
Matteo Avalle, Politecnico di Torino, Italy *
Alfredo Pironti, INRIA, France
Davide Pozza, Teoresi Group, Italy
Riccardo Sisto, Politecnico di Torino, Italy

Abstract
This paper presents JavaSPI, a “model-driven” development framework that allows the user to reliably develop security protocol implementations in Java, starting from abstract models that can be verified formally. The main novelty of this approach stands in the use of Java as both a modeling language and the implementation language. The JavaSPI framework is validated by implementing a scenario of the SSL protocol. The JavaSPI implementation can successfully interoperate with OpenSSL, and has comparable execution time with the standard Java JSSE library.

Keywords: Formal methods; Java; ProVerif; Model-driven development

I. Introduction
Security protocols are distributed algorithms that run over untrusted networks with the aim of achieving security goals, such as mutual authentication of two protocol parties. In order to achieve such goals, security protocols typically use cryptography. It is well known that despite their apparent simplicity it is quite difficult to design security protocols right, and it may be quite difficult to find out all the subtle flaws that affect a given protocol logic. Research on this topic has led to the development of specialized formal methods that can be used to rigorously reason about protocol logic and to prove that it does really achieve its intended goals under certain assumptions, e.g. (Blanchet, 2009). One problem that remains with this solution is the gap that exists between the abstract protocol model that is formally analyzed and its concrete implementation written in a programming language. The latter may be quite different from the former, thus breaking the validity of the formal verification when the final implementation is considered. In order to solve this problem two approaches have been proposed. On one hand, model extraction techniques, e.g. (O'Shea, 2008; Bhargavan, Fournet, Gordon, & Tse, 2008; Backes, Maffei, & Unruh, 2010; Chaki & Datta, 2009), automatically extract an abstract protocol model that can be verified formally, starting from the code of a protocol implementation. On the other hand, code generation model-driven techniques, e.g. (Pironti & Sisto, 2007; Kiyomoto, Ota, & Tanaka,
2008; Almeida, Bangerter, Barbosa, Krenn, Sadeghi, & Schneider, 2010; Bhargavan, Corin, Deniélou, Fournet, & Leifer, 2009; Balser, Reif, Schellhorn, Stenzel, & Thums, 2000; Song, Perrig, & Phan, 2001), automatically generate a protocol implementation, starting from a formally verified abstract model. In either case, if the automatic transformation is formally guaranteed to be sound, it is possible to extend the results of formal verification done on the abstract protocol model to the corresponding implementation code.

Model-driven development (MDD) offers the advantage of hiding the complexity of a full implementation during the design phase, because the developer needs only focus on a simplified abstract model. Moreover, since the implementation code is automatically generated, it is possible to make it immune from some low-level programming errors, such as memory leakages, that could make the program vulnerable in some cases but that are not represented in abstract models.

However, MDD usually requires a high level of expertise, which limits its adoption, because formal languages used for abstract protocol models are generally not known by code developers, and quite different from common programming languages. For example, the user needs to know the formal spi calculus language in order to properly work with the Spi2Java framework (Pironti & Sisto, 2007).

Our motivation is to solve this problem and make MDD approaches more affordable. To achieve this, our contribution is the proposal of a new framework, based on Spi2Java, called JavaSPI, where the abstract protocol model is itself an executable Java program.

This little but significant difference grants several different improvements over frameworks like Spi2Java:

- it is not necessary to learn a new completely different modeling language anymore (Java is also used as a modeling language);
- standard Java Integrated Development Environments (IDEs), to which the programmer is already familiar, can be used to develop the security protocol model like it was a plain Java program, making full use of IDE features such as code completion, or live compilation;
- it is possible to debug (or simulate) the abstract model using the same debuggers Java programmers are used to;
- thanks to Java annotations, information about low-level implementation choices and security properties can be neatly embedded into the abstract model.

The viability of the proposed approach is validated by a case study where interoperable client and server sides of a specific SSL scenario are implemented. The interoperability capabilities are demonstrated by running alternatively the client and the server against the OpenSSL 0.98o corresponding implementations, while the performances of the generated code are compared against the Oracle’s Java official implementation of SSL contained in the JSSE library.

The rest of the paper is organized as follows. Section II analyzes related work and Spi2Java in particular, highlighting its main limitations. Then, section III illustrates the JavaSPI framework in detail, while section IV reports about the SSL case study. Finally, section V concludes.

