, A, Na, Ub, {Yb}pub(A), {hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))}priv(B)>.
_trace.trace("cAB!<B, Tb, Nb, A, Na, Ub, {Yb}pub(A), {hash(Tb, Nb, A, Na, Ub, {Yb}pub(A))}priv(B)>.");
send(cAB, pair(B, pair(Tb, pair(Nb, pair(A, pair(Na,
        pair(Ub, pair(encrypt(Yb, pub(A)), encrypt(hash(pair(Tb,
                pair(Nb, pair(A, pair(Na, pair(Ub, encrypt(Yb, pub(A))))))), priv(B)))))))), _trace);

// cAB?(tmp5).
_trace.trace("cAB?(tmp5).");
final Pair tmp5 = getTermFactory().unpackPair(recv(cAB, _trace));
if (tmp5 == null) throw new AssignmentException("tmp5", null);
_trace.update("tmp5", tmp5);

// let (tmp6, tmp8) = tmp5 in
_trace.trace("let (tmp6, tmp8) = tmp5 in ");
final Identifier tmp6;
final Encryption tmp8;
{
```

```
         InputStream _is = new ByteArrayInputStream(tmp5.getData());
         tmp6 = getTermFactory().unpackIdentifier(_is);
         if (tmp6 == null) throw new AssignmentException("tmp6", null);
         tmp8 = getTermFactory().unpackEncryption(_is);
         if (tmp8 == null) throw new AssignmentException("tmp8", null);
      }
      _trace.update("tmp6", tmp6);
      _trace.update("tmp8", tmp8);


      // case tmp8 of {tmp7}pub(A) in
      _trace.trace("case tmp8 of {tmp7}pub(A) in ");
      final Nonce tmp7 = getTermFactory().unpackNonce(decrypt(tmp8, pub(A)));
      if (tmp7 == null) throw new AssignmentException("tmp7", null);
      _trace.update("tmp7", tmp7);

      // [tmp6 is A]
      _trace.trace("[tmp6 is A]");
      if (!tmp6.match(A))
      {
         _trace.trace("MATCH FAILED: tmp6 != A.");
         _trace.trace("Variable tmp6's value is " + tmp6.toString());
         _trace.trace("Variable A's value is " + A.toString());
         _trace.trace("PROCESS STUCK");
         throw new StuckException("Match failed: tmp6 is not equal to A.");
      }

      // [tmp7 is Nb]
      _trace.trace("[tmp7 is Nb]");
      if (!tmp7.match(Nb))
      {
         _trace.trace("MATCH FAILED: tmp7 != Nb.");
         _trace.trace("Variable tmp7's value is " + tmp7.toString());
         _trace.trace("Variable Nb's value is " + Nb.toString());
         _trace.trace("PROCESS STUCK");
         throw new StuckException("Match failed: tmp7 is not equal to Nb.");
      }

      // nil
      _trace.trace("nil");
      return;

