GENERATING NETWORK SECURITY PROTOCOL IMPLEMENTATIONS FROM FORMAL SPECIFICATIONS

Benjamin Tobler
Department of Computer Science, University of Cape Town
Private Bag, Rondebosh 7701, South Africa
btobler@cs.uct.ac.za

Andrew C.M. Hutchison
Department of Computer Science, University of Cape Town
Private Bag, Rondebosch 7701, South Africa
hutch@cs.uct.ac.za

Abstract We describe the Spi2Java code generation tool, which we have developed in an attempt to bridge the gap between formal security protocol specification and executable implementation. Implemented in Prolog, Spi2Java can input a formal security protocol specification in a variation of the Spi Calculus, and generate a Java code implementation of that protocol. We give a brief overview of the role of code generation in the wider context of security protocol development. We cover the design and implementation of Spi2Java which we relate to the high integrity code generation requirements identified by Whalen and Heimdahl. By defining a Security Protocol Implementation API that abstracts cryptographic and network communication functionality we show that protocol logic code can be separated from underlying cryptographic algorithm and network stack implementation concerns. The design of this API is discussed, particularly its support for pluggable implementation providers. Spi2Java's functionality is demonstrated by way of example: we specify the Needham-Schroeder Public Key Authentication Protocol, and Lowe's attack on it, in the Spi Calculus and examine a successful attack run using Spi2Java generated implementation of the protocol roles.

Keywords: Code generation, Formal methods, Java, Process algebra, Prolog, Security, Spi Calculus

Introduction

Formal methods have been widely and successfully used to specify network security protocols and analyse their security properties to ensure correctness M. Burrows and Needham, 1996; L. Gong and Yahalom, 1990; Lowe, 1995; Abadi and Gordon, 1998; Thayer et al., 1999. The same emphasis has, however, not been placed on the correctness of concrete implementations of these security protocol specifications. This is evident when one considers the number of security alerts issued for implementations of various security protocols. 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 and open source organizations OpenSSL, KDE and Apache CERT, a; CERT, c; KDE, ; CERT, b. It is clear then that security protocol research has been successful in verifying specifications, but that errors can still be introduced during implementation, leaving a gap between specification (formal and otherwise) and implementation.

In this paper we examine an approach to bridging this gap, by means of automatic code generation, in a manner that complements and integrates with the already existing formal methodologies for security protocol analysis.

Our approach entails the specification of a security protocol, in a variation of the Spi Calculus, which is used as input into our Spi2Java code generation tool. Spi2Java compiles the specification down to Java code that is a concrete implementation of the protocol.

Choosing the Spi Calculus as a specification language provides the benefits of formal specification: it allows the security protocol to be subject to analysis to ensure the desired security properties (i.e. one or more of *authenticity*, *confidentiality* and *integrity*) hold. Its formally defined semantics also provide a precise definition of the expected behaviour of the protocol, and so facilitates code generation and verification. These properties are particularly useful in helping to meet some of Whalen and Heimdahl's requirements for high-integrity code generation identified in Whalen and Heimdahl, 1999.

Regarding related work, we are aware of some other projects in this area: one on generating code from CAPSL specifications Millen and Muller, 2001, COSP-J Didelot, , AVGI Dawn Xiaodong Song and Phan, 2001 and another tool also called Spi2Java (which only came to our attention after the initial draft of this paper). COSP-J is based to some extent on Casper, a tool for converting fairly abstract security protocol specifications to CSP specifications, and produces Java code that implements protocols. Perrig et. al. briefly describe a tool for automatic security protocol implementation as part of AVGI in Dawn Xiaodong Song and Phan, 2001, however we have not been able to find further details of this tool in any available publications. Durante et. al. describe their own Spi2Java tool in Davide Pozza and Durante, 2004. They do not address the issue of the correctness of the generated code wrt Spi, nor do they discuss the implementation of the tool itself, i.e. how the translation from Spi to Java is performed. We have, as yet, not found any published work detailing verification or proof of correctness of automatically generated code that implements security protocols.

Though there is some overlap with these projects, we believe aspects of our Spi2Java tool and our continuing work on it, make some contribution to the area of code generation for automatic security protocols implementation. We use the Spi Calculus as input, allowing our Spi2Java to complement verification tools, such as the MMC model checker for the π and Spi Calculi Ping Yang and Smolka, 2003. Though abstraction of security functionality, e.g. Java's Cryptographic Extensions, is definitely not novel, our clean and complete separation of generated protocol logic code from cryptographic and network implementation specifics via an API provides even greater flexibility to the protocol implementor. Finally our continuing work towards meeting the requirements of high integrity code generation, specifically proving that our mapping from Spi to Java code segments preserves the Spi semantics in the Java code, will hopefully provide a high level of confidence in the correctness of the protocol logic implementation.

