Provably Correct Java Implementations of Spi Calculus Security Protocols Specifications

Alfredo Pironti[a,*], Riccardo Sisto[a]

[a] *Politecnico di Torino*
Dip. di Automatica e Informatica
c.so Duca degli Abruzzi 24, I-10129 Torino (Italy)

Abstract

Spi Calculus is an untyped high level modeling language for security protocols, used for formal protocols specification and verification. In this paper, a type system for the Spi Calculus and a translation function are formally defined, in order to formalize the refinement of a Spi Calculus specification into a Java implementation. The Java implementation generated by the translation function uses a custom Java library. Formal conditions on such library are stated, so that, if the library implementation code satisfies such conditions, then the generated Java implementation correctly simulates the Spi Calculus specification. A verified implementation of part of the custom library is further presented.

Key words: Model-based software development, Correctness preserving code generation, Code verification, Formal methods, Security protocols

1. Introduction

Security protocols are communication protocols that use cryptographic primitives in order to protect some assets. The Spi Calculus [1] is a formal domain-specific language that can be used to abstractly specify security protocols. A Spi Calculus specification can then be passed to an automatic tool, usually a model checker (e.g. [2]) or a theorem prover (e.g. [3]), that verifies some security properties on it [4, 5]. By verifying these properties, the designer can ascertain that the protocol actually works as intended and that it is resilient to attacks performed by a (Dolev-Yao [6]) attacker.

However, a real implementation of a security protocol implemented in a programming language may significantly differ from the corresponding verified formal specification expressed in Spi Calculus. It follows that the real behavior of the protocol differs from the verified one, possibly enabling attacks that are not present in the formal specification. For instance, a protocol implementation may miss to perform a check on a received nonce that is prescribed by the protocol specification, and this missed check may enable a replay attack on the implementation.

Moreover, security protocols are usually applied in safety or mission critical environments. For these reasons, assessing the correctness of a protocol implementation with respect to its formally verified specification is a great challenge for the software engineering community. By testing the protocol implementation, only few scenarios are taken into account. This approach may not give enough confidence about implementation correctness, due to two facts: (1) the distributed and concurrent nature of security protocols, that generates a large (usually unbounded) number of possible application scenarios; (2) the presence of an active attacker, who can behave in the worst way and is not under the software developer control.

In principle, in order to ensure that the formal model is correctly refined by the implementation, two development methodologies can be used, namely model extraction [7, 8, 9, 10] and code generation [11, 12, 13]. By using the first methodology, one starts by manually developing a full blown implementation of a security protocol, and automatically extracts a formal model from it. The extracted formal model is then verified in order to check the desired security

*Corresponding author
Email addresses: alfredo.pironti@polito.it (Alfredo Pironti), riccardo.sisto@polito.it (Riccardo Sisto)

properties. By using the second methodology, one starts from a formal specification of a security protocol, and automatically generates the code that implements it; the developer must still provide the implementation details that are not caught by the formal model. In order for each methodology to be useful it must be possible to formally show that a refinement relation between the implementation code and the abstract formal model exists and that this relation preserves security properties.

The work presented in this paper aims to improve the correctness assurance that can be achieved by using an automatic code generation approach. A translation function from the Spi Calculus specification language to the Java implementation language had been informally described in [11, 12]. This translation function is now formally defined here. The translation function relies on a Java library that essentially wraps the calls to the Java Cryptography Architecture library. The latter implements cryptographic primitives in the Java environment. In this paper it is formally shown that, if it is assumed that the implementation of the library satisfies some conditions that are formally expressed, then the generated Java code correctly refines the abstract model. Moreover, a verified implementation of part of the library is presented in this paper. A preliminary version of this work appeared in [14].

The remainder of this paper is organized as follows. Section 2 describes some works related to the one presented here. Section 3 briefly describes the Spi Calculus and presents a type inference system for the Spi Calculus and a formal translation from the Spi Calculus to Java. Section 4 presents the main correctness property of the translation: the generated Java implementation simulates the Spi Calculus specification it has been generated from. Moreover, a possible verified implementation of part of the custom library used by the generated code is shown. Finally, section 5 concludes and gives some hints for future work.

2. Related Work

To our knowledge, there are two independent works dealing with generation of Java code starting from Spi Calculus specifications, namely [11, 12] and [13]. However, none of these works provides rigorous formal proofs for the generated code.

The AGC-C# tool [15] automatically generates C# code from a verified Casper script. Since in that work a formal translation function is not provided, it is not possible to obtain soundness proofs. Moreover, the generated code is obtained from a manually modified version of the verified Casper script. These manual changes may introduce errors, and the generated code starts from a model that is not the verified one, thus possibly invalidating a soundness assertion.

In [16], a manual refinement of CSP protocol specifications into JML constraints is described. However, no formal translation rules from CSP processes to JML constraints are provided, and, like in the previous case, the lack of automatic tools makes the manual refinement process error prone.

In web services, security properties are expressed at a higher level, as policy assertions [17, 18]. Rather than specifying *how* security is achieved, through the coordination of cryptographic primitives inside the protocol specification, a policy assertion specifies a property that must hold for a specific set of SOAP messages.

The tool described in [19] checks user given policy assertions in order to find common flaws. However, it does not provide any formal proof about the correctness of the user-provided policy assertions against a specification. Moreover, there is still the need to verify that the policy assertion implementations are correct. The work presented in [10] gives a verified reference implementation of the WS-Security [20] protocol, written in F#. This implementation can be a starting point in future complete protocol implementations. However, to our knowledge, no tool that verifies a user-provided implementation of policies, nor that generates a correct implementation from user-provided policies, is currently available for web services policy assertions.

