The JavaSPI Framework for Security Protocol Implementation

Matteo Avalle, Alfredo Pironti, Riccardo Sisto, Davide Pozza
Dip. di Automatica e Informatica
Politecnico di Torino, Torino, Italy
Email: {matteo.avalle, alfredo.pironti, riccardo.sisto, davide.pozza}@polito.it

***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. By using the SSL handshake protocol as a reference example, this paper illustrates the JavaSPI framework.**

***Keywords*-Formal methods; Java; Security protocols; 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 a protocol logic and to prove that it does really achieve its intended goals under certain assumptions (e.g. [1]).

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. [2], [3]) 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. [4], [5]) 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 [4].

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*[1], 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* 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 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.

II. Background and Related Work

Model-driven development of security protocols based on formal models has been experimented using various

[1]Available online at http://staff.polito.it/riccardo.sisto/javaSPI/

languages and tools. One of the most comprehensive approaches is Spi2Java, which enables semi-automatic development of interoperable Java implementations of standard protocols [4].

This framework models protocols in spi calculus, a formal process algebraic language. With this language it is possible to write an abstract model of a protocol which can be automatically analyzed in order to formally verify that there are no possible attacks on the protocol under the modeling assumptions made. Of course, this requires that the protocol expected goals be formally specified too. The analysis can be done, for example, by the automatic theorem prover ProVerif [1], which can work on spi calculus.

Once the abstract model has been successfully analyzed, and it has been shown that it is free from logical flaws, 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; and (iii) concrete binary representations of messages and corresponding Java implementations of marshaling functions.

The Spi2Java framework also requires the user to manually edit and keep in sync the model and an intermediate XML file containing refinement information, which is error prone and time consuming. By keeping refinement information neatly integrated 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. [5]), but they present the same or larger limitations.

Other researchers have explored the model extraction approach (e.g. [2], [3]). 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, in [2], a full Java implementation must be provided, before a model can be extracted. In contrast, with JavaSPI, the programmer only writes a simplified Java model of the protocol, from which a code generator generates the full implementation.

In [3], 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 [3] there is no neat distinction between protocol logic and lower-level details such as cryptographic algorithms and parameters or data marshaling. Moreover, in [3] programs are written in F#, which is far less known than Java, thus making the tool of lesser impact to common developers.

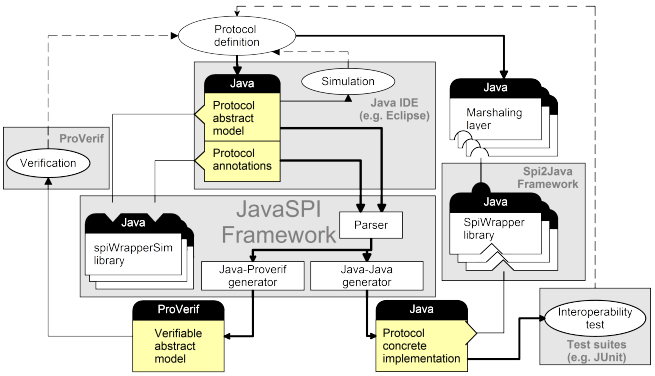

Figure 1. The complete workflow provided by JavaSPI.

Other researchers have focused on different model-driven approaches, starting from UML representations of security protocols (e.g. [6], [7]). 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

JavaSPI has been developed as a set of tools and utilities which enables 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.

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 described in this paper has been completely developed from scratch: still, some architectural choices have been made to allow re-use of parts of the Spi2Java framework.

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 [3], 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 use 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 surrounds 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 control flow, 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.