The layout of this paper is as follows: An overview of the security protocol development process is given indicating the role of a formal specification language throughout the process and emphasising the implementation and implementation verification phases of the process, where code generation can be used. We argue for the suitability of the Spi Calculus in the role of specification language, and define a variation of it to facilitate code generation. We cover the separation of protocol logic implementation from cryptographic algorithm and network communication implementation, by abstraction using the SPP API and the resulting benefits of this. We also define a mapping from Spi Calculus constructs to Java code segments and describe the code generation tool we have developed in Prolog that defines rules for these mappings. We look at verification of the tool and the generated code as well as current work validating the mappings from Spi Calculus constructs to Java code. Finally, we conclude the paper by assessing the contribution of this approach to bridging the gap between security protocol specification and implementation.

1. Security Protocol Development

Given that developing protocols to provide network security is a specialisation of software development in general, a security protocol development process could be described as follows (see Figure 1):

Requirements: Like any system, there may be requirements unrelated to security. However the requirements of interest in this paper are the desired se-

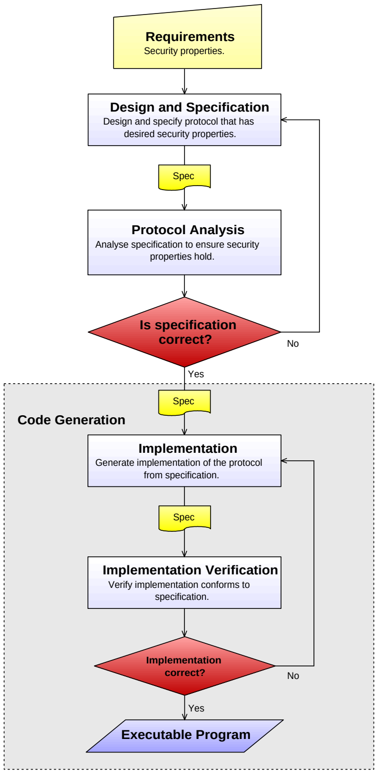

Requirements
Security properties.
Design and Specification
Design and specify protocol that has desired security properties.
Spec
Protocol Analysis
Analyse specification to ensure security properties hold.
Is specification correct?
No
Yes
Code Generation
Spec
Implementation
Generate implementation of the protocol from specification.
Spec
Implementation Verification
Verify implementation conforms to specification.
Implementation correct?
No
Yes
Executable Program

Figure 1. Security protocol development process.

curity properties of the protocol - authentication, confidentiality and integrity. For brevity we view these requirements as input to the development process as opposed to a phase.

Design and Specification: A protocol that attempts to meet the requirements is designed and specified. The number of messages exchanged, message contents and the cryptographic mechanisms employed will all depend on the security properties stated in the requirements.

Protocol Specification Analysis: The protocol specification is analyzed, potentially by means of inference logics, e.g. BAN M. Burrows and Needham, 1996 and GNY L. Gong and Yahalom, 1990 and attack analysis techniques, e.g. strand-space analysis Thayer et al., 1999 and model checking Lowe, 1996; Denker, 2000 to determine whether the required security properties hold. If not, the analysis results are fed back to the design and specification phase and the protocol is modified or redesigned. Otherwise the process can progress to the implementation phase.

Implementation: The protocol is implemented by, either manually or automatically, generating code that can be compiled to an executable program that conforms to the specification.

Implementation Verification: Two approaches can be taken to ensure the correctness of the concrete implementation. In both cases the ideal is to be able to prove that the generated code is a refinement of the security protocol specification. Either the generated code for each protocol must be verified to ensure equivalence to the specification, or the translation process must be proven correct and the mappings from specification language constructs to implementation code must shown to preserve the specification language semantics Whalen and Heimdahl, 1999.

This paper is concerned mainly with the last two phases: implementation and implementation verification, as indicated in the boxed area of Figure 1. These phases produce the output of the development process: an executable program that conforms to the security protocol specification for which the desired security properties hold.

Obviously it is desirable, and convenient, to be able to generate a protocol implementation from a specification defined in the same language used for analysing the correctness of the specification. This avoids the possibility of protocol semantics being lost in the translation from one specification language to another, and helps preserve them during the implementation phase. To that end we have selected the Spi Calculus to use as input for our code generator.

2. The Spi Calculus

The Spi Calculus, defined by Abadi and Gordon in Abadi and Gordon, 1998, consists of a number of terms, such as names, variables, encrypted terms and pairs, and some simple processes which incorporate actions that include sending and receiving messages, decryption, term matching and parallel composition. Each of the processes has a simple, well defined behaviour. Despite it small size and relative simplicity, the Spi Calculus (and the π-calculus it is based on) is powerful in its ability to both describe, and reason about, the behaviour of concurrent systems Milner et al., 1992 - of which security protocols are a special case.

The behaviour of the processes of the π-calculus is defined formally by transition semantics in Milner et al., 1992. These semantics are extended in Abadi and Gordon, 1998 to define the behaviour of constructs that the Spi-Calculus introduces to model cryptographic operations such as symmetric and asymmetric encryption and message digests. These formal definitions provide a basis for reasoning about the security properties of protocols specified in the Spi Calculus as demonstrated in Abadi and Gordon, 1998. As formal specifications describe the expected behaviour of a protocol explicitly, they also provide a superior guide for the implementor, whether automated or human.