Available online at http://typhoon5.polito.it/javaspi/
II. Background and related work

Model-driven development of security protocols based on formal models has been experimented using various languages and tools. One of the most comprehensive approaches is Spi2Java, which enables semi-automatic development of interoperable Java implementations of standard protocols (Pironti & Sisto, 2007).

In this framework, protocols are modeled in spi calculus, a formal domain-specific process algebraic language. A spi calculus protocol model can be automatically analyzed in order to formally verify that there are no possible attacks on the protocol under the modeling assumptions made. For this to be done, the protocol expected goals must be formally specified too. The formal analysis can be done, for example, by the automatic theorem prover ProVerif (Blanchet, 2009), whose input language is a superset of spi calculus.

Once the abstract model has been successfully analyzed, and it has been shown that it is free from logical flaws, it can be refined up to the point that a Java implementation can be derived for each protocol role. During this refinement step, the abstract model must be enriched with all the missing protocol aspects that are needed in order to get a concrete and interoperable Java implementation:

(i) concrete Java implementations of cryptographic algorithms with their actual parameters
(ii) Java types to be used for terms
(iii) concrete binary representations of messages and corresponding Java implementations of marshaling functions

In the Spi2Java framework, the spi calculus model and this refinement information are kept in two separate but coupled files. When a change to the model is done, it is under the user’s responsibility to keep the coupled refinement file up to date, which is error prone and time consuming. By keeping refinement information neatly integrated in the source code as Java annotations, JavaSPI also solves these engineering issues.

In addition to Spi2Java, other approaches based on code generation are documented in literature, e.g. (Kiyomoto, Ota, & Tanaka, 2008; Almeida, Bangerter, Barbosa, Krenn, Sadeghi, & Schneider, 2010; Bhargavan, Corin, Denielou, Fournet, & Leifer, 2009; Balser, Reif, Schellhorn, Stenzel, & Thums, 2000; Song, Perrig, & Phan, 2001), but they present the same or larger limitations.

Other researchers have explored the model extraction approach e.g. (O’Shea, 2008; Bhargavan, Fournet, Gordon, & Tse, 2008; Backes, Maffei, & Unruh, 2010; Chaki & Datta, 2009). These techniques, like JavaSPI, do not expose the programmer to specialized formal specification languages, but they lack the model-driven approach, so that all the code must be written manually by the programmer.

For example, the Elyjah framework (O’Shea, 2008) requires a full Java implementation to be developed, before a model can be extracted and verified. In contrast, with JavaSPI, the programmer only writes a simplified Java model of the protocol, from which a code generator generates the full implementation. The abstract Java model developed with JavaSPI enables features such as symbolic execution of the protocol, and the use of Java annotations keeps implementation and verification details neatly separated from the Java model. These features are...
inherently difficult to achieve in Java model-extraction frameworks such as Elyjah. In (Bhargavan, Fournet, Gordon, & Tse, 2008), model extraction is performed on full implementations written in F#. The F# implementation can be linked either to a concrete or to a symbolic library of cryptographic and communication primitives, which enables protocol symbolic simulation, just like when the JavaSPI abstract Java model is executed. However, in (Bhargavan, Fournet, Gordon, & Tse, 2008) there is no neat distinction between protocol logic and lower-level details such as cryptographic algorithms and parameters or data marshaling. Moreover, in (Bhargavan, Fournet, Gordon, & Tse, 2008) programs are written in F#, which is far less known than Java, thus making the tool of lesser impact to common developers.

Other researchers have focused on different model-driven approaches, starting from UML representations of security protocols e.g. (Jürjens, 2005; Basin, Doser, & Lodderstedt, 2006). While UML modeling is agreed to be an essential design phase in very large scale software projects, it is often the case that the UML modeling overhead is deemed too expensive for the typical application size of a security protocol, thus being not accepted by the average security protocol implementer.

III. The JavaSPI Framework

The main contributions of this paper are the development of the JavaSPI framework and also the implementation of a case study, which will be described later in Section IV. JavaSPI has been designed as a set of tools and utilities that enable the user to model a cryptographic protocol by following the workflow shown in Figure 1: basically, the user is intended to develop abstract models in the form of typical Java applications, but using a specific library which is part of the JavaSPI framework, named SpiWrapperSim, which contains a set of basic data types along with the networking and cryptographic primitives.

