Spi2Java: automatic cryptographic protocol java code generation from spi calculus

Original
Spi2Java: automatic cryptographic protocol java code generation from spi calculus / Pozza, Davide; Sisto, Riccardo; Durante, L.. - STAMPA. - 1:(2004), pp. 400-405. (18th International Conference on Advanced Information, Networking and Applications Fukuoka, Japan March 2004) [10.1109/AINA.2004.1283943].

Publisher:
IEEE

Published
DOI:10.1109/AINA.2004.1283943

Spi2Java: Automatic Cryptographic Protocol Java Code Generation from spi calculus

Davide Pozza, Riccardo Sisto
Politecnico di Torino
Dip. di Automatica e Informatica
c.so Duca degli Abruzzi 24
I-10129 Torino (Italy)
Davide.Pozza, Riccardo.Sisto@polito.it

Luca Durante
IEIIT - CNR
c/o Politecnico di Torino
c.so Duca degli Abruzzi 24
I-10129 Torino (Italy)
Luca.Durante@polito.it

Abstract

The aim of this work is to describe a tool (Spi2Java) that automatically generates Java code implementing cryptographic protocols described in the formal specification language spi calculus. Spi2Java is part of a set of tools for spi calculus, also including a pre-processor, a parser, and a security analyzer. The latter can formally analyze protocols and detect protocol flaws. When a protocol has been analyzed and an adequate confidence about its correctness has been reached, Spi2Java can generate a corresponding correct Java implementation of the protocol, thus dramatically reducing the risk of introducing security flaws in the coding phase.

1 Introduction

One of the most challenging practical problems in modern computer science is how to ensure design and implementation correctness of security protocols. The role of such protocols is to achieve security goals such as authentication, confidentiality and integrity, by using cryptography. For this reason they are also called cryptographic protocols.

Recently, many research efforts have been dedicated to the problem of analyzing the logical correctness of cryptographic protocols (e.g.[5][3][9][10][14]), whereas the implementation correctness problem has not yet been considered so much. One of the possible approaches to ensure implementation correctness is to produce implementations automatically from formal specifications [10][15][12]. If the code generator is such that the generated code faithfully implements the specification and avoids programming errors that can lead to security breaches, implementation correctness is achieved. Therefore, if the source specification is logically correct, so is the implementation. In this paper we show how this approach can be put into practice in a framework where the target code language is Java and cryptographic protocols are specified in spi calculus [2], a process algebraic specification language specifically tailored for such protocols.

The rest of the paper is organized as follows. Section 2 briefly introduces spi calculus, section 3 presents the architecture of Spi2Java, and sections 4-7 describe its components. Section 8 gives some experimental results, section 9 discusses related work, and section 10 concludes.

2 Spi calculus

The spi calculus is defined in [2] as an extension of the π calculus [11] with cryptographic primitives. It is a process algebraic language designed for describing and analyzing cryptographic protocols. The spi calculus has two basic language elements: terms, to represent data, and processes, to represent behaviors. In this paper we present only some features of spi calculus, through an example, due to the limited space. Fig. 1 shows the spi calculus[1] specification of the *Andrew*[8] key exchange protocol.

The specification is composed of two process descriptions named `pA` and `pB`, which represent the two roles of the protocol. The `Inst` process represents the interaction scenario where an instance of `pA` and an instance of `pB` run concurrently. The initiator role process `pA` and the `Inst` process are parameterized by `M`, which is the data that must be sent. `M` occurs explicitly as a parameter, because this is required by the security analysis tool [5]. In contrast, the other protocol parameters are all implicit.

The left column of Fig. 1 shows the exchanged messages using the informal, intuitive representation often encountered in the literature, where $\texttt{A} \longrightarrow \texttt{B} : \sigma$ means that `A` sends message σ to `B`. The central column shows the corresponding spi calculus specification for process `pA`, whereas the

---

[1] Spi2Java uses some typographic conventions respect to the original spi calculus