In the context of program refinement, as described by Morgan in Morgan, 1998, Spi fulfills the roles of specification - it is high level abstraction that facilitates understanding - and to some extent code - executable instructions generally in an imperative language. Spi provides an abstraction that allows security protocols to be easily understood. It also serves as code in the sense that it defines executable behaviour for all its processes, as unlike the π-calculus it is based on, Spi does not define the non-executable binary summation process that specifies that a process P can behave as either process Q or R arbitrarily. In this context Spi2Java is essentially a compiler for the Spi Calculus.

Variation for Code Generation

To facilitate code generation we define a few variations to the standard Spi Calculus. Firstly, some minor syntactic changes are defined to allow security protocols to be described in plain text files. These include using the *!* and *?* characters to indicate output and input respectively - as in Occam PAP, 1995. We also 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 the Security Protocol Language described in Crazzolara, 2003. A process that checks the validity of a timestamp is also defined.

Secondly, only a subset, albeit a comprehensive one, of the terms and processes defined by the original calculus are supported: the successor term and integer case and replication processes are not supported.

Syntax and Semantics

A brief description of the syntax of the Spi Calculus variant and an informal description of the behaviour of its processes is given below for convenience. Apart from the variations we have defined, the following is just a summary of the description in Abadi and Gordon, 1998, which also contains the formal definition of the language.

An infinite set of *names* and an infinite set of *variables* over those names are assumed. Names range over principal identifiers, nonces, keys and other values. Letting c, m, n, p and r range over names and x, y and z over variables, the terms are defined by the grammar:

$L,\ M,\ N\ ::=$

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

and the processes by:

$P,\, Q ::=$

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

The behaviour of these processes is described informally as follows:

- $c!\langle N\rangle.P$ will output N on channel c when an interaction with an input process occurs, and then run as P.
- $c?(x).P$ will input a term, say N, on channel c when an interaction occurs and then run as $P[N/x]$ i.e. P with N substituted for all free occurrences of x.
- $(n)P$ creates a new, private name n and behaves like P. This process is used to model the generation of nonces.
- $[M\; is\; N]P$ behaves like P if the term M is the same as the term N or else it does nothing.
- nil 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.
- $case\, L\, of\; \{x\}N\, in\, P$ runs as $P[M/x]$ if M is L decrypted with with N, otherwise it does nothing.
- $case\, T\, valid\, in\, P$ runs as P if the timestamp T is valid otherwise does nothing.

To accommodate implementation, a preamble declaring variable types is specified. The supported types are *channel*, *encryption*, *hash*, *id*, *key*, *nonce*, *term* (generic or compound value) and *time*.

3. Protocol Specification Example

As an example of protocol specification using the Spi Calculus we specify the Needham-Schroeder Public Key Authentication protocol. We first give the generally used standard notation version, which does not have formally defined semantics, and then a specification in Spi.

$$
\begin{array}{ll}
1 & A \rightarrow B : \{n, A\}pub(B) \\
2 & B \rightarrow A : \{n, m\}pub(A) \\
3 & A \rightarrow B : \{m\}pub(B)
\end{array}
$$

This informal description, though simple and fairly intuitive, leaves the specification of most protocol actions implicit. The burden is on the protocol implementor to use her experience and understanding of security protocols, to determine the sequence of programmatic actions that implement this protocol correctly and with all of the designer's intended semantics. In particular, this example demonstrate the failure of the standard notation to explicitly specify when nonces should be instantiated and whether, and how, values in a received messages should be verified.

In contrast the Spi specification of the same protocol indicates exactly when nonces should be instantiated and which received values to verify and how.

The Spi specification defines a process for the initiator role in the protocol:

$$
\begin{array}{ll}
channel\, c & encryption\, l \\
id\, A, B, y & nonce\, m, n, x \\
term\, j &
\end{array}
$$

$$
\begin{array}{lll}
Init(A, B, c) & = & (n) \\
 & & c!\langle\{n, A\}pub(B)\rangle. \\
 & & c?(l). \\
 & & case\, l\, of\, \{j\}priv(A)\, in \\
 & & let\, (x, m) = j\, in \\
 & & [x\, is\, n] \\
 & & c!\langle\{m\}pub(B)\rangle. \\
 & & nil
\end{array}
$$

This process states explicitly when the initiator should generate the nonce *n* to challenge the responder, and how the first nonce in the message returned by the responder, indicated by the variable *x*, should be matched against it.

The responder process is specified as follows:

$$
\begin{array}{ll}
channel\, c & encryption\, l, k \\
id\, B, x & nonce\, m, n, y \\
term\, j &
\end{array}
$$

$$\begin{array}{lll}
Resp(B,c) & = & c?(l). \\
 & & case\, l\, of\ \{j\}priv(B)\, in \\
 & & let\,(n,\, x)\ =\ j\, in \\
 & & (m) \\
 & & c!\langle\{(n,\, m)\}pub(x)\rangle. \\
 & & c?(k). \\
 & & case\, k\, of\ \{y\}priv(B)\, in \\
 & & [y\, is\, m] \\
 & & nil
\end{array}$$

Like the initiator process, it also explicitly defines the generation of a challenge nonce and verification of the initiator's response to the challenge.