In the model extraction approach, the work in [7] provides an abstraction from a subset of F# code to applied π-calculus with a formal correctness proof. In [7], like in this paper, correctness of some low level cryptographic libraries is assumed, that is, the concrete low level libraries are assumed to behave like the abstract symbolic counterparts. Although promising, this approach currently uses a starting language that is not very common in the programming practice, and the constraints on the selected subset of F# currently allow only the verification of *ad hoc* written code.

$L, M, N ::=$	terms
n	name
(M, N)	pair
0	zero
$suc(M)$	successor
x	variable
$\{M\}_N$	shared-key encryption
$H(M)$	hashing
M^+	public part
M^-	private part
$\{[M]\}_N$	public-key encryption
$[\{M\}]_N$	private-key signature

Table 1: Spi Calculus terms.

$P, Q, R ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$P\|Q$	composition
$!P$	replication
$(\nu n)P$	restriction
$[M \; is \; N]P$	match
$\mathbf{0}$	nil
$let \; (x, y) = M \; in \; P$	pair splitting
$case \; M \; of \; 0 : P \; suc(x) : Q$	integer case
$case \; L \; of \; \{x\}_N \; in \; P$	shared-key decryption
$case \; L \; of \; \{[x]\}_N \; in \; P$	decryption
$case \; L \; of \; [\{x\}]_N \; in \; P$	signature check

Table 2: Spi Calculus processes

3. Formalizing the Translation

3.1. The Spi Calculus

The Spi Calculus extends the π-calculus [21] by adding a fixed set of cryptographic primitives to the language, namely symmetric and asymmetric encryptions, and hash functions, thus enabling the description of the main security protocols. Adding more custom cryptographic primitives to the language, so that more security protocols can be described, is rather straightforward.

A Spi Calculus specification is a system of concurrent processes that operates on untyped data, called terms. Terms can be exchanged between processes by means of input/output operations. Table 1 contains the terms defined by the Spi Calculus, while table 2 shows the processes.

A name n is an atomic value, and a pair (M, N) is a compound term, composed of the terms M and N. The 0 and $suc(M)$ terms represent the value of zero and the logical successor of some term M, respectively. A variable x represents any term, and it can be bound once to another term. The term $\{M\}_N$ represents the encryption of the plaintext M with the symmetric key N, while $H(M)$ represents the result of hashing M. The M^+ and M^- terms represent the public and private part of the keypair M respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively.

Informally, the $\overline{M}\langle N\rangle.P$ process sends message N on channel M, and then behaves like P. Conversly, the $M(x).P$ process receives a message from channel M, and then behaves like P, with variable x bound to the received term in P. The general forms $\overline{M}\langle N\rangle.P$ and $M(x).P$ allow for the channel to be an arbitrary term M. However, the only useful cases are for M to be a name, or a variable that gets instantiated to a name. A process P can perform an input or output operation if and only if there is a reacting process Q that is ready to perform the dual output or input operation. Note,

however, that processes run within an environment (the Dolev-Yao attacker) that is always ready to perform input or output operations. Composition $P|Q$ means parallel execution of processes P and Q, while replication $!P$ means an unbounded number of instances of P run in parallel. The restriction process $(\nu n)P$ indicates that n is a fresh name in P, that is a name never previously used and unknown to the attacker. The match process $[M\ is\ N]\ P$ executes like P, if M equals N, otherwise it is stuck. The nil process does nothing. The pair splitting process $let\ (x, y) = M\ in\ P$ binds the variables x and y to the two components of the pair M. Otherwise, if M is not a pair, the process is stuck. The integer case process $case\ M\ of\ 0 : P\ suc(x) : Q$ executes like P if M is 0. If instead M is $suc(N)$, this process executes like Q and x gets bound to N. If M is neither 0 nor $suc(N)$, the process is stuck. If L is $\{M\}_N$, that is a term encrypted with key N, then the shared-key decryption process $case\ L\ of\ \{x\}_N\ in\ P$ executes like P, with x bound to M. If instead L is not a term encrypted with key N, the process is stuck. The behavior of the decryption and signature check processes is analogous. The free names and free variables of a process P are denoted by the disjoint sets $fn(P)$ and $fv(P)$ respectively, and $fnv(P) = fn(P) \cup fv(P)$. A process is *closed* if it has no free variables.

For brevity, the formal work presented in this paper is shown in detail only on a subset of the Spi Calculus, including the most significant constructs. More specifically, the integer case, asymmetric cryptography, hashing and related terms are not shown. The complete work considering the full Spi Calculus can be found in [22].

When defining the translation function, only the sequential part of the Spi Calculus is considered: composition, replication and recursive processes are not handled. This is not an important limitation, and it does not prevent the results obtained in this paper to apply to most security protocols. Indeed, each session of a security protocol is very often composed of two or more sequential agents acting concurrently in a distributed environment. Multiple protocol sessions can run concurrently. Accordingly, a security protocol is generally described in Spi calculus by an expression of the form $!P_1|!P_2|\cdots|!P_n$, where each protocol agent P_i is a sequential process. Replication is used to express the possibility of spawning multiple instances of each agent, acting in different protocol sessions, and composition is used to express that the different agents run concurrently. The composition and replication processes are rarely used within each agent's specification.

During formal verification, the whole protocol specification (including replication of agents and their composition) is considered, so that all the possible scenarios are verified. When deriving the implementation, however, each sequential agent is translated into Java separately. Replication and composition are implicitly implemented by running several instances of the sequential actors in different Java threads, or in different machines.

Allowing replication and composition inside each actor would be practically not so useful. At the same time, it would make the formalism and the proofs presented here much more complex. This would happen because, as it will be clear in the rest of the paper, more semantic rules would have to be considered for both Spi Calculus and Java, and each proof would have to be extended to consider the possbile interleavings of concurrent Java threads. Recursion could be useful to describe protocol runs of unbounded length, but this feature is normally not allowed by verification tools.