<table>
<tr><td>A->B: A,Na</td><td>pA(M):=
(@Na)
cAB<A,Na>.</td><td>pB():=
cAB(xA,xNa).</td></tr>
<tr><td>B->A: {(Na,k1AB)}kAB</td><td>cAB(xMSG).
(@KeyStore)
KeyStore<B>.
KeyStore(kAB).
case xMSG of
{xNa,xk1AB}kAB in
[xNa is Na]</td><td>(@KeyStore)
KeyStore<xA>.
KeyStore(kAB).
(@k1AB)
cAB<{xNa,k1AB}kAB>.</td></tr>
<tr><td>A->B: {Na}k1AB</td><td>cAB<{Na}xk1AB>.</td><td>cAB(xMSGcypher).
case xMSGcypher of
{xnewNa}k1AB in
[xnewNa is xNa]
KeyStore<xA,k1AB>.</td></tr>
<tr><td>B->A: Nb</td><td>cAB(dummy).
KeyStore<B,xk1AB>.</td><td>(@Nb)
cAB<Nb>.</td></tr>
<tr><td>A->B: {M}k1AB</td><td>cAB<{M}xk1AB>.0</td><td>cAB(Mcypher).
case Mcypher of
{x}k1AB in 0</td></tr>
<tr><td colspan="3" align="center">Inst(M):=(pA(M)|pB())</td></tr>
</table>

Figure 1. The Andrew Protocol spi calculus specification

right column shows the corresponding behavior of process pB.

The Andrew protocol assumes that each process has a local key store where symmetric keys are stored. Since the key store explicitly partakes in the protocol, it must be modelled in spi calculus. Our simple modelling strategy is to represent the key store as a separate process (not shown in Fig. 1) that interacts with the corresponding protocol principal through a dedicated communication channel (the *KeyStore channel*). The operations of getting and storing a key are modelled as inputs and outputs on the key store channel respectively. More precisely, a key is stored in the key store under an alias, which permits its unique identification. So, the operation of retrieving a stored key is represented by the statements $\mathtt{KeyStore} < \mathtt{xA} > .\mathtt{KeyStore}(\mathtt{kAB})$ where KeyStore denotes the interaction channel, xA is the alias and kAB is the variable where the key extracted from the key store is saved. The corresponding storing operation is described by the statement $\mathtt{KeyStore} < \mathtt{xA}, \mathtt{k1AB} >$ where k1AB is the key that must be stored under the alias xA. Note that the visibility of the KeyStore term is restricted with the @ operator, so it is considered private for the process. In a run of the Andrew protocol, five messages are exchanged between pA and pB over channel cAB: **1)**pA sends pB its identifier A and Nonce Na. pB receives the message and stores the two fields in variables xA and xNa respectively. **2)**pB retrieves key kAB, shared with pA, from its local KeyStore and builds a new fresh key k1AB, that together with xNa (Na) is encrypted with kAB and the result is sent to pA. pA receives the message and decrypts it by means of kAB retrieved from its local Key Store. The two fields of the computed cleartext (Na and k1AB) are stored in xNa and xK1AB and the match between the value of Na and xNa is checked. **3)**pA sends pB the nonce Na encrypted with the shared key k1AB. pB decrypts the message and checks the match between the just received nonce xnewNa and xNa. Then pB stores k1AB under the alias A in its local KeyStore, thus overwriting kAB. **4)**pB sends pA a fresh nonce Nb. pA receives the nonce and replaces kAB with k1AB in its local KeyStore. Now the key is fully agreed. **5)**pA use k1AB to encrypt the secret message M and sends it to pB. pB receives the encrypted message, decrypts it and stores M in variable x.

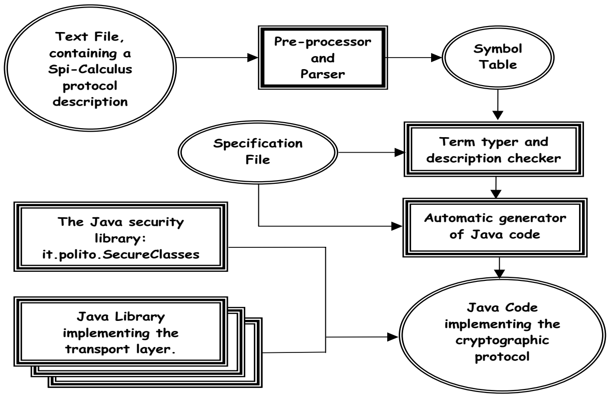

Figure 2. The Spi2Java Program Architecture

3 The Tool Architecture

The generated code is organized as one independent program for each protocol role, and such programs can be activated at need whenever a new session of the protocol must be executed. Therefore, for the purpose of code generation, Spi2Java operates for a single protocol role at time (like pA of Fig. 1), whereas processes that specify only particular instantiation scenarios of protocol sessions (like Inst of Fig. 1) are not relevant and are ignored during code generation.