A run of the protocol is specified by the parallel execution of the initiator and responder processes:

$$channel\, c \quad id\, A,\ B$$

$$\begin{array}{l}
NSRun(A,\, B,\, c) = \\
\qquad (Init(A,\, B,\, c) \mid Resp(B,\, c))
\end{array}$$

where c is a channel allowing A and B to communicate.

Lowe's Attack on the Needham-Schroeder Protocol

Even if the implementation is faithful to the specification, an attacker can successfully masquerade as a legitimate participant in the Needham-Schroeder protocol, described by Lowe in Lowe, 1995, as follows:

$$\begin{array}{llll}
1 & A & \rightarrow\ C: & \{n,\, A\}pub(C) \\
2 & C(A) & \rightarrow\ B: & \{n,\, A\}pub(B) \\
3 & B & \rightarrow\ C(A): & \{n,\, m\}pub(A) \\
4 & C & \rightarrow\ A: & \{n,\, m\}pub(A) \\
5 & A & \rightarrow\ C: & \{m\}pub(C) \\
6 & C(A) & \rightarrow\ B: & \{m\}pub(B)
\end{array}$$

where C is the attacker who leads B to erroneously believe that he is communicating with A, when in fact he is talking to C.

In Spi this attacker role is specified as follows:

$$\begin{array}{ll}
channel\, cA,\ cB & encryption\, k,\ l,\ p \\
id\, A,\ B,\ C & nonce\, m,\ n,\ x \\
term\, j &
\end{array}$$

$$
\begin{aligned}
&Attack(cA,\, cB,\, B,\, C) \;=\\
&\qquad cA?(l).\\
&\qquad case\, l\, of\; \{j\}priv(C)\, in\\
&\qquad let\,(n,\, A) \;=\; j\, in\\
&\qquad cB!\langle(n,A)pub(B)\rangle.\\
&\qquad cB?(p).\\
&\qquad cA!\langle p\rangle.\\
&\qquad cA?(k).\\
&\qquad case\, k\, of\; \{m\}priv(C)\, in\\
&\qquad cB!\langle mpub(B)\rangle.\\
&\qquad nil
\end{aligned}
$$

and a run of successful attack can be specified by:

$$channel\, cAC,\, cBC \quad id\, A,\, B,\, C$$

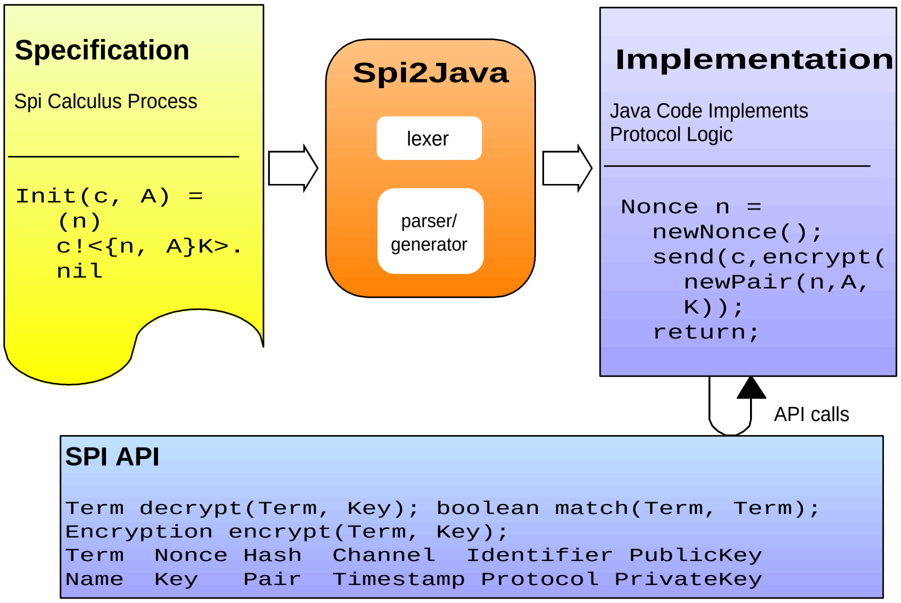
$$
\begin{aligned}
NSRun(A,\, B,\, c) = \;& Init(A,\, C,\, cAC)\,|\\
& Attack(cAC,\, cBC,\, B,\, C)\,|\\
& Resp(B,\, cBC)
\end{aligned}
$$

where cAC is a channel for communication between A and C, and cBC is a channel for communication between C and B. The use of two separate channels allow the attacker to control communication between A and B at the network level. This approach models the Dolev-Yao attacker capabilities Dolev and Yao, 1981, where an attacker is able to intercept and remove messages sent by the legitimate protocol participants, as well as introduce new messages onto the network.