The semantics of the Spi Calculus has been originally expressed by means of a reaction relation $P \rightarrow P'$, a reduction relation $P > P'$ and structural equivalence $P \equiv P'$ [1]. $P \rightarrow P'$ means that P can evolve into P' after a message exchange between two parallel components of P, $P > P'$ means that P can evolve into P' by performing some other (internal) operation, while $P \equiv P'$ allows processes to be rearranged so that the previous rules can be applied. As the focus of this paper is on sequential processes interacting with the environment, an equivalent semantics based on a classical labeled transition system (LTS) is introduced. To simplify formalism, it is assumed that for any process P the set of free names and the set of bound names in P are disjoint; it is still possible to deal with any Spi Calculus process, by first applying α-renaming to its bound names. For any sequential process P, a τ transition $P \xrightarrow{\tau} P'$ means that P can evolve into P' without interaction with its environment. Formally, it means $P > P'$ or $P \equiv P'$. It is worth noting that the evolution $P \rightarrow P'$ is not possible if P is a sequential process. Instead, $P \xrightarrow{m!N} P'$ and $P \xrightarrow{m?N} P'$ mean that P can interact with its environment by respectively sending or receiving N on channel m. Formally,

$$P \xrightarrow{m!N} P' \quad \text{means} \quad \exists \overline{y} | \forall Q.\ (P|m(x).Q) \rightarrow (\nu \overline{y})(P'|Q[N/x])$$
$$P \xrightarrow{m?N} P' \quad \text{means} \quad \exists \overline{y} | \forall Q.\ (P|(\nu \overline{y})\overline{m}\langle N \rangle .Q) \rightarrow (\nu \overline{y})(P'|Q)$$

where $\overline{y}$ is a possibly empty list of names.

According to these definitions, it can be shown that the semantics of Spi Calculus sequential processes can be

Figure 1: Type hierarchy for the selected Spi Calculus subset.

expressed by the rules

$$\begin{array}{rcl} \overline{m}\langle N\rangle.P & \stackrel{m!N}{\rightarrow} & P \\ m(x).P & \stackrel{m?N}{\rightarrow} & P[^N/_x] \\ [M\ is\ M]\,P & \stackrel{\tau}{\rightarrow} & P \\ let\ (x,y) = (M,N)\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x][^N/_y] \\ case\{M\}_N\ of\ \{x\}_N\ in\ P & \stackrel{\tau}{\rightarrow} & P[^M/_x] \end{array}$$

$$\frac{P \stackrel{\mathcal{L}}{\rightarrow} P'}{(\nu n)P \stackrel{\mathcal{L}}{\rightarrow} (\nu n)P'}, n \notin fn(\mathcal{L})$$

where $\mathcal{L}$ ranges over labels and $fn(\mathcal{L})$ is the set of names included in label $\mathcal{L}$.

3.2. The Type System

Spi Calculus terms are untyped. During formal verification this feature is essential to find type flaw attacks, that are attacks based on type confusion. However Java is statically typed. In order to enable the former language to be translated to the latter, it is necessary to assign a static type to every term used in a Spi Calculus specification. Types can be inferred automatically to some extent, so that the user work is minimized. However, unfortunately, it is not possible to reuse existing type systems for the Spi Calculus, because they have different purposes. For example, in [23] a generic type system is developed in order to describe some process behavior properties, such as deadlock-freedom or race-freedom. These properties are not so related to the common security properties, such as secrecy or authentication. It turns out that, for our purposes, the type system in [23] is even too much expressive about program behavior, but it does not assign static types to terms, thus being useless for our purposes.

The type system and associated type inference algorithm developed in this paper recall some standard type systems for the λ-calculus, such as the one in [24]. Essentially, the type system allows the type of a term to be inferred by looking at the context where that term is used. The type system relies on a set of known, hierarchically related (by the subtype relation <:) types, depicted in figure 1. *Message* is the top type, representing any message, so that every term has type *Message*. The types that directly descend from *Message* correspond to different forms of Spi Calculus terms, while the subtypes of *Name* correspond to different usages of names. The user is allowed to extend the given type hierarchy, by adding more specialized types.

In order to formalize the type system, a typing context Γ is defined as a set containing type assignments of the form $x : A$, where x ranges over names and variables and A over types. The judgment $\Gamma \vdash M : A$ means that M is well formed and must have type A in the typing context Γ (M can still have subtypes of A). The typing context Γ must contain type assignments for all the free names and free variables of M. The judgment $\Gamma \vdash P$ means that process P is well formed in the typing context Γ. It is worth noting that the proposed type system does not assign types to processes, but only to terms. Indeed, in order to enable translation into Java, it is sufficient to assign a static type to each term, because terms are translated into Java typed data. This is not needed for processes instead, which are translated into sequences of Java statements. For P to be well formed within Γ, it is necessary (but not sufficient) that all of its free names and variables appear in Γ. As it will be clear later on, given a generic Spi Calculus process P, it may not be possible to find Γ such that P is well formed. Since our translation function only translates well formed processes, it turns out that only the set *S pi* of well formed processes, which is a subset of all Spi Calculus processes, is translated by our function. It is worth noting that this constraint does not alter the Dolev-Yao attacker model. Indeed, during formal verification of a (well formed) process, the attacker is still modeled as the (possibly non well formed) environment.