   }

}
```

Bibliography

[1] M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. Technical Report SRC Research Report 149, Digital Systems Research Centre, 1998.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[3] Ana Cavalcanti and Augusto Sampaio. From CSP-OZ to Java with Processes. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 161. IEEE Computer Society, 2002.

[4] Andrew Phillips and Susan Eisenbach and Daniel Lister. From Process Algebra to Java Code. In *FTfJP'02*. http://www.cs.kun.nl/ erikpoll/ftfjp/2002.html, June 2002.

[5] Benjamin Tobler. Sprite Website. University of Cape Town, http://people.cs.uct.ac.za/~btobler/sprite/.

[6] Benjamin Tobler and Andrew Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In *To Appear in Proceedings of CSES 2004 2nd International Workshop on Certification and Security in Inter-Organizational E-Services*. Kluwer, 2004.

[7] S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, Electronic Notes in Theoretical Computer Science, 2003. To appear.

[8] E. Borger and W. Schulte. A programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java, volume 1523 of Lect. Notes in Comp. Sci.*, pages 353–404. Springer-Verlag, 1999.

[9] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *2nd IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET '99)*. IEEE Computer Society, 1999.

[10] J. Bowen C.A.R. Hoare, H. Jifeng and P. Pandya. ESPRIT BRA 3104 ProCoS project: Provably Correct Systems. Technical report, Oxford University Computing Laboratory, 1990.

[11] CERT. CERT Advisory CA-2003-26 Multiple Vulnerabilities in SSL/TLS Implementations. http://www.cert.org/advisories/CA-2003-26.html.

[12] CERT. Microsoft private communication technology (pct) fails to properly validate message inputs. CERT, http://www.kb.cert.org/vuls/id/586540.

[13] CERT. Vulnerability Note VU#104280 Multiple vulnerabilities in SSL/TLS implementations. CERT, http://www.kb.cert.org/vuls/id/104280.

[14] Christopher Colby and Radha Jagadeesan and Konstantin Laufer and Lalita Jategaonkar Jagadeesan and Carlos Puchol. Design and Implementation of Triveni: a Process-algebraic API for Threads + Events. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, page 58. IEEE Computer Society, 1998.

[15] F. Crazzolara. *Language, Semantics, and Methods for Security Protocols*. PhD thesis, University of Aarhus, 2003.

[16] F. Crazzolara and G. Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.

[17] S. Crocker D. Eastlake and J. Schiller. Randomness Recommendations for Security, 1994.

[18] Riccardo Sisto Davide Pozza and Luca Durante. Spi2Java: Automatic Cryptographic Protocol Java Code Generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA'04) Volume 1*, page 400. IEEE, 2004.

[19] Adrian Perrig Dawn Xiaodong Song and Doantam Phan. AGVI - Automatic Generation, Verification, and Implementation of Security Protocols. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 241–245. Springer-Verlag, 2001.

[20] G. Denker. Design of a CIL connector to Maude. In N. Heintze H. Veith and E. Clarke, editors, *Workshop on Formal Methods and Comuter Security*. Carnegie Mellon University, July 2000.

[21] G. Denker and J. Millen. CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory, 1999.

[22] G. Denker and J. Millen. CAPSL integrated protocol environment, 2000. http://citeseer.nj.nec.com/denker00capsl.html.

[23] X. Didelot. COSP-J: A Compile for Security Protocols. Master's thesis, University of Oxford.

[24] D. Dolev and A.C. Yao. On the security of public key protocols. Technical Report STAN-CS-81-854, Stanford University, 1981.

[25] S. Drossopoulou and S. Eisenbach. Towards and operations semantics and type soundness for java.

[26] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[27] H. Abelson et al. *R5Rs Scheme Revised(5) Report on the Algorithmic Language Scheme*, March 2001.

[28] FIPS. Derived Test Requirements [DTR] for FIPS PUB 140-2, Security Requirements for Cryptographic Modules, 2002. http://csrc.nist.gov/cryptval/140-2.htm.

[29] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154. IEEE Computer Society, 2003.

[30] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *CSFW '00: Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, page 255. IEEE Computer Society, 2000.

[31] D.C. Hyde. Introduction to the programming language Occam, 1995.

[32] D. Caromel I. Attali and M. Russo. A formal executable semantics for java.

[33] Ian Sommerville. *Software Engineering (5th ed.* Addison Wesley, Redwodd City, CA, USA, 1995.

[34] G. Steele J. Gosling, B. Joy and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[35] B. Jacobs and F. Piessens. A Pi-Calculus Semantics for Java: The Full Definition, 2003.

[36] KDE. KDE Security Advisory: KDE 2.2 / Konqueror Embedded SSL vulnerability. http://www.kde.org/info/security/advisory-20030602-1.txt.

[37] R. Needham L. Gong and R. Yahalom. Reasoning about belief in cryptographic protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[38] Leonardo Freitas. JACK: A process algebra implementation in Java. Master's thesis, Centro de Informatica, Universidade Federal de Pernambuco, April 2002. http://www.cin.ufpe.br/ lf25.

[39] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[40] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[41] M. Abadi M. Burrows and R. Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[42] Mario Caccamo, Federico Crazzolara and Giuseppe Milicia. χ-Spaces From a model to a working language. Technical report, Aarhus University, 2002.

[43] S. McAlearney. Patches Issued for Multiple RADIUS Implementation Flaws. http://infosecuritymag.techtarget.com/2002/mar/digest07.shtml.

[44] Birgit Pfitzmann Michael Backes and Michael Waidner. A composable cryptographic library with nested operations (extend abstract), 2003. http://citeseer.ist.psu.edu/671432.html.

[45] Michael Burrows, Martin Abadi and Roger Needham. A logic of authentication. Technical Report Technical Report 39, Digital Systems Research Center, February 1989.

[46] Michael Möller. Specifying and Checking Java using CSP. Technical report, Universität Oldenburg, 2002.

[47] MicheleBugliesi, Ricardo Forcardi and Matteo Maffei. Authenticity by tagging and typing. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, pages 1–12, October 2004.

[48] J. Millen. nspk.cil. http://www.csl.sri.com/users/millen/capsl/jparse/nspk.cil.

[49] J. Millen. Prelimenary translator, CAPSL to CIL. http://www.csl.sri.com/users/millen/capsl/capsltrans.html.

[50] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[51] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100(1):1–40 and 41–77, 1992.

[52] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1998.

[53] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[54] C. R. Ramakrishnan Ping Yang and Scott A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *Lecture Notes in Computer Science*, pages 116–131, New York, NY, January 2003. Springer.

[55] E. Saul. Facilitating the Modelling and Automated Analysis of Cryptographic Protocols. Master's thesis, University of Cape Town, July 2001.

[56] B. Schneier. Cryptography: The importance of not being different, March 1999. http://www.schneier.com/crypto-gram-9904.html.

[57] SGS-THOMSON Microelectronics Limited. *Occam 2.1 reference manual*, 1995.

[58] SPORE. CCITT X.509 (3). http://www.lsv.ens-cachan.fr/spore/ccittx509_3.html.

[59] SUN. JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification. http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[60] Sun. Java Cryptography Architecture API Specification & Reference, 2004. http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html.

[61] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 1999.

[62] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999. http://isabelle.in.tum.de/Bali/papers/Springer98.html.

[63] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. http://www.schneier.com/paper-ssl.html.

[64] M.W. Whalen and M.P.E. Heimdahl. On the requirements of high-integrity code generation. In *Proceedings of the Fourth IEEE High Assurance in Systems Engineering Workshop*, November 1999.

[65] J. Wielemaker. SWI-Prolog 5.2.10 Reference Manual, 2003. http://www.swi.psy.uva.nl/projects/SWI-Prolog/Manual/.

[66] A.F. Yasinsac. *Evaluating Cryptographic Protocols*. PhD thesis, University of Virginia, 1996.