4. The Spi2Java Code Generator

Rules Based Implementation

Spi2Java is implemented in Prolog using the Definite Clause Grammar rules supported by most Prolog engines Wielemaker, 2003. 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”* Whalen and Heimdahl, 1999, 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. Whether or not the mappings preserve the semantics of Spi in the Java code is another matter, broached later in this paper.



```
Init(c, A) =
   (n)
   c!<{n, A}K>.
   nil
```

```
Nonce n =
  newNonce();
  send(c,encrypt(
    newPair(n,A,
    K));
  return;
```

```
Term decrypt(Term, Key); boolean match(Term, Term);
Encryption encrypt(Term, Key);
Term   Nonce Hash   Channel  Identifier PublicKey
Name   Key   Pair   Timestamp Protocol PrivateKey
```

Figure 2. Code generation using Spi2Java.

It is important to note 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 Whalen and Heimdahl, 1999; C.A.R. Hoare and Pandya, 1990.

The SPP API

In our approach we separate the implementation of the protocol logic from that of the cryptographic algorithms and network communications. We define:

Protocol Logic as the code that maintains protocol state, determines when and if messages are sent and received, the contents of outgoing messages, the expected contents of incoming messages, storing message components and and determining which incoming message components to verify and what components they should be verified against.

and

Cryptographic and Communications Provider: as provider specific code that handles the packing and unpacking of message components into byte streams, implements cryptographic algorithms (e.g. symmetric and asymmetric encryption and message digests) and manages network protocol specific aspects (e.g. message packing and unpacking, network addresses of principals and message transport).

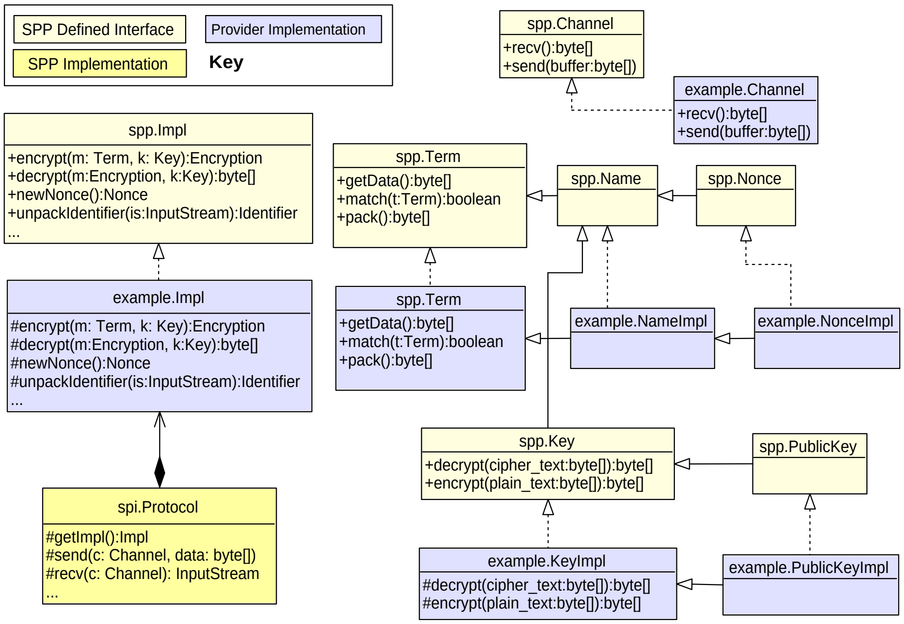


Figure 3. UML class diagram of a subset of the SPP API.

Spi2Java generates code that implements the protocol logic. This code makes calls to the Security Protocol Primitives (SPP) API that abstracts the low level cryptographic and network communications details. The bridge design pattern is employed here to decouple the provider specific cryptographic and communication implementation from the protocol logic code. SPP defines a set of interfaces for message components such as nonces, principal identifiers and keys as well as for cryptographic and communications operations (see Figure 3).

The abstract factory method design pattern is used allowing different providers to be plugged into the API. It gives the protocol logic code a single point of access to instances of the concrete provider classes that implement the SPP defined interfaces for value types and cryptographic and network operations.

The flexibility of this approach means that providers can be changed without affecting the protocol logic code generated by Spi2Java. For example: a provider that implements RSA asymmetric and AES symmetric encryption, with a stream message and component format using a TCP/IP stack for communications can be swapped for a provider that implements elliptic curve asymmetric and triple DES symmetric encryption, a bitmap message format and X.25 network communications, with a simple change of a configuration or command line option.