![Figure 1: The complete workflow provided by JavaSPI](image-url)
The logical execution of the protocol can be simulated by simply debugging the abstract code. The protocol security properties can be formally verified by using the JavaSPI Java-ProVerif converter that produces an output compatible with the ProVerif tool. Once a model has been properly designed, it can be refined by adding implementation information by means of Java annotations, as defined in the SpiWrapperSim library. From the annotated Java model a concrete implementation of the protocol can be generated by using the JavaSPI Java-Java converter. The entire JavaSPI framework and tools described in this paper have been completely developed from scratch: still, some architectural choices have been made to allow re-use of parts of the Spi2Java framework.

**III-A. Developing the abstract model**

The JavaSPI framework includes a Java library, called SpiWrapperSim, which can be used to write abstract security protocol models as Java applications and to simulate them. Models that can be expressed in this way are instances of the class of models that can be described by the input language of ProVerif. Based on this, the framework provides the Java-ProVerif tool that transforms a Java model into the corresponding ProVerif model, which can be analyzed by ProVerif. Note that differently from (Bhargavan, Fournet, Gordon, & Tse, 2008), here the ProVerif model is not extracted from the Java code, rather the model, expressed in the Java syntax, is translated into the ProVerif syntax. A Java model differs from the final Java implementation because it is as abstract as the ProVerif model. Moreover, the Java model can also be executed like any regular Java application. Its execution in fact simulates the underlying model that it describes, thus giving the user the possibility to debug the abstract model. In this execution messages are represented symbolically, and input/output operations are implemented by exchanging symbolic expressions over in-memory channels behaving according to the classical spi calculus semantics.

In order to get a Java program that models a protocol in this way, the user must write Java according to a particular programming pattern. Only the SpiWrapperSim library can be used for cryptographic and input/output operations, and some restrictions on the Java language constructs that can be used for the description of each process apply. These restrictions, documented in the library JavaDoc, naturally lead the user to develop models in the right way. A protocol role (a “process”) is represented by a class that inherits from the library class spiProcess. In this way, the common code needed for simulation that is the context of the protocol algorithm is hidden inside the superclass. Moreover, objects derived from spiProcess are allowed to use some protected methods that enable common operations, like the parallel instantiation of sub-processes. The class that inherits from spiProcess must define the doRun() method, which is the abstract description of the protocol role. Any message, complex at will, can be represented by an immutable object belonging to a class that inherits from the Packet library class. The fields of this class are the fields of the message. The class must be made immutable by declaring all fields as final. This is necessary as, in spi calculus, each variable can
be bound only once. Using mutable Java objects would be possible but it would then entail more complex relationships between the Java code and the underlying model.

The only class types the user is allowed to instantiate are the ones provided by the SpiWrapperSim library, plus the ones used as arguments of methods of such classes (e.g. `String`). The primitive type `int` is also admitted, but only for loop flow control, with the constraint that each loop must be bounded and the bound must be known at compile time.

Conditional statements are possible only with equality tests (via the `equals()` method) and with tests on the return values of certain operations of the library. SpiWrapperSim is very similar to the SpiWrapper library that provides the implementations of the spi calculus cryptographic and communication operations in the Spi2Java framework. This is a precise architectural choice that greatly facilitates the last development step, i.e. the refinement of the abstract model into a concrete implementation. Indeed, the implementation code is based on the SpiWrapper library.

**Java abstract model**

```java
1  Message m = new Identifier("Secret Message");
2  Nonce n = new Nonce();
3  SharedKey s = new SharedKey(n);
4  SharedKeyCiphered<Message> mk =
    new SharedKeyCiphered<Message>(m, s);
```

**Java concrete implementation**

```java
1  Message m = new IdentifierSR("Secret Message");
2  Nonce n = new NonceSR("8");
3  SharedKey s = new SharedKeySR(n, "DES", "64");
4  SharedKeyCiphered mk =
    new SharedKeyCipheredSR(m, s, "DES",
    "1234567801g=", "CBC",
    "PKCS5Padding", "SunJCE");
```

**ProVerif model**

```java
1  new m1;
2  new n2;
3  let s4 = SharedKey(n2) in
4  let mk6 = SymEncrypt(s4, m1) in
```

Figure 2: An example of how four lines of the abstract model are converted into the corresponding concrete implementation and ProVerif syntax

As showed in Figure 2, thanks to this choice even the syntax used in the two codes is very similar; the main difference is that the abstract model lacks many implementation details, like the encryption algorithms of each cryptographic function call, or the marshaling functions (whose implementation is included in the “SR”-suffixed classes in the example showed).