The Spi2Java program is composed of two modules: a *Term Typer and Description Checker* and an *Automatic Generator of Java Code*. The generated code is based on Java library modules implementing, in a configurable way, the elementary operations that can occur in spi calculus descriptions. Fig. 2 shows the dataflow for the whole tool architecture.

4 The Term Typer and Description Checker

Spi calculus is not typed, so the *Term Typer and Description Checker* is responsible to fill the information gap between protocol specification and implementation for what concerns data types. In particular, this functional block automatically checks whether term variables are used consistently within a protocol role process and, if this check is positive, automatically assigns concrete Java types to term variables. Term type assignments are performed by an al-

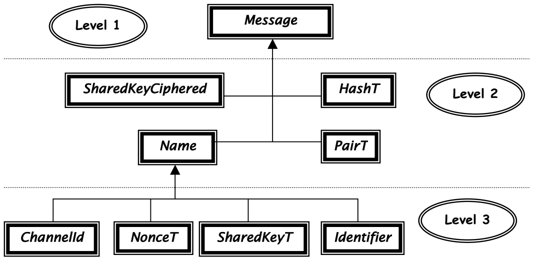

Figure 3. The term type class hierarchy

gorithm that associates any term variable with the most specialized Java class that safely represents it. The user can manually enforce more specialized types for certain variables by means of the *Specification* file, which is read and interpreted by this module. Manually specifying a more specialized type is possible but not necessary. For example, if a process does not perform any operation on a term except sending it out, possibly encapsulated within a structured term, the Message type is perfectly appropriate for that term.

We use a class hierarchy organized on three specialization levels (Fig. 3 shows only some of the classes, due to the limited space, although Spi2Java deals with all the spi calculus features) for those terms that could be sent over channels and another simple hierarchy for communication channels, since channel classes depend on the applied Transport Library and on the process role (client/server). So a term is typed as **Channel** when it is a generic communication channel used to send/receive messages or as **KeyStore** which is a possible specialization of Channel, when it represents the access point to a local key store where keys and/or digital certificates are stored. Here is a brief description of the meaning of term classes shown in Fig. 3. **Message** is the less specialized type, because it represents any message. A term is typed as Message whenever the algorithm is not able to determine a more specialized type for it. The internal Java representation of the Message type is simply a byte array. **Name** is a partially specialized type that represents any non-structured spi calculus term (i.e. a spi calculus name). It is important to note that Name is implemented by a class that cannot be instantiated, because objects of this class are always objects of more specialized concrete classes. A Name can be any level 3 class object but it can even be an object of a new user-defined derived class. **ChannelId** represents a channel identifier. It is useful for sending over an existing channel the necessary information for opening a new communication channel with a server role. **SharedKeyT** represents a key for use with symmetric cryptosystems. **NonceT** represents a randomly chosen sequence of bits. **Identifier** represents some information which identifies an entity in a unique way. For example, it can be used as an alias to identify a key or a certificate stored inside a KeyStore. **HashT** represents the result of applying a cryptographic hash function on some data (also known as message digest). **SharedKeyCiphered** represents the result of a symmetric cryptographic operation on some data. The operation can be either an encryption or a decryption. **PairT** represents a container of a couple of objects that can be of heterogeneous types. A tuple of objects is translated, inside the program, into nested Pair objects.