Code Generation

Spi2Java emits code for each Spi process definition in a method that is named after that definition. The Java code segments for all binding Spi processes, i.e. those that result in the substitution of a name value for a variable, are emitted inline. For example the Spi pair splitting process, $let\,(x,\,y) = M\,in\,P$, maps to the Java code segment (discussed in detail later):

```
// let (x, y) = M in
final <TypeX> x;
final <TypeY> y;
{
   InputStream _is =
     new ByteArrayInputStream(
        M.getData());
    x = getImpl().unpackNonce(
       _is);
    y = getImpl().unpackIdentifier(
       _is);
}
// Code for process P...
```

Making a method call, instead of emitting code inline, to implement this process would look something like

```
// let (x, y) = M in
<TypeX> x;
<TypeY> y;
split(&x, &y, M);
// Code for process P...
```

in C++. However this is not possible in Java as it does not have language support for “out parameters” (or pointers to pointers in C++ parlance) which would be required to assign values to the variables *x* and *y*. Using wrapper classes to encapsulate these variables, in lieu of out parameters, is a possible alternative, but would introduce an extra level of complexity to the code, as well as cluttering it with method calls instantiate wrappers and to set and get values from the wrapper instances.

Names of temporary variables required by some code segments are re-used in inlined code segments throughout the emitted code. Re-use is possible by declaring and operating on temporary variables in locally nested scopes. The nested scopes are introduced by a new Java block declared by the “{” and “}” symbols, as demonstrated in the pair splitting code segment listing. This approach is preferable to the alternatives: inlining code without nesting it in a new scope, requiring the use of arbitrarily large numbers of temporary variable names, which introduces extra logic to the code generator in order track them, or making method calls, which introduces new scope, but for the reasons discussed previously is not a viable alternative.

Using methods calls instead of inlined code segment would also break the safe practice of declaring all Java variables that correspond to Spi variables as *final*, meaning they can only be assigned to once, which faithfully implements the Spi model of processes substituting variables with name values as they run.

Spi Process to Java Mapping

Spi2Java uses the mappings defined in this subsection to generate Java code from Spi processes. As mentioned, the specification of these mappings is properly defined by the Prolog rules which associate each Spi process type with a Java code segment template. The Prolog direct clause grammar rules manage parsing the Spi processes, and each such rule has sub-rules that generate the Java code segment to be emitted from the associated code template.

Spi2Java also emits code to trace the protocol progress and state by means of updating a user interface. The interface also provides the user with a way control the run of protocol role, by stepping through the Spi processes that define it. In the Java code segment listings that follow we omit the tracing code for the purposes of clarity and brevity.

Output, $c!\langle N\rangle.P$, maps to a call to the *void spi.Protocol.send(spp.Channel, spp.Term)* method with the parameters channel *c*, the channel for communication, and term *N*, the term to be communicated. The code emitted for this process is thus:

```
// c!<N>.
send(c, N);
// Code for process P...
```

The parameters *c* and *N* may be Java variables, or expressions, that evaluate to values of types *spp.Channel* and *spp.Term* respectively.

Input, $c?(x).P$, maps to a call to the *Term spi.Protocol.recv(spp.Channel)* method, which is passed *c* as the channel parameter and returns a type that is of, or extends, *spp.Term*. The emitted code is of the form:

```
// c?(x).
final <Type> x =
   getImpl().unpack<Type>(recv(c)));
// Code for process P...
```

where <*Type*> is the type of the variable *x* and is determined by a lookup table that the code generator creates from the specification's variable type preamble.

Restriction, (n), is used to model nonce and timestamp creation and maps to a call to either *spp.Nonce spi.Protocol.newNonce()* or *spp.Timestamp spi.Protocol.newTimestamp()* depending on the type, again determined from the lookup table, of the variable *n*. Thus if *n* is of type *nonce* the code generator emits code of the form:

```
// (n)
Nonce n = newNonce();
// Code for process P...
```

or, if *n* is of type *time*, code of the form:

```
// (n)
Timestamp n = newTimestamp();
// Code for process P...
```

to implement this process.

Term Matching, $[MisN]P$, is mapped to a return statement that is guarded by checking a call to *boolean spi.Protocol.match(Term, Term)*:

```
// [m is n]
if (!match(m, n))
{
   return;
}
// Code for process P...
```

The method *boolean spi.Protocol.match(Term, Term)* is implemented as follows:

```
protected final boolean match(
   final Term a,
   final Term b)
{
   return a.match(b);
}
```

The method relies on the correctness of the provider's implementation of the *boolean spp.Term.match(Term)*method on the *spp.Term* interface. Implementations must return a boolean value indicating whether the parameter the method is called with, is equal to the instance it is called on.

Pair Splitting, $let\,(x,\,y)\;=\;M\,in\,P$, 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 segment for this process is thus:

```
// let (x, y) = M in
final <TypeX> x;
final <TypeY> y;
{
   InputStream _is =
     new ByteArrayInputStream(
        M.getData());
    x = getImpl().unpackNonce(
        _is);
    y = getImpl().unpackIdentifier(
        _is);
}
// Code for process P...
```

where <*TypeX*> and <*TypeY*> are the types of the variables *x* and *y* respectively.

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

```
// case L of {x}N in
final <Type> x =
    getImpl().unpack<Type>(
        decrypt(L, N));
// Code for process P...
```

5. Implementation Example

To demonstrate Spi2Java we generate an implementation from the Spi specification of the Needham-Schroeder protocol, and the attack on it, given earlier. We list some sample generated code and then look at a run of a successful attacked on the protocol using the generated code for the initiator, responder and attacker roles.