Within the SpiWrapperSim library a set of annotations was also developed, which can be used during refinement to assign, for each object, its implementation details. As annotations do not affect the simulation phase, they
can be specified later on, just before generating the concrete implementation.

By using this technique the implementation details and the code both reside on
the same file: this means that JavaSPI is not affected by the sync problems
described previously for Spi2Java. Moreover, each annotation has a scope and a
default value, so that it is not necessary to specify each implementation detail for
each object used in the code, but it is possible to specify just the implementation
details that differ from the default values.

By following the intended workflow, the Java model can be converted into a
ProVerif compatible model, or a concrete Java implementation can be derived
from the Java model.

The next two subsections will cover these two cases.

III-B. Java-ProVerif conversion and formal verification

The mapping from Java to ProVerif syntax is based on simple rules, developed in
this work along with the corresponding converter, that are informally
exemplified in Table I. Each Java statement that may occur in a doRun method is
mapped to a corresponding ProVerif equivalent piece of code. For simplicity, the
table does not include the name mangling algorithm, which is needed in order to
disambiguate variable names in ProVerif, and whose outcome is shown in Figure
2.

Table I: A significant portion of the conversion mapping between the Java model and ProVerif model.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Java</th>
<th>ProVerif</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh</td>
<td>Type a = new Type();</td>
<td>New a;</td>
</tr>
<tr>
<td>Assign</td>
<td>Type a = b;</td>
<td>let a = b in</td>
</tr>
<tr>
<td>Hashing</td>
<td>Hashing a = new Hashing(b);</td>
<td>let a = H(b) in</td>
</tr>
<tr>
<td>Send</td>
<td>cAB.send(a);</td>
<td>out(cAB, a);</td>
</tr>
<tr>
<td>Receive</td>
<td>Type a = cAB.receive(Type.class);</td>
<td>in(cAB, a);</td>
</tr>
<tr>
<td>SharedKey</td>
<td>SharedKey key = new SharedKey(a);</td>
<td>let key = SharedKey(a)</td>
</tr>
<tr>
<td>Encrypt</td>
<td>SharedKeyCiphered a = new SharedKeyCiphered&lt;Type&gt;(b, key);</td>
<td>let a = SymEncrypt(key, b) in</td>
</tr>
<tr>
<td>Decrypt</td>
<td>Type a = b.decrypt(key);</td>
<td>let a = SymDecrypt(key, b) in</td>
</tr>
<tr>
<td>Error handled</td>
<td>ResultContainer&lt;Type&gt; c = Let b = a.decrypt_w(key);</td>
<td>SymDecrypt(key, a)</td>
</tr>
<tr>
<td>Decipher</td>
<td>if (c.isValid()) {</td>
<td>in (</td>
</tr>
<tr>
<td></td>
<td>... } else { ... }</td>
<td>) else ( ... )</td>
</tr>
<tr>
<td>Packet Comp.</td>
<td>PacketType m = new PacketType(a, b, ...)</td>
<td>let m = (a, b, ...) in</td>
</tr>
<tr>
<td>Packet Split</td>
<td>Type a = b.getField();</td>
<td>let a = b_getField in (*)</td>
</tr>
<tr>
<td>Match Case</td>
<td>If (a.equals(b)) {</td>
<td>(Client(c, d, ...)</td>
</tr>
<tr>
<td></td>
<td>... } else { ... }</td>
<td>Server(e, f, ...)</td>
</tr>
<tr>
<td>Start</td>
<td>SpiProcess a = new Client(c, d, ...);</td>
<td>(Client(c, d, ...)</td>
</tr>
<tr>
<td></td>
<td>SpiProcess b = new Server(e, f, ...);</td>
<td>Server(e, f, ...)</td>
</tr>
<tr>
<td></td>
<td>start(a, b);</td>
<td>start(a, b);</td>
</tr>
</tbody>
</table>
Type stands for any class name, PacketType stands for any user-defined Packet class name, Field stands for any field name in a Packet class, while a... f and key stand for variable names.