The typing rules for the Spi Calculus subset considered in this work are reported in figure 2; this type system uses the standard subsumption rule (not shown). For brevity, only some significant rules are commented. The (T-Pair) rule states that if M_1 has type A_1 and M_2 has type A_2, then it is possible to state that the pair (M_1, M_2) has type $A_1 \times A_2$. It is worth noting that this formalization keeps, for each pair, information on the types of the contained items. A possible Java implementation of this feature can be obtained by using Java generic types, introduced in Java 5.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(T-NameVar)} \qquad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2}\ \text{(T-Pair)}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : ShKey}{\Gamma \vdash \{M\}_K : ShKeyC\langle A\rangle}\ \text{(T-ShKC)}$$

$$\frac{\Gamma \vdash M : Channel \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\langle N\rangle.P}\ \text{(P-Out)}$$

$$\frac{\Gamma \vdash M : Channel \qquad \Gamma, x : A \vdash P}{\Gamma \vdash M(x).P}\ \text{(P-In)}$$

$$\frac{\Gamma \vdash M : Message \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash [M\ is\ N]\ P}\ \text{(P-Match)}$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ \text{(P-Nil)} \qquad \frac{\Gamma, n : A \vdash P \qquad A <: Name\ \wedge\ A \not<: Channel}{\Gamma \vdash (\nu n)P}\ \text{(P-Restr)}$$

$$\frac{\Gamma \vdash M : ShKeyC\langle A\rangle \qquad \Gamma \vdash K : ShKey \qquad \Gamma, x : A \vdash P}{\Gamma \vdash case\ M\ of\ \{x\}_K\ in\ P}\ \text{(P-ShKD)}$$

$$\frac{\Gamma \vdash M : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash P}{\Gamma \vdash let\ (x, y) = M\ in\ P}\ \text{(P-PSplit)}$$

Figure 2: Typing rules for the selected Spi Calculus subset.

The (P-Match) rule states that if M and N are two well formed messages, and P is a well formed process, then the process that matches M and N, and then behaves like P, is well formed. It is worth noting that the (P-Match) rule does not require M and N to have the same type: we consider well formed a process where M and N are matched, even if they have different, incompatible types (e.g. M is typed as *Name* and N as $Name \times Name$). This is not an issue because when the desired security properties are checked against the untyped Spi Calculus, type flaw attacks are taken into account. This means that even an erroneous Java implementation that would consider equal two terms with incompatible types is considered in the formal verification step. Adding to (P-Match) the constraint stating that the type of M must equal the type of N would also be possible, though probably too rigid.

In the (P-ShKD) rule, note that the third premise requires the variable x to be added to the typing context, because x appears free in P, thus satisfying the necessary condition on Γ. The same reasoning holds for the first premise of (P-Restr). In the latter rule, however, it is also required that $A <: Name\ \wedge\ A \not<: Channel$. This is motivated because the restriction process generates a fresh *name*, and not a fresh term. The additional requirement avoids fresh channels to be created because they are essentially useless under our assumptions of sequential agent behavior. Indeed, a channel is useful for communication only if it is known by more than one sequential agent.

It is worth noting that using Java generic types can save run-time downcasts, when the type of a term is statically known; if the type is to be discovered at run-time, then a downcast will be necessary anyway, and it may fail.

It is worth noting also that this rather standard type system shares some common properties with other well known type systems [24]. In particular, the canonical form lemma can be proven, stating that if a term M has a particular type A, then it can only assume the particular values of A. Moreover, it can be shown that a type inference algorithm terminates finding the principal type for every term, if it is possible to find one.

Finally, the user can manually refine the type of a particular term. A user-provided type refinement for a given term c can be represented by adding a custom downcast rule, only valid for term c, to the type system. This is needed, because some types (e.g. the subtypes of *Channel*) cannot be inferred automatically only on the basis of their usage. For example, if the user wants to specify that c is a *Tcp/Ip Channel*, then the following rule is added.

$$\frac{\Gamma \vdash c : Channel \qquad Tcp/Ip\ Channel <: Channel}{\Gamma \vdash c : Tcp/Ip\ Channel}$$

It is worth noting that the properties of the type system are still preserved even when custom downcasts are added, thanks to the premises of the downcast rules. Indeed, the first premise ensures that the downcast is performed only if the type inference algorithm can infer that c must have type *Channel*; the second rule ensures that the downcast required by the user is coherent with the type hierarchy.

3.3. The Translation Function

The translation from Spi Calculus to Java is formalized by a set of functions, each dealing with a particular aspect of the translation. All of these functions operate on well formed Spi Calculus processes and terms, that is processes and terms for which a type derivation tree can be found.

Each sequential Spi Calculus process, typically representing one of the protocol roles, is translated into a sequence of Java statements implementing the Spi Calculus process. These statements are embedded into a `try` block, followed by a `catch` block, which are in turn embedded into a method that is invoked when a protocol run is requested. All the Java code surrounding the generated statements is called here the "context". The generated method will have one input parameter for each free name or variable of the Spi Calculus process. For example, the $\overline{c}\langle M\rangle.c(x).\mathbf{0}$ process has

a free name c and a free variable M: the generated Java method will have two input parameters, because it is assumed that the user will provide sensible values for these two terms.

Without proper encapsulation, translating a Spi Calculus process into Java would generate complex and non modular code. This complexity would make it difficult to show that the generated code correctly refines the Spi Calculus process. Indeed, all the implementation details that are abstracted away in Spi Calculus must be explicitly handled in Java. For example, every Spi Calculus encryption implies the creation and initialization of a Java `Cipher` object from the Java Cryptography Architecture (JCA). This object must be fed with data to be encrypted, and finally the resulting encryption can be obtained. Analogously, sending data over a channel requires direct handling of sockets or of other data structures, depending on the underlying medium for that channel.