The generated code for the *Init* process, which specifies the initiator role of the protocol, is given by the listing in Figure 4. Again, code generated for tracing and state monitoring, and some generated comments, have been omitted. Some minor formatting changes have also been made to facilitate typesetting.

Figure 5 shows a screenshot of the traces and final states of concurrent initiator, responder and attack runs of the Needham-Schroeder protocol roles. The implementation of each role's process prints the time of execution and the specification of each action to the trace window. Each process also updates the state window whenever a variable is substituted with a value.

The screenshots - showing the final states of the initiator, responder and attacker runs - clearly demonstrate that the attacker has subverted the protocol by gaining possession of the nonces m and n which the responder believes to be suitable shared secrets between himself and the initiator. Thus the attacker can masquerade as the initiator. Should the responder base the confidentiality and/or authenticity of continued communication with the party he believes to be initiator, the attacker will be able to continue this charade.

6. Current Work

While we have addressed the issue of verifying Spi2Java in terms of correctly performing the specified mappings from Spi to Java code, the issue of the correctness of those mappings needs to be resolved. We are currently working on showing that the mappings preserve the Spi semantics in the Java code and will correct any mapping definitions that fail to do so.

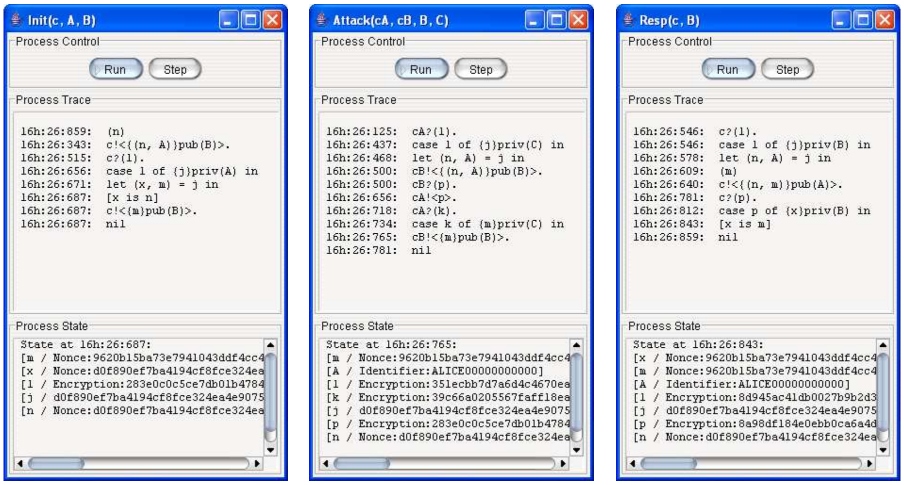
Our approach will follow the refinement methodology and involve setting up, and satisfying, proof obligations for each Java code segment. This entails

```
// Init(c, A, B) =
public void Init(final Channel c, final Identifier A,
    final Identifier B)
{
    final Nonce n = newNonce(); // (n)
    // c!<{(n, A)}pub(B)>.
    send(c, encrypt(newPair(n, A), pub(B)));
    // c?(l).
    final Encryption l = getImpl().unpackEncryption(
        recv(c));
    // case l of {j}priv(A) in
    final Term j = getImpl().unpackTerm(
        decrypt(l, priv(A)));
    // let (x, m) = j in
    final Nonce x; final Nonce m;
    {
        InputStream _is = new ByteArrayInputStream(
            j.getData());
        x = getImpl().unpackNonce(_is);
        m = getImpl().unpackNonce(_is);
    }
    // [x is n]
    if (!match(x, n)) return;
    // c!<{m}pub(B)>.
    send(c, encrypt(m, pub(B)));
    // nil
    return;
}
```

Figure 4. Generated code for the Needham-Schroeder initiator role.

relating Spi semantics to those of the Java language. While there is no official formal semantics for the Java language, an Abstract State Machine (also referred to as evolving algebras) semantics has been defined in Borger and Schulte, 1999. We intend to use this definition of Java for this process along with the transition semantics defined for the Spi Calculus.