Java abstract model

```
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

```
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

```
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.

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.

As it is possible to notice in Figure 2, thanks to this choice even the syntax used in the two codes is very similar; the main difference is just 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 shown).

The SpiWrapperSim library also provides a set of annotations 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 to 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.

Table I
A SIGNIFICANT PORTION OF THE CONVERSION MAPPING BETWEEN THE JAVA MODEL AND PROVERIF MODEL.

Statement	Java	ProVerif
Fresh	**Type** $a = new$ **Type** ();	$new\ a$;
Assign	**Type** $a = b$;	$let\ a = b\ in$
Hashing	$Hashing\ a =$ $new\ Hashing(b)$;	$let\ a =$ $H(b)\ in$
Send	$cAB.send(a)$;	$out(cAB, a)$;
Receive	**Type** $a =$ $cAB.receive($ **Type**$.class)$;	$in(cAB, a)$;
SharedKey	$SharedKey\ key =$ $new\ SharedKey(a)$;	$let\ key =$ $SharedKey(a)\ in$
Encrypt	$SharedKeyCiphered$ $<$ **Type** $> a = new$ $SharedKeyCiphered$ $<$ **Type** $> (b, key)$;	$let\ a =$ $SymEncrypt$ $(key, b)\ in$
Decrypt	**Type** $a =$ $b.decrypt(key)$;	$let\ a =$ $SymDecrypt$ $(key, b)\ in$
Error handled Decipher	$ResultContainer$ $<$**Type** $> c =$ $a.decrypt\_w(key)$; $if(c.isValid()\{$ **Type** $b =$ $c.getResult()$; $...\}else\{...\}$	$let\ b =$ $SymDecrypt$ $(key, a)\ in\ ($ $...$ $)else($ $...$ $)$
Packet Comp.	$\mathbf{PacketType}\ m = new$ $\mathbf{PacketType}(a, b, ...)$	$let\ m =$ $(a, b, ...)\ in$
Packet Split	**Type** $a =$ $b.get\mathbf{Field}()$;	$let\ a =$ $b\_get\mathbf{Field}\ in\ \ (*)$
Match case	$if(a.equals(b))\{$ $...\}else\{...\}$	$if\ a = b\ then($ $...)else(...)$
Start	$SpiProcess\ a =$ $new\ Client(c, d, ...)$; $SpiProcess\ b =$ $new\ Server(e, f, ...)$; $start(a, b)$;	$(Client(c, d, ...)\|$ $Server(e, f, ...))$

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.

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 figure does not consider the addition of the numeric suffix in ProVerif, needed in order to disambiguate variable names, as shown in Figure 2.

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 Java code

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

that receives a message of type MyPacket and extracts its three fields is converted into the following ProVerif code:

```
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.

Translating plain Java models into ProVerif is not enough to enable automatic verification of security properties. Indeed, the formal specification of the security properties to be proved must be given to ProVerif.

The JavaSPI library provides a specific annotation set for expressing security properties in the Java model. 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 secrecy, in this way:

```
@Secret SharedKey DHx = new SharedKey(pl);
```

The corresponding ProVerif generated code will look like this:

```
(* Secrecy queries *)
query attacker:DHx53.
```

Authentication can be expressed instead as correspondence assertions on the order of events. In JavaSPI, a process can rise 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 data associated to that event. This method has no effect in the code, but it is translated to a corresponding event in ProVerif. Finally, correspondence between events, such as "if $event(n1, x)$ happened, then $event(n2, x)$ must have happened before"

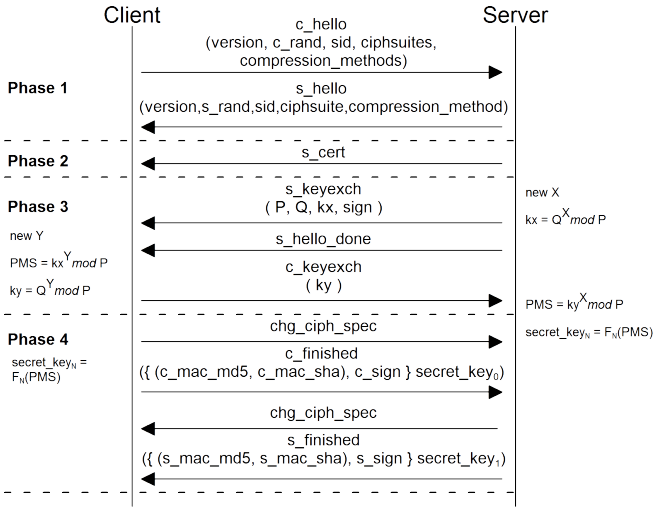

Figure 3. SSL message exchange in the selected scenario.

Server.java

```
class Server extends SpiProcess { ...
 @Override void doRun(final Channel c,
   final Identifier SSL_VERSION_3_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.

can be specified by a specific annotation associated with the instantiation process class.

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 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. For example, this enables implementations of protocols featuring algorithm negotiation.

IV. The SSL Case Study

In order to provide a validation example of 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.

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

```
@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.

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 the server identity.

Finally, in order to grant 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 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, any static detail can be specified once, on the head of the class, while the dynamic details and the special cases are specified in front of each variable that needs 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`.

The amount of required annotations does not burden the code: 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, few default values have been used; in other words, default values where not crafted to suite the SSL example.

The generated client and server implementation have been successfully tested for interoperability against OpenSSL 0.9.8o.

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.

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 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

[1] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, 2009.

[2] N. O'Shea, "Using Elyjah to analyse Java implementations of cryptographic protocols," in *Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, 2008, pp. 221–226.

[3] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, pp. 1–61, 2008.

[4] A. Pironti and R. Sisto, "An experiment in interoperable cryptographic protocol implementation using automatic code generation," in *IEEE Symposium on Computers and Communications*, 2007, pp. 839–844.

[5] S. Kiyomoto, H. Ota, and T. Tanaka, "A security protocol compiler generating C source codes," in *Information Security and Assurance*, 2008, pp. 20–25.

[6] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[7] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: from UML models to access control infrastructures," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 39–91, 2006.