In this paper a Java library called *SpiWrapper* is formally defined. It encapsulates some implementation complexity, allowing the generated code to be modular and easy to be mapped back to the original Spi Calculus specification. By defining a formal semantics for the intended behavior of this library, formal verification of the generated code is modularized too. One goal is to check that the generated code correctly refines the Spi Calculus specification, by assuming that the used SpiWrapper library behaves as specified. Another independent goal is to provide a formally verified implementation of such a library.

The SpiWrapper library offers an abstract Java class for each type depicted in figure 1. Each abstract class implements the operations that can be performed on the corresponding type. For instance, the abstract class `Pair<A,B>` offers the methods `A getLeft();` and `B getRight();` that retrieve the first and second items of the pair.

Each SpiWrapper class is abstract because only the internal data representation is handled; the marshalling functions, which encode the internal representation into the external one and vice versa, are declared but not implemented. This enables the user to define her own marshalling layer by extending the abstract class and implementing the marshalling and unmarshalling functions. It could be argued that letting the user implement the marshalling functions could introduce security flaws that were not present in the abstract model. However, as stated in [25], if some static checks on the user-written code are performed, such possible flaws are avoided.

Let Spi be the aforecited set of well formed Spi Calculus processes, $SpiTerm$ the set of well formed terms, and $Java$ the set of strings representing sequences of Java statements. Then, the function $tr_p : Spi, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ generates the Java statements for the Spi Calculus process given as its first parameter. In Java, all variables must be declared and initialized before they can be used. The second parameter of tr_p, let us call it $built \in 2^{SpiTerm}$, traces the well formed terms that have already been declared and initialized in the Java code. Moreover, some value should be returned after a successful protocol run (for example a negotiated shared secret, or a secure session id). The third parameter of tr_p, let us call it $return \in 2^{SpiTerm}$, contains the well formed terms that must be returned if a protocol run ends successfully. In order to return the desired values to the user, a Java object declared as `Map<String,Message> _return` is maintained, that maps the Java name of every term to be returned onto its value (that is the Java object implementing it). The map is filled as long as the values to be returned become available.

Before showing the definition of tr_p, some auxiliary functions are introduced. $ub : SpiTerm, 2^{SpiTerm} \rightarrow 2^{SpiTerm}$ takes two parameters: a term M and the *built* set. The *ub* function updates the *built* set by adding M and its subterms to it. Formally, *ub* is defined as

$$ub(M, built) = built \cup subterms(M)$$

where $subterms(M)$ is the set containing M and all its subterms.

$ret : SpiTerm, 2^{SpiTerm} \rightarrow Java$ generates the Java code that fills the `_return` map. The first parameter is a term M that will be possibly returned; the second parameter is the *return* set. If M is in the *return* set, the *ret* function puts the reference to the Java object implementing M into the map. For every function that returns Java code, the following typographical conventions are used: the returned text is quoted by double quotes; inside the quoted text, *italic* is used for functions that return text, while `courier` is used for verbatim returned text. The *ret* function is formally defined as

$$ret(M, return) = \text{“”, if } M \notin return$$
$$\text{“}\texttt{\_return.put("}J(M)\texttt{",}J(M)\texttt{);”, otherwise}$$

where $J(M)$ is a bijection that gives the name of the Java variable for term M, by mangling it.

$tr_t : SpiTerm, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ takes a term M, the *built* set and the *return* set. It generates the Java code that declares and initializes (hence “builds”) the Java variable $J(M)$ for the given Spi Calculus term M. The tr_t function is formally defined in figure 3. All declared Java variables are actually also marked as `final`, which however

$tr_t(M, built, return)$ = "", if $M \in built$
$tr_t(n, built, return)$ =
  "$T(n)$ $J(n)$ `= new` $Ts(n)$`(`$Param(Ts(n))$`;`
  $ret(n, return)$"
$tr_t((M, N), built, return)$ =
  "$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
  $T((M, N))$ $J((M, N))$ `= new`
  $Ts((M, N))$`(`$J(M)$`,`$J(N)$`,`
  $Param(Ts((M, N)))$`;` $ret((M, N), return)$"
$tr_t(\{M\}_N, built, return)$ =
  "$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
  $T(\{M\}_N)$ $J(\{M\}_N)$ `= new` $Ts(\{M\}_N)$`(`$J(M)$`,`
  $J(N)$`,`$Param(Ts(\{M\}_N))$`;` $ret(\{M\}_N, return)$"

Figure 3: Definition of the tr_t function.

is omitted here for brevity. For every term M, if M is already in the *built* set, then no code is generated, because the Java variable has already been declared and initialized. The $T(M)$ function returns the inferred type for the term M, which corresponds to one of the SpiWrapper abstract classes. The $Ts(M)$ function returns instead the name of the concrete user-provided Java class implementing the marshalling functions and extending the class returned by $T(M)$. Finally the $Param(Ts(M))$ function returns some user-defined parameters needed to make the protocol interoperable; such parameters depend on the type of the term. For example, if the type of term M is $ShKey$ (a shared key), then the parameters will be the key length, the key type and the desired JCA provider. The reader should not be distracted by interoperability details, though; more details on interoperability can be found in [12].

In the name n case, the code emitted by the *ret* auxiliary function is appended to the generated code, so that, if n is to be returned, it is added to the `_return` map.

In the pair (M, N) case, first tr_t is invoked on M and N to ensure they are built. It is worth noting that, by invoking $tr_t(N, ub(M, built), return)$, N is built by taking into account that M and all its subterms have already been built, so they are not built twice. For example, if $M = (a, b)$ and $N = (b, c)$, then b is built when M is built, and it must not be built again when N is built. Once M and N are built, tr_t appends the code that actually builds the pair, and the (possibly empty) code needed to return the pair. It is worth noting that it is only needed to explicitly call *ret* on the pair, and not on its components, because the recursive invocations of tr_t on the components already ensure they are added (if needed) to the `_return` map as soon as they are built.