5 The SecureClasses Library

The *SecureClasses* security library provides a set of classes that implement in a flexible and configurable way all the elementary data types and cryptographic operations that can be abstractly expressed in spi calculus. This library acts as a general interface toward security providers, which are responsible to provide the concrete implementations of cryptographic algorithms. The providers used to test the *SecureClasses* Library and the generated code are those by: *SUN*[2] and *IAIK*[3]. This library heavily relies on Java Serialization to build data packets to be sent on communication channels and/or to be encrypted, principally because with the term type hierarchy, the same message can be automatically interpreted at different specification levels, at need. The *SecureClasses* library has been designed with special care, pursuing several goals: **1)** There is a strict correspondence whereby each spi calculus term corresponds to a Java class in the *SecureClasses* library as shown in Fig. 3. Note that each spi calculus statement corresponds to a simple Java construct with a method call in one of the term objects. **2)**Classes and methods hide the internal complexity of the cryptographic algorithms behavior and management. **3)**The user is able (by means of a special class where constants can be modified) to customize the internal behavior of classes, choosing the security provider, the algorithm and the related parameters for each different kind of cryptographic operation. **4)**The efficiency of the generated code, thanks to the efficiency of the classes implementation. **5)** Each class implementation has been kept as close as possible to its abstract model (a complete adherence is not achievable because the used cryptography is not perfect), and programming errors that can lead to known security breaches have been avoided.

6 The Transport Layer Interface

The *SecureClasses* library includes three interfaces named *ChannelId* to represent channel identifier, *ChannelT* to represent a generic client/server communication channel,

---

[2] The SUN-JCE provider is furnished as extension with the JCE 1.2.x or included inside the JDK 1.4, it is available at http://java.sun.com

[3] The IAIK-JCE provider is a product of IAIK, it is available at http://www.iaik.tugraz.at

and *ServerT* to represent the generic server process waiting for incoming client requests. All these classes are the interaction point with the Transport Layer Library that is used. In this way, transport layer independence is achieved for the generated code. The user can specify it by means of the *Specification* file.

The transport layer classes hide transport layer management and enable the direct translation of any spi calculus input/output operation into proper Java code.

7 The Java Code Automatic Generator

The Java automatic generator provides the Java implementation of the protocol role described in spi calculus and is partially guided by the *Specification* file, where the user can specify several implementation choices, such as for example to assign the role of client or server to a spi calculus process, by assigning a listening channel to a server role, to describe what terms are return parameters and which transport layer library must be used. The generated code uses classes and methods provided by the *it.polito.SecureClasses* and by the Transport Library module that has been chosen.

Starting from a spi calculus specification, and the related *Specification* file, the Code Generator writes the Java protocol implementation class on the *Protocol* file. The Code Generator also produces an application skeleton (on the *Application* file) and some other class files useful to launch the client application/server, since the user will typically use the protocol handshake as a prelude of a target application.