(*) Variable b_getField is created in ProVerif code during a Packet splitting phase which is automatically generated after any Decrypt or Receive statement that produces a Packet object.

Conversion of loops requires special handling. ProVerif does not support unbounded loops natively, but they can be easily encoded as recursive processes. However, ProVerif often experiences termination problems when loops encoded as recursive processes are used. Because of this limitation of the verification engine, the restriction of having only bounded loops was introduced in the Java modeling language, so that the conversion tool can perform loop unrolling in order to eliminate loops.

The fields of a Java Packet object are translated into nested pairs. In order to facilitate code translation and readability, a new variable is introduced in ProVerif for each field. For example, let us consider a class called MyPacket with three fields called a, b and c, all of type Nonce: the following Java code receives a message of type MyPacket and extracts its three fields.

```java
MyPacket p = channel.receive(MyPacket.class);
Nonce a = p.getA();
Nonce b = p.getB();
Nonce c = p.getC();
```

This group of four Java instructions is converted into the following ProVerif code:

```proverif
in(channel1, p2);
(* Packet expansion *)
let p2_getA3 = GetLeft(p2);
let tmp4 = GetRight(p2);
let p2_getB5 = GetLeft(tmp4);
let p2_getC6 = GetRight(tmp4);
(* Variable assignment *)
let a7 = p2_getA3;
let b8 = p2_getB5;
let c9 = p2_getC6;
```

By using this technique the converter is forced to write, in ProVerif, more code lines than with the Java syntax, but this disadvantage is overcome by the fact that this technique totally hides to ProVerif the additional complexity that custom packet types could cause, thus avoiding the risk to generate diverging code.

There is also another particular characteristic of ProVerif which actually needs to be taken into consideration: its syntax does not allow writing any expression after an if/else statement. This poses some limits to the Java-ProVerif conversion, as it generates some situations in which a simple rule-based mapping is not feasible.

The naïve solution of forbidding the users to write Java code after an if/else statement is not acceptable, because it would limit the expressiveness freedom a Java developer usually has and exploits. For this reason, a pre-parsing Java algorithm has been developed, to inline all the Java code appearing after an if/else branch, so that it can be more easily mapped to ProVerif syntax...
statements. This operation, again, generates a ProVerif file that can be more complex than the Java model, but this can be considered an acceptable tradeoff, as in this way it is not necessary to limit the developer too much. Moreover, ProVerif files are not meant to be read by any developer. They just need to be used with the corresponding verification tool.

Translating plain Java models into ProVerif is not enough to enable automatic verification of security properties: two types of information need to be added:

- The initial attacker knowledge
- The security properties that have to be checked.

By default, the initial attacker knowledge is automatically generated this way: constants shared by the communication actors are considered public constant data, and the communication channels are considered public free names. However sometimes some communication protocols may work in a slightly different way for various reasons: for example, two actors may share a common secret symmetrical key which must be considered unknown to the attacker.

For this reason, the user can have control over the initial attacker knowledge, by overriding the default behavior by means of a single annotation, called @pVarDef(PRIVATE|PUBLIC). This annotation can be scoped to a single variable or to an entire block of code: in this case every variable declared inside the code block inherits the pVarDef property of the block, unless a more specific, inner-scoped annotation affects the variable declaration.

With these simple rules it is possible to express very complex initial attacker knowledge bases with a very small effort: in fact, in a simple protocol, the files that model the actor behaviors do not need these annotations. The pVarDef annotation is just added to the instancer process, by defaulting its variables as PUBLIC. Changing this behavior just implies adding few annotations on some variables in the instancer, when these variables must be considered PRIVATE.

Note that the pVarDef annotation has a direct influence on how the ProVerif code is generated: every PUBLIC variable is declared as a free or constant term (whether the variable is a channel or any other data type), which are particular elements globally available throughout the entire protocol code. As this behavior is not logically the same of the Java model, a particular variable renaming technique has been applied in order to avoid name conflicts.

A specific annotation set has been developed within the JavaSPI library to express security properties. These annotations are then processed during conversion to ProVerif and translated into corresponding queries in the output ProVerif code.