Finally, in the shared key ciphered $\{M\}_N$ case, first M and N are built, then the shared key ciphered object is instantiated and assigned to the variable named $J(\{M\}_N)$. Finally, the *ret* function is invoked.

It is worth noting that no case is available for variables. Indeed, variables cannot be "built", rather they are declared and assigned by the code that implements the Spi Calculus processes that bind variables.

Now that all the auxiliary functions have been defined, the formal definition of tr_p is given in figure 4. Like for tr_t, all declared variables are also marked as `final`, though not shown here. Moreover, the translated Spi Calculus process is also printed as a Java comment to improve readability of the generated code (not shown here).

When translating the output process, first N is built, then the `send` method is invoked on the channel referenced by $J(M)$. $J(N)$ is the first `send` argument and it will be sent over the channel. As M is enforced to be a free name by the type system, there is no need to build it.

For the input process the `T receive(Class<T>,...);` method is invoked. Its arguments allow this method to create an instance of $Ts(x)$, fill it with the received data, and return its reference that is assigned to the Java variable $J(x)$. Like in the output case, channel M must be a free name, so it is not built.

In the decryption process the decryption key is passed as argument. If `ShKC<T>` is the type of $J(L)$, the `decrypt` method returns a newly created object of type `T` containing the decrypted data.

In the restriction process the name n is built, then the rest of the process is translated. When n is built a constructor is called, which generates the Java implementation of a new fresh name of the expected type.

With the pair splitting process first M is built, then the x and y variables are declared and assigned, which corre-

$tr_p(\mathbf{0}, built, return)$ = ""
$tr_p(\overline{M}\langle N\rangle.P, built, return)$ =
"$tr_t(N, built, return)$
$J(M)$.send($J(N)$,$Param(Ts(N))$);
$tr_p(P, ub(N, built), return)$"
$tr_p(M(x).P, built, return)$ =
"$T(x)$ $J(x)$ = $J(M)$.receive($Ts(x)$.class,
$Param(Ts(x))$); $ret(x, return)$
$tr_p(P, ub(x, built), return)$"
$tr_p(case\ L\ of\ \{x\}_N\ in\ P, built, return)$ =
"$tr_t(L, built, return)$ $tr_t(N, ub(L, built), return)$
$T(x)$ $J(x)$ = $J(L)$.decrypt($J(N)$,
$Param(Ts(x))$); $ret(x, return)$ $tr_p(P,$
$ub(L, built) \cup ub(N, built) \cup ub(x, built), return)$
$tr_p((\nu n)P, built, return)$ =
"$tr_t(n, built, return)$ $tr_p(P, ub(n, built), return)$"
$tr_p(let\ (x, y) = M\ in\ P, built, return)$ =
"$tr_t(M, built, return)$
$T(x)$ $J(x)$ = $J(M)$.getLeft(); $ret(x, return)$
$T(y)$ $J(y)$ = $J(M)$.getRight(); $ret(y, return)$
$tr_p(P, ub(M, built) \cup ub(x, built) \cup ub(y, built),$
$return)$"
$tr_p([M\ is\ N]P, built, return)$ =
"$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
if (!$J(M)$.equals($J(N)$))
{ throw new MatchException(); }
$tr_p(P, ub(M, built) \cup ub(N, built), return)$"

Figure 4: Definition of the tr_p function.

sponds to the Spi Calculus binding of a variable. It is worth noting that when the following P process will then be translated, M, x and y will all have already been built. Being variables, x and y are not built by the `new` operator, rather they are assigned the value (Java reference) of another term.

In the match case M and N are first built. If they are not equal execution is stopped by throwing an exception, which is handled by the context; else the match is successful, and execution can continue with the translation of the P process, where both M and N are marked as built. The context handles any possibly thrown exception by setting the `_return` map to `null`, thus simulating a stuck Spi Calculus process (a successful run of a protocol that does not need to return any value, still returns an empty map, and not `null`).

It is worth noting that when a message is received from a channel or a plaintext is reconstructed from a ciphertext, a new SpiWrapper object holding the obtained data must be created, even if the corresponding Spi Calculus term is already instantiated in another Java object. For example, the Java code implementing the Spi Calculus process $\overline{c}\langle M\rangle.c(x).\mathbf{0}$, will store one object for the M term, and one different object for the received x term. It may happen, however, that x is assigned the same value of M (simulating that the Spi Calculus process receives exactly the M term back), although they are two different objects. For this reason, equality of objects cannot be checked by means of reference equality, but the `equals` method must check if the value of the two objects is the same. Using singleton instances to represent Spi Calculus terms, and thus letting the match case check for reference equivalence, would also be possible but it would not be better. Indeed, in the `receive` (`decrypt`) method, it would be necessary to check the content of received (decrypted) data, to decide if their representing singleton is already instantiated or not.

Finally, the skeleton of the generated method looks like

```
Map<String,Message>
```

```
  generatedSpi(@InputParams@) {
 Map<String,Message> _return =
  new TreeMap<String,Message>();
 try { @GeneratedSpiImpl@ }
 catch { _return = null; }
 return _return;
}
```

where `@InputParams@` gets substituted by the free variables of the translated Spi Calculus process, and `@GeneratedSpiImpl@` by the generated Java code.