The Protocol file is generated by syntax directed translation of the spi calculus behavior expression. More precisely the spi calculus syntax tree is visited and for each spi calculus operation, the Java code that implements it is generated, preceded by a description comment. The latter enhances code readability and makes the correspondence with the spi calculus specification visible. Fig. 4 shows the most interesting piece of code generated in the Protocol file for the `pA` process of the Andrew protocol. Return objects are retrievable by the public class method `getReturnParameter(inti)` that is not shown here. `ClassCastExceptions` are generated when a wrong cast happens and this may happen during message receiving and deserialization operations.

```java
public class andrewPA_Protocol {

/* Object containing Return Parameters */
private Message retPar;

/* The number of Return Parameters */
private int nPar;

public andrewPA_Protocol ( Message M_1, IdentifierT A_0, IdentifierT B_0,
        TcpIpClientChannel cAB_0, LoadKeyStore KeyStore_5)throws ProtocolException {
   try {

     /* cAB_0<(A_0,Na_2)> */
     NonceT Na_2 = new NonceT();
     PairT Pair__A_0__Na_2 = new PairT(A_0, Na_2);
     cAB_0.Send( Pair__A_0__Na_2 );

     /* cAB_0(xMSG_4) */
     SharedKeyCiphered xMSG_4 = (SharedKeyCiphered) cAB_0.Receive();

     /* KeyStore_5<B_0>
        KeyStore_5(kAB_7) */
     PasswordManager pm0 = new ConstantPassword();
     SharedKeyT kAB_7 = new SharedKeyT(B_0.getIdentifier(), KeyStore_5.getKeyStore(), pm0);

     /* case xMSG_4 of {_w0_8}kAB_7 in */
     SharedKeyCiphered _w0_8 = new SharedKeyCiphered(xMSG_4.getEncoded(), kAB_7,
                                 Cipher.DECRYPT_MODE, xMSG_4.getIV() );

     /* let (xNa_9,xk1AB_9) = _w0_8 in */
     PairT Pair__xNa_9__xk1AB_9 = (PairT) DeserializeT.getDeserializeT( _w0_8.getEncoded() );
     NonceT xNa_9 = ( NonceT ) Pair__xNa_9__xk1AB_9.getFirst();
     SharedKeyT xk1AB_9 = ( SharedKeyT ) Pair__xNa_9__xk1AB_9.getSecond();

     /* [xNa_9 is Na_2] */
     if( !xNa_9.isEqual( Na_2 ) )
       throw new ProtocolException("Match test is false!");

     /* cAB_0<{Na_2}xk1AB_9> */
     SharedKeyCiphered Na_2_SharedKeyCiphered_xk1AB_9 = new SharedKeyCiphered(
                 SerializeT.getSerializeT( Na_2 ), xk1AB_9, Cipher.ENCRYPT_MODE, null );
     cAB_0.Send( Na_2_SharedKeyCiphered_xk1AB_9 );

     /* cAB_0(dummy_12) */
     Message dummy_12 = ( Message ) cAB_0.Receive();

     /* KeyStore_5<(B_0,xk1AB_9)> */
     PasswordManager pm1 = new ConstantPassword();
     xk1AB_9.addToKeyStore( B_0.getIdentifier(), KeyStore_5.getKeyStore(), true , pm1 );

     /* cAB_0<{M_1}xk1AB_9> */
     SharedKeyCiphered M_1_SharedKeyCiphered_xk1AB_9 = new SharedKeyCiphered(
                 SerializeT.getSerializeT( M_1 ), xk1AB_9, Cipher.ENCRYPT_MODE, null );
     cAB_0.Send( M_1_SharedKeyCiphered_xk1AB_9 );

     /* Build the container, for the objects we have to return. */
     nPar = 1;
     retPar = ( SharedKeyT )xk1AB_9;

   } catch( java.lang.ClassCastException cce ) {
     throw new ProtocolException("An unexpected object has been received belonging to
                                   class: " + cce.getMessage() );
   }

 }
```

Figure 4. The Andrew pA Protocol code

7.1 Generated Java Code Characteristics

Spi2Java is coupled with a protocol analyzer [5] that detects design protocol flaws. Therefore, if we have described a protocol that is considered secure, we want to obtain the more faithful adherence between the specification and the implementation to maintain the design security. Notice that spi calculus is able to describe not only the message exchange but also what are the checks that must be performed in the implementation, thus the implementation must not implement all possible checks but only the specified ones. In this way a direct translation from spi calculus into Java code is possible without any implicit behavioral assumptions. So we can provide an adherent and faithful code implementation of the described protocol, because we provide a strict correspondence between the spi calculus description elements and Java code fragments. The correspondence is achieved since the typing of terms allows to establish a mapping from behavior expressions to behavior logics which use classes of the *SecureClasses* library and from terms to classes of the *SecureClasses* library. In this way we can grant the protocol implementation design correctness. Nevertheless the protocol security analyzer [5] assumes that all the cryptographic operations are implemented with a perfect encryption (i.e. this means that hash of different message never collide), that unluckily doesn't exist in the reality. So we can't prove in a formal way that the protocol design security is assured, but we can limit our considerations to this code property only as intuitive. However notice that also if it is impossible to achieve the perfect encryption, it is possible to draw near. For this purpose we give the possibility to change both security providers and algorithms for any kind of cryptographic operation, allowing the user to find the implementation that best matches the perfect encryption assumption. This capability also gives the chance to easily substitute an algorithm implementation that is affected by an error, immediately as soon as it is discovered. About the programming code security we have to consider ours libraries (*SecureClasses* and *TcpIpLayer*) and we can only affirm that we have carefully developed playing attention on Java Security Guidelines [1] and we have well tested all classes of our libraries, so we hope that are immune of implementation weakness.