A variable can be marked as @Secret in order to specify that ProVerif should verify its syntactic secrecy. For instance:

```java
@Secret
Nonce DHPrivate = new Nonce();
```

The corresponding ProVerif generated code will look like this:

```proverif
(* Secrecy queries *)
query attacker:DHPrivate21.
```

If the @Secret term is a compound term or anyway a term that needs to be constructed over another one, the translation becomes slightly more complex: in fact, as ProVerif cannot directly verify the secrecy of variables, but only of fresh names or terms built upon them, the ProVerif query that will be generated regards the entire composition of the term, along with queries about the secrecy of any ground term involved in the composition. For this reason, during
verification some false alerts may be reported by ProVerif, for example when a
complex secret term is composed of a mix of secret and publicly available terms:
in such cases the secrecy verification of public terms will certainly fail.
In the current version of the JavaSPI framework the task of recognizing such
false alerts and safely ignore them is left to the user. In fact, the bigger goal of
interpreting ProVerif output to consistently report it to the user into the JavaSPI
environment is a major ongoing effort that is scheduled to be included in the
next version of the framework. Within this bigger goal, automatic recognition of
such false alerts is a planned feature.
In order to verify authentication properties, instead, it is possible to use
correspondence assertions. In JavaSPI, a process can raise an event by calling the
event(String name, Message... data) method provided by the SpiProcess class,
where name specifies the name of the event, and data the set of variables
associated to that event.
This method has no effect in the code, but it is translated to a corresponding
 event in ProVerif. Once the event sets are defined it is possible to use them to
write some interrogations: for example, the reachability of every event, which
increases the confidence in the model correctness, is automatically queried,
while in order to check other more complex properties a set of annotations was
developed: for example, the correspondence between events, such as “if
\textit{event(n1,xy,...)} happened, then \textit{event(n2,xy,...)} must have happened before” can
be specified by the \texttt{@PEvinj} annotation, associated with the instantiation process
class:
\begin{verbatim}
    @PEvinj(\{"n1", "n2"\})
    public class Master extends SpiProcess ... 
\end{verbatim}
This technique can be used to write more advanced queries, by extending the
number of events in a \texttt{PEvinj} clause to three or more, or by combining multiple
\texttt{PEvinj} annotations by using another annotation, called \texttt{PInjList}, like in this
example:
\begin{verbatim}
    @PInjList({
        @PEvinj(\{"n1","n2","n3"\}),
        @PEvinj(\{"m1", "m2"\})
    })
\end{verbatim}
As a design choice, all the queries are written by using just the name of the
events, without referencing also the data associated with the event calls (as
opposed to what ProVerif does): by default, the exact comparison of all the
parameters will be verified. This design choice allows queries to be stated in a
very simple form, even if it slightly reduces the overall expressive power. Note
that this slightly reduced expressiveness did not prevent nor made more
complex the development of the SSL case study.
With this set of techniques a user can express the main part of basic ProVerif
queries. There is still the possibility, however, that the user needs to write a
more complex interrogation, not expressible with just these annotations. For this
reason a particular annotation has been provided to enable the user to directly
write a custom query with the ProVerif syntax. This, however, is an advanced
feature that can just be used by experienced developers who actually know the
ProVerif query syntax: for this reason, it is a feature of little interest for the
purposes of this paper, and it will not be discussed in more detail.
**III-C. Implementation generation**

The last development stage is the automatic generation of the protocol implementation code from the model. As SpiWrapperSim is similar to the library used for the concrete implementation, there is a strict correspondence between the abstract code (the model) and the concrete code (the implementation). The implementation aspects that are missing in the abstract model can all be specified by means of annotations.

One of such aspects is the choice of the marshaling functions to be used for each object. A default marshaling mechanism based on Java serialization is provided by a library called spiWrapperSR, which extends spiWrapper. The user however can provide custom implementations of the marshaling functions. This is a key factor enabling development of interoperable implementations of standard protocols, where the specific marshaling functions to be used are specified by the protocol standard.

Another key feature of JavaSPI enabling interoperability is the ability of resolving Java annotations values either statically at compile time, or dynamically at run time, like in this example:

```java
Identifier algorithm = channel.receive(Identifier.class);
@Algo(Type=Types.varname, value="algorithm")
SharedKey k = new Sharedkey(n);
```

Here it is possible to notice how the algorithm name for the key “k” is not directly hardcoded in an annotation, but this value will change at run time by assuming the value of the “algorithm” variable.

This technique is particularly useful to implement, as example, protocols featuring algorithms negotiation.