4. Soundness

The formal definitions of the translation functions tr_p, tr_t, *ub* and *ret* allow some properties about the generated code to be stated. In order for our translation to work, three minor, reasonable assumptions are made explicit. For any term M, it is assumed that $Ts(M) <: T(M)$ holds. This means that the user-provided concrete class implementing marshalling functions is extending the appropriate abstract SpiWrapper class. For every constructor c offered by the SpiWrapper class $T(M)$, it is assumed that a constructor c' exists in the user-provided class $Ts(M)$ that extends $T(M)$. The c' constructor is assumed to have the same parameters of c and to be implemented only by a call to the `super` method. Finally, it is assumed that $Param(Ts(M))$ returns the correct number and type of user-provided interoperability parameters. Given a well formed Spi Calculus process P, let $t(P) \in 2^{SpiTerm}$ be the set of all the terms in P. Under the assumptions made, the following theorem can be proven.

Theorem 1. *If* $\Gamma \vdash P$ *and return* $\subseteq t(P)$*, then* $tr_p(P, fnv(P), return)$ *is well formed.*

By "well formed" we mean a sequence of Java statements that, when put in the context, forms a Java program that compiles, that is a correct Java program from a syntactic and static semantic point of view. It is worth noting that the terms in $fnv(P)$ are the protocol input parameters, so their corresponding Java variables are already declared in the context, as method input parameters. For brevity, this paper does not include complete proofs of theorems. They can be found in [22].

As an informal proof sketch of theorem 1, consider for example the case where an output process $\overline{M}\langle N\rangle.Q$ is translated into Java. First, it is shown that the tr_t function declares and assigns, if needed, the N term (while M must be a free name because of the type system, so it is already declared and initialized in the context). Then, it is shown by inspection that the code implementing the output process is actually well formed. Finally, it is shown by induction that the code implementing the Q process is well formed, letting the whole generated code be well formed.

Now the semantic properties of the generated Java code are discussed. The main goal is to show that, under some assumptions on the behavior of the SpiWrapper classes, the security properties verified on the Spi Calculus abstract specification are preserved by the generated Java implementation. Since a Dolev-Yao attacker is considered, the focus is on safety security properties that can be defined by means of trace predicates. For example, secrecy and authentication are safety properties. In a Dolev-Yao context, liveness properties cannot be proven, because the attacker is always able to drop messages.

In order to prove security properties preservation from Spi Calculus to Java, it is shown that the generated Java code simulates the corresponding Spi Calculus process. That is, for each trace that can be executed in the Java domain, the same trace exists in the Spi Calculus domain. As a corollary, the Java traces are a subset of the Spi Calculus traces. Finally, if a Spi Calculus specification is proven secure against a safety property, it means that all of its traces are safe, and so the subset of Java traces is. Technically, a weak simulation relation between the generated Java code and the corresponding Spi Calculus is shown. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a concrete process, but each process is still allowed to perform any internal step in between two external states. More details about the weak refinement used here can be found, for example, in [26].

In order to state and prove the simulation relation, the semantics of the Spi Calculus must be written in a slightly different way. According to the notation introduced in section 3, a transition can be in general written as

$$P \xrightarrow{\mathcal{L}} P'\sigma'$$

	`new TMarsh(params)`$, Val, Res$	$\xrightarrow{\tau}{}^{*}$	$o, \{(o,n)\} \cup Val, Res \ \wedge \ n \notin codom(Val)$
	c`.send(`$\mathcal{M}$`)`$, \{(c,c),(\mathcal{M},M)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}\xrightarrow{c!M}\xrightarrow{\tau}{}^{*}$	$unit, \{(c,c),(\mathcal{M},M)\} \cup Val, Res$
	c`.receive(Ts.class,params)`$, \{(c,c)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}\xrightarrow{c?N}\xrightarrow{\tau}{}^{*}$	$\mathcal{N}, \{(c,c),(\mathcal{N},N)\} \cup Val, Res$
$M = N$	$\Rightarrow$ a`.equals(`b`)`$, \{(a,M),(b,N)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	`true`$, \{(a,M),(b,N)\} \cup Val, Res$
$M \neq N$	$\Rightarrow$ a`.equals(`b`)`$, \{(a,M),(b,N)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	`false`$, \{(a,M),(b,N)\} \cup Val, Res$
	`new PairMarsh(`$\mathcal{A}$`,`$\mathcal{B}$`,params)`$, \{(\mathcal{A},A),(\mathcal{B},B)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	$o, \{(\mathcal{A},A),(\mathcal{B},B),(o,(A,B))\} \cup Val, Res$
	o`.getLeft()`$, \{(o,(M,N)),(\mathcal{M},M)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	$\mathcal{M}, \{(o,(M,N)),(\mathcal{M},M)\} \cup Val, Res$
	o`.getRight()`$, \{(o,(M,N)),(\mathcal{N},N)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	$\mathcal{N}, \{(o,(M,N)),(\mathcal{N},N)\} \cup Val, Res$
	`new ShKCMarsh(`$\mathcal{M}$`,`$\mathcal{K}$`,params)`$, \{(\mathcal{M},M),(\mathcal{K},K)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	$o, \{(\mathcal{M},M),(\mathcal{K},K),(o,\{M\}_K)\} \cup Val, Res$
	o`.decrypt(`$\mathcal{K}$`,params)`$, \{(\mathcal{K},K),(o,\{M\}_K)\} \cup Val, Res$	$\xrightarrow{\tau}{}^{*}$	$\mathcal{M}, \{(\mathcal{M},M),(\mathcal{K},K),(o,\{M\}_K)\} \cup Val, Res$

Table 3: Formal semantics for the selected SpiWrapper classes subset.

where $\mathcal{L}$ is the transition label, and σ' is a (possibly empty) substitution that binds variables. If substitutions are not applied, but they are explicitly indicated as postfix operators instead, a generic state P of a system run will be written as $P_1\sigma$, where P_1 includes all the free variables of P, as well as all the bound variables that have been substituted in the past evolution that has led to P, while σ incorporates such substitutions. Using this representation for processes, a generic transition can be written as

$$P_1\sigma \xrightarrow{\mathcal{L}} P'_1\sigma\sigma'$$

and a state can be divided into two components: a process expression followed by a variable substitution. In the LTS for the Spi Calculus, all states are defined as external.