Furthermore, the implementation of our libraries, and then of the protocol, is considered secure against some of the most common implementation weakness like: **Buffer overruns**[4], because the adopted implementation language is Java, which cannot be affected by this kind of attacks (except for overflows in the JVM itself). In fact Java uses the following to safeguard the memory: array bounds are checked for each array access; there aren't pointers, memory is managed by reference (pointers are one of the most bug-prone aspects of C and C++); object casting is restricted (necessary to ensure type safety); variables cannot be used before they are initialized (another memory-protection mechanism); garbage collection automatically frees memory (avoiding memory deallocation errors). **Type flaws** that occur when a message is interpreted in an incorrect form, because all messages are typed and code always checks type inconsistencies and raises an exception when a mismatch occurs. Moreover notice that in our implementation all message are serialized, so the deserialization mechanism fails and raises an exception if a type flaw occurs. **False input attacks** because they rely on unchecked input parameters, whereas checks on objects are already specified in the spi calculus description, and their specification correctness is verified by the security analyzer program [5]. Moreover the implementation of our classes provides all the necessary checks and generates an exception whenever a constraint is violated.

The SecureClasses library allows to hide the complexity of cryptographic algorithms, and allows the user to independently choose the Security Provider, the related algorithms and their parameters. The complexity of the cryptographic algorithms is hidden by the use of classes belonging to the type hierarchy and by a class containing constants which permit to customize the library.

The code is easily readable, secure and optimized with the meaning explained below. Code implementation security is related to the intrinsic security of the protocol specification and also to the absence of errors and security flaws in ours libraries and in the Security Providers adopted. The protocol code implementation is optimized in the sense that each object is created only when it is really needed: this means that at each time all live objects are only those strictly needed.

8 Testing and experiments

We have tested the *it.polito.SecureClasses* library using all the features supported by the *IAIK*[5] and *SUN*[6] providers. Moreover we have tested *Spi2Java* using several simple ad-hoc protocol examples and some real known protocols: *Andrew*[8], *KSL*[8], *SSL*[6], *Needham-Schroeder*[4].

9 Related work

In the last years some tools have been developed to specify, design, verify and implement cryptographic protocols. While a lot of papers address protocol verification, only three of them address automatic code generation [10] [15] [12].

We have chosen Java as the target language for protocol implementation, as in [10] [15] [12], due to the language excellent security architecture and resistance to common security attacks, as is possible to understand reading [18] [7].

The choice of spi calculus as the language for protocol specification gives some advantages with respect to previous works, because it allows to explicitly specify which checks the protocol must perform. This implies that the code generator, knowing what kind of controls must be implemented, can avoid to generate controls that are not required, thus producing an optimized protocol code. All the other tools [10] [15] [12], starting from protocols specified by means of formal languages without the above feature, must always implement all the possible checks. Moreover, all the other tools [10] [15] [12] require that each term type is explicitly specified, while our tool is able to understand the correct type of terms in an automatic way, directly in almost all cases or after another term has been typed by the user.

Cryptographic Code Generation From CAPSL [12] starts from the *CAPSL* or *CIL* specification languages. The produced code includes a demonstration environment, useful to view the protocol behavior, that shall be removed or modified for a direct use in application environments. This environment represents the ”man in the middle” attack, so it receives all messages exchanged between parties showing protocol handshakes. Our code does not contain a demonstration environment, but we can add such a feature in the

---

[4] as is possible understand reading [18][7]

[5] The IAIK-JCE provider is a product of IAIK, it is available at http://www.iaik.tugraz.at

[6] The *SUN-JCE* provider is furnished as extension with the *JCE* 1.2.x or included inside the *JDK* 1.4, it is available at http://java.sun.com

transport layer directly (building a new transport library), thus allowing the redirection of messages towards a demonstration application able to behave as an attacker.

Moreover in [12] the generated code is inefficient because it runs by interpreting an abstract data structure. A further limitation is the dependence of key objects on encryption and decryption algorithms, which are fixed as DES for symmetric encryption. Another limitation is the absence of public encryption which is substituted in the code by a dummy encryption operation.

The *AGVI* [15] tool generates code using the same protocol description taken by the protocol analyzer *Athena* [14]. [15] contains few information about code generation and implementation. Such information is probably reported in [13], which, however is not reachable on the web.