The last thing that needs to be performed is to specify how the various constants of the protocol have to be initialized. Since in general different actors of a protocol may need different constants, the user can specify, for each actor, a piece of code that initializes every parameter before calling the protocol method in the proper way.

The initialization code must be written into the `doInit` method, which overrides the one in the `SpiProcess` class. The code inside `doInit` is neither considered during simulation nor in ProVerif verification, but it will be replicated verbatim in the concrete Java implementation. This technique avoids the need of modifying the generated code at all. To integrate the generated code into a bigger security-aware application only its interfaces will be needed.

**IV. The SSL case study**

In order to validate the proposed JavaSPI approach, a simplified but interoperable implementation of both the client and server sides of the SSL handshake protocol has been developed.
The considered scenario, depicted in Figure 3, can be logically divided into four different phases:

1. Client and server exchange two “hello” messages which are used to negotiate protocol version and ciphersuites.
2. The server authenticates itself to the client by sending its certificate \( s\_cert \).
3. Diffie-Hellman (DH) key exchange is performed; note that the server DH parameters are signed by the server.
4. Finally, the session is completed by the exchange of encrypted “Finished” messages.

For simplicity, in the considered scenario both the developed client and server only support version 3.0 of the protocol with DSA server certificate. Other ciphersuites or other protocol features such as session resumption or client authentication are not considered. Indeed, the goal is to validate the methodology with a minimal, yet significant example, rather than provide a full reference implementation of the SSL protocol.

The SpiWrapperSim library has been used to develop the abstract model of the SSL protocol. This includes eight new Packet classes representing the structures of the different types of exchanged messages and a client and a server SpiProcess classes. In addition, an “instancer” process called Master that just runs an instance of client and server in parallel has been added in order to simulate protocol execution. Figure 4 provides a code excerpt of the Java SSL model, while the complete version of the code is available online².

² At the address: http://typhoon5.polito.it/javaspi
Server.java

class Server extends SpiProcess {
    @Override void doRun(final Channel c,
                           final Identifier SSL_VERSION3_0, ...)
    {
        final Pair<Identifier, DHHashing> c_key_exch =
            c.receive(Pair.class);
        final DHHashing c_DHy = c_key_exch.getRight();
        final Triplet PMSp =
            new Triplet(c_DHy, DH_x, DH_P);
        final DHHashing common_key =
            new DHHashing(PMSp);
        ...
    }
}

Master.java

class Master extends SpiProcess {
    @Override void doRun() {
        ...
        final Client c = new Client( ... );
        final Server s = new Server( ... );
        start(c, s);
    }
}

Figure 4: An excerpt of the SSL protocol abstract model.

After defining the model the following properties have been expressed and
successfully verified:

- Secrecy of the client and server DH secret values.
- Server authentication, expressed as an injective correspondence between
  the correct termination of the two processes: each time a client correctly
  terminates a session, agreeing on all relevant session data and the server
  identity, a server must have started a session, agreeing on the same
  session data and on the server identity.

Finally, in order to get interoperability, a custom marshaling library compliant
with the SSL standard has been developed.

Besides setting the marshaling layer, it was also necessary to specify by means of
annotations the needed cryptographic details, such as algorithms and related
parameters. In the sample SSL protocol both compile time and run time
resolution features of JavaSPI have been exploited. Even if this protocol
implementation uses many “hardcoded” parameters, like the ciphersuites and
the key strengths, other information is only known at run time: for example, the
initialization vectors used for shared key encryption are calculated from the
shared secret, thus they change at each run.

As shown by the code excerpt in Figure 5, static details can be specified once, on
the head of the class, while dynamic details and special cases are specified just
before each variable needing them. In the sample code, the initialization vector is
computed by applying a hash function and is stored in variable c_write_iv. Then,
an annotation specifies that the initialization vector for the ciphered message
received in variable c_encrypted_Finish is the value in variable c_write_iv.
@SharedKeyA(Algo="3DES", Strength="168")
@SharedKeyCipheredA("Algo="3DES", Mode="CBC")

public class Server extends spiProcess {
    ...  
    final Hashing c_write_iv = new Hashing(PA3);
    ...  
    @Iv(type=Types.varName, value="c_write_iv")
    final SharedKeyCiphered
        <Pair<Pair<Hashing, Hashing>, Hashing>>
        c_encrypted_Finish =
        c.receive(SharedKeyCiphered.class);
    ...  