An LTS for a Java sequential program generated by the tr_p function is now defined. In order to relate the Java behavior with the Spi Calculus behavior, the Java LTS uses the same abstract labels used for the Spi Calculus LTS. Let j be the Java code that is going to be executed, $JavaVar$ the set of identifiers that can be used as variables in Java programs, and $JavaObj$ the set of object identifiers. Then a generic state (j, Val, Res) is defined by the code j that is going to be executed, plus a partial function $Val : JavaObj \rightarrow SpiTerm$ and a partial function $Res : JavaVar \rightarrow JavaObj$. Val maps each Java object that has been created by previously executed code to the Spi Calculus term the object is implementing. Res maps each Java variable in the scope of j to the referenced Java object. For example, $Val(o) = \{M\}_N$ means that the Java object o implements the $\{M\}_N$ Spi Calculus term; Res(`var`) $= o$ means that the Java variable `var` references the object o. The intended invariant that should hold is $Val(Res(J(M))) = M\sigma$, where σ is the variable substitution in the corresponding Spi Calculus process. That is, the object referenced by the Java variable $J(M)$ must implement the $M\sigma$ term, which is the run-time value of the M Spi Calculus term. A Java state (j, Val, Res) is defined as external if and only if $j = tr_p(P, dom(Val \circ Res \circ J), return)$ for some Spi Calculus process P that does not begin with a restriction and for some return set $return$. It is worth noting that, since $Val \circ Res \circ J$ is a composition of partial functions, the domain of J is properly restricted such that its codomain is the domain of Res. In turn this is restricted so that its codomain is the domain of Val. The transitions of the form

$$j, Val, Res \xrightarrow{\mathcal{L}} j', Val', Res'$$

take from one generic state to another, following an abstract operational semantics for the Java language.

In this work, an operational semantics is defined that, if implemented by the SpiWrapper classes, makes it possible to have a weak simulation relation between the Spi Calculus process and the generated Java code. The formal semantics for the SpiWrapper classes presented in this work is reported in table 3. A `void` method returns the *unit* value, while `true` and `false` are the boolean values; variable assignment evolves into the *unit* value, but its side effect is to map the variable to the assigned object, formally

$$T(x)\ J(x) = o, Val, Res \xrightarrow{\tau} unit, Val, \{(J(x), o)\} \cup Res$$

Generic types are not taken into account into the definition of the dynamic semantics, because in Java they are implemented by erasure, that is they are checked at compile time and then discarded in the compiled bytecode. For this reason the dynamic semantics can be defined without them. It is assumed that if a method cannot succeed (for example, a wrong key is passed to the `decrypt` method), it throws an exception that simulates the stuck process. Standard congruence and computation semantic rules are assumed for the sequential concatenation of statements, and for the other Java statements.

The simulation relation S that relates external Spi Calculus states to external Java states is formally defined as

$$S(((\nu\overline{n})P)\sigma, (j, Val, Res)) \Leftrightarrow j = tr_p(P, dom(Val \circ Res \circ J), return) \wedge \\ \sigma_{|fnv(P)} = Val \circ Res \circ J_{|fnv(P)} \ \wedge \ \sigma \supseteq Val \circ Res \circ J$$

for any Spi Calculus process P that does not begin with a restriction, and any Val, Res such that $dom(Val \circ Res \circ J)$ is closed under the *subterms* function. Informally, a Spi Calculus state $((\nu\overline{n})P)\sigma$ and a Java state (j, Val, Res) are S-related iff the Java state is external and the invariant $M\sigma = Val(Res(J(M)))$ holds. It is worth noting that the domain of $Val \circ Res \circ J$ is required to contain all the free names and variables in P. However, some compound terms may not (yet) be stored in Java memory (because they will be built by the code generated by the $tr_t(\cdot)$ function). It is enough to require that the invariant holds for the already built terms, which are stored in Java memory.

Theorem 2. *If the SpiWrapper library behaves as specified in table 3, then, for any external state* (j', Val', Res')

$$S((\nu\overline{n})P\sigma, (j, Val, Res)) \wedge \\ j, Val, Res \xrightarrow{\tau}{}^{*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau}{}^{*} j', Val', Res' \Rightarrow \\ P\sigma \xrightarrow{\mathcal{L}} (\nu\overline{m})P'\sigma' \ \wedge \ S((\nu\overline{n})(\nu\overline{m})P'\sigma', (j', Val', Res'))$$

Theorem 2 formally expresses the simulation relation between a Spi Calculus process and its corresponding generated Java program. If the simulation relation holds between a state of a Spi Calculus process and a state of its corresponding Java program, and if the Java program can evolve into a new external state, then the Spi Calculus process can evolve into a new external state too, and the new external states are still related by the simulation relation S.

As an informal proof sketch, consider for example the case where an output process $\overline{M}\langle N\rangle.Q$ is implemented. On the Java side, it is first shown that the code generated by the $tr_t(\cdot)$ function, invoked on N, executes by properly setting up the memory, namely by creating a Java object such that $Val(Res(J(N))) = N\sigma$. Since M must be a free name, $Val(Res(J(M))) = M\sigma$ holds by hypotheses. Then, it is shown that the rest of the code executes as

$$J(M).\texttt{send}(J(N), Param(Ts(N))), Val, Res \xrightarrow{\tau}{}^{*} \xrightarrow{M