SPEAR II [10] provides code generation from an abstract protocol specification in the *GYPSIE* [16] environment, while parameters and settings for code generation are specified in the graphical *GENIE* [17] environment. The produced code is based on *Cryptix*[7] and *Crypto-J*[8] cryptographic libraries. A good feature of [10] is that it uses the accepted standard *ASN.1* for describing messages, thus allowing the generated code to communicate with other non-SPEAR II implementations.

All the above projects [10] [15] [12] generate a code that is not *Java-Security-Provider*-independent as ours. Provider independence is a good feature, because if a security flaw is found in a specific library, it is possible to replace the security provider with another one, unaffected by the problem, without modifying the generated code. Only the code produced by *SPEAR II* [10] is Transport Layer independent and translates from protocol specification to code implementation directly, as we do.

10 Conclusions

A new automatic Java code generator for cryptographic protocols specified in spi calculus has been developed, to be integrated in a specification and verification environment for security protocols. Spi2Java provides the protocol implementation together with a skeleton code, useful to develop an application that uses the protocol.

Spi2Java has a module that associates a type to each spi calculus term in an automatic or semi-automatic way and checks for abstract description incongruities.

With the *SecureClasses* library, we have been able to hide the complexity of the cryptographic algorithms and offer maximum flexibility, allowing the choice of a Security Provider, an algorithm and the algorithm parameters for each kind of cryptographic operation. Moreover, a strict correspondence between spi calculus objects and classes allows us to guarantee a high confidence level about code correctness. The definition of Transport Layers as modules allows the user to choose and replace the transport protocol in an easy way.

The produced Java code optimizes the creation time of needed object, avoids common implementation attacks and maintains an high understandability thanks to the presence of comments before each behavior expression.

References

[1] Security code guidelines. SUN Microsystems. Online, available at: http://java.sun.com/security/seccodeguide.html.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols the spi calculus. *Inf. Comput.*, 1(148):1–70, 1999.

[3] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Trans. on Softw. Eng. and Meth.*, 9(4):443–487, 2000.

[4] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Comm. of the ACM*, 24(8):533–536, 1981.

[5] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Trans. on Softw. Eng. and Meth.*, 12(2):222–284, Apr. 2003.

[6] A. O. Freier, P. Carlton, and P. Kocher. The ssl protocol version 3.0, 1996. Online, available at: http://home.netscape.com/eng/ssl3/draft302.txt.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. SUN Microsystems, 2nd edition. Online, available at: http://java.sun.com/docs/books/vmspec/.

[8] G. Lowe. Some new attacks upon security protocols. *In 9th IEEE Comp. Sec. Found. Work.*, pages 162–169, 1996.

[9] G. Lowe. Casper: A compiler for the analysis of security protocols. *J. Comput. Secur.*, 6(1):53–84, 1998.

[10] S. Lukell and C. Veldman. Automated attack analisys and code generation in a unified, multi-dimensional security protocol engineering framework. *Comp. Science Hon.*, 2002.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, pages 1–77, 1992.

[12] F. Muller and J. Millen. Cryptographic protocol generation from capsl. Tech. Rep. SRI-CSL-01-07, SRI Int., 2001.

[13] A. Perrig, D. Phan, and D. Song. Acg - automatic code generation and automatic implementation of a security protocol. Tech. Rep. 00-1120, Univ. of California, 2000.

[14] A. Perrig, D. Song, and S. Berezin. Athena: a novel approach to efficient automatic security protocol analisys. Tech. rep., Univ. of California and Carnegie Mellon Univ.

[15] D. Phan, A. Perrig, and D. Song. Agvi - automatic generation, verification, and implementation of security protocols. Tech. rep., Univ. of California.

[16] E. Saul. Facilitating the modelling and automated analysis of cryptographic protocols. Master's thesis, Univ. of Cape Town, 2001.

[17] C. Veldman, S. Lukell, and A. Hutchison. Attack modelling, code generation and performance analysis in a multi-dimensional security protocol engineering framework. Project report, Univ. of Cape Town, 2002.

[18] F. Yellin. Low level security in java. Online, available at: http://java.sun.com/sfaq/verifier.html.

---

[7] The Cryptix library is available from http://www.cryptix.org

[8] The Crypto-J library is an RSA product, it can be obtained form http://www.rsa.com