Figure 5: An excerpt of the Java model with annotations on it.

The amount of required annotations does not burden the code too much: the SSL example required about 60 annotations in total (client + server), which amounts to about 10% of the whole model size. To make this measure significant, default values were not crafted to suite the SSL example, rather the scoping feature of annotations was exploited, so that SSL-specific default values could be annotated just once at the class scope.  
The generated client and server implementations have been successfully tested for interoperability against OpenSSL 0.9.8o.

Performance considerations
One claimed disadvantage of code generation techniques is that as the code is automatically generated it will never be as optimized as it is possible to do by manually writing the code. Nonetheless, with cryptographic protocols it is often the case that the main computing effort lies in the computation of cryptography: for this reason the possible overhead due to potential code inefficiency is often negligible with respect to the overall computing time. In order to experimentally confirm this claim, we compared the performance of the SSL client implementation generated with JavaSPI to the performance of a corresponding code into the JSSE library, which is the Oracle’s Java official implementation of SSL. The two codes have been executed against the same SSL server, based on the OpenSSL application. To ensure that the two clients are effectively performing the same operations, a custom Certificate validator has been written for the JSSE implementation in order to treat the certificates in the same way they are treated by the JavaSPI SSL implementation. As a further check, some network packet sniffing has been preliminarily performed in order to ensure that the same ciphersuites were used, and the same messages were exchanged. Finally, in order to run the two applications in the same environment and limit random components in the measurements, the tests were run keeping every communication local, thus eliminating random network latencies. Moreover, the two implementations were run in the same Java virtual machine a thousand times and the mean execution time and its standard deviation were computed. Since in the first run a Java program is affected by the Java class loader latency, the time of the first run has been excluded, while all other measurements have been used to compute mean and standard deviation values.
The obtained results demonstrates that the processing times of the two implementations are nearly the same; the performance difference between the two implementations is just about 5% in favor of the JSSE code. As stated before, the explanation of this fact is that both the pieces of code are using exactly the same cryptographic library (JCA) and the DSA signature check and DH modular exponentiation performed in the SSL protocol take the main part of the total protocol execution time. It is likely that the JSSE implementation is much more optimized than the JavaSPI auto-generated code, but this performance boost just affects a very small portion of the total execution time. In conclusion, the performance results show us a very small difference between an optimized version of the code, written by hand, and an automatically generated implementation. This inefficiency might be considered non negligible in some particular cases, but in any other situation having an implementation with an high level of trustworthiness and correctness can greatly balance this small performance penalty.

V. Conclusion

The JavaSPI framework enables model-driven development of security protocols based on formal methods without the need to know specialized formal languages. Knowledge of a formal language is replaced by knowledge of a Java library and of a set of language restrictions, which is easier to learn for Java experienced programmers. Moreover, standard IDEs can be used to develop the Java model, with the benefit of having access to all the development features offered by such IDEs. This could potentially enable any common developer to perform formal verifications of security protocols. Even if this can be considered a usability improvement, in some situations this can be considered a dangerous feature, as developers who are completely new to the modeling world could generate wrong formal proofs by simply verifying wrong queries. Anyway, even in these environments, JavaSPI could still be of great use as it simplifies the interaction between modelers and developers by forcing the two categories to speak “the same language”.

The proposed approach, along with the provided toolchain and libraries, enables (i) interactive simulation and debugging of the Java model, via standard Java debuggers available in all common IDEs; (ii) automatic verification of the protocol security properties, via the de-facto standard ProVerif tool; and (iii) automatic generation of interoperable implementation code, via a custom tool, driven by Java annotations embedded into the model files. Compared to similar frameworks, like Spi2Java, JavaSPI is easier to use, while retaining the nice feature of enabling fast development of protocol implementations with high integrity assurance given by the linkage between Java code and verified formal models. Future work includes focusing on the formalization of the relationship between Java and spi calculus semantics, in order to get a soundness proof for the Java code, once the ProVerif model is verified. From an engineering point of view, porting the ProVerif verification results directly to the Java model and better engineering the way security properties are expressed in Java could further improve usability and accessibility of the
proposed framework. Moreover, further tests could be performed in order to demonstrate that quite every Java developer is able to design and validate a communication protocol by just reading the framework documentation.

References