Successfully completing this task will allow us to meet the second requirement for high integrity code generation identified in Whalen and Heimdahl, 1999 which states "*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.*"



```
16h:26:859:  (n)
16h:26:343:  c!<{(n, A)}pub(B)>.
16h:26:515:  c?(l).
16h:26:656:  case l of {j}priv(A) in
16h:26:671:  let (x, m) = j in
16h:26:687:  [x is n]
16h:26:687:  c!<{m}pub(B)>.
16h:26:687:  nil
```

```
State at 16h:26:687:
[m / Nonce:9620b15ba73e7941043ddf4cc4
[x / Nonce:d0f890ef7ba4194cf8fce324ea
[l / Encryption:283e0c0c5ce7db01b4784
[j / d0f890ef7ba4194cf8fce324ea4e9075
[n / Nonce:d0f890ef7ba4194cf8fce324ea
```

```
16h:26:125:  cA?(l).
16h:26:437:  case l of {j}priv(C) in
16h:26:468:  let (n, A) = j in
16h:26:500:  cB!<{(n, A)}pub(B)>.
16h:26:500:  cB?(p).
16h:26:656:  cA!<p>.
16h:26:718:  cA?(k).
16h:26:734:  case k of {m}priv(C) in
16h:26:765:  cB!<{m}pub(B)>.
16h:26:781:  nil
```

```
State at 16h:26:765:
[m / Nonce:9620b15ba73e7941043ddf4cc4
[A / Identifier:ALICE00000000000]
[l / Encryption:351ecbb7d7a6d4c4670ea
[k / Encryption:39c66a0205567faff18ea
[j / d0f890ef7ba4194cf8fce324ea4e9075
[p / Encryption:283e0c0c5ce7db01b4784
[n / Nonce:d0f890ef7ba4194cf8fce324ea
```

```
16h:26:546:  c?(l).
16h:26:546:  case l of {j}priv(B) in
16h:26:578:  let (n, A) = j in
16h:26:609:  (m)
16h:26:640:  c!<{(n, m)}pub(A)>.
16h:26:781:  c?(p).
16h:26:812:  case p of {x}priv(B) in
16h:26:843:  [x is m]
16h:26:859:  nil
```

```
State at 16h:26:843:
[x / Nonce:9620b15ba73e7941043ddf4cc4
[m / Nonce:9620b15ba73e7941043ddf4cc4
[A / Identifier:ALICE00000000000]
[l / Encryption:8d945ac41db0027b9b2d3
[j / d0f890ef7ba4194cf8fce324ea4e9075
[p / Encryption:8a98df184e0ebb0ca6a4d
[n / Nonce:d0f890ef7ba4194cf8fce324ea
```

Figure 5. Concurrent runs of Spi2Java generated implementations of the Needham-Schroeder participant roles and attack.

7. Conclusion

In this paper we have described our approach to bridging the gap between security protocol specification and implementation using the Spi2Java code generation tool. We have shown that by a using formal specification language, the Spi Calculus, and implementing Spi2Java in a logic programming language, Prolog, we can progress towards the ultimate goal of meeting Whalen and Heimdahl's requirements for high integrity code generation. Further to this goal, the approach to our current work - validating the refinement of the Spi Calculus processes to Java code - is briefly outlined. Despite this validation not being complete, we list the mappings from Spi processes to Java code segments that Spi2Java currently uses. This acts not only as a reference for the potential user but also to highlight the relative simplicity of the mappings.

The separation of protocol logic implementation from cryptographic and network specific implementation concerns by the SPP API, contributes to implementation correctness, by allowing Spi2Java to be focused on just the protocol logic aspect and not on lower level abstractions which would make the generated code far more complex.

Finally we demonstrate the potential of Spi2Java by using it to implement not only the legitimate roles of the Needham-Schroeder protocol, but also the attacker role described by Lowe. These implementations are executed concurrently to give a trace of a successful run of the attack on the protocol.

We believe that the above indicates that using a more formal and automated approach to implementing network security protocols simplifies the process and reduces the potential for errors. Hence it adds value to the process of security protocol development as a whole.

References

(1995). *Occam 2.1 reference manual*. SGS-THOMSON Microelectronics Limited.

Abadi, M. and Gordon, A. (1998). A Calculus for Cryptographic Protocols: The Spi Calculus. Technical Report SRC Research Report 149, Digital Systems Research Centre.

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

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

CERT. CERTő Advisory CA-2003-26 Multiple Vulnerabilities in SSL/TLS Implementations.

CERT. Microsoft private communication technology (pct) fails to properly validate message inputs.

CERT. Vulnerability Note VU#104280 Multiple vulnerabilities in SSL/TLS implementations.

Crazzolara, F. (2003). *Language, Semantics, and Methods for Security Protocols*. PhD thesis, University of Aarhus.

Davide Pozza, R. S. and Durante, L. (2004). Spi2Java: Automatic Cryptographic ProtocolJava Code Generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA'04) Volume 1*, page 400. IEEE.

Dawn Xiaodong Song, A. P. and Phan, D. (2001). 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.

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

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

Dolev, D. and Yao, A. (1981). On the security of public key protocols. Technical report, Stanford University.

KDE. KDE Security Advisory: KDE 2.2 / Konqueror Embedded SSL vulnerability.

L. Gong, R. N. and Yahalom, R. (1990). Reasoning about belief in cryptographic protocols. In Cooper, D. and Lunt, T., editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society.

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

Lowe, G. (1996). 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.

M. Burrows, M. A. and Needham, R. (1996). 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.

Millen, J. and Muller, F. (2001). Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100(1):1–40 and 41–77.

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

Ping Yang, C. R. R. and Smolka, S. A. (2003). 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. Springer.

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

Whalen, M. and Heimdahl, M. (1999). On the requirements of high-integrity code generation. In *Proceedings of the Fourth IEEE High Assurance in Systems Engineering Workshop*.

Wielemaker, J. (2003). SWI-Prolog 5.2.10 Reference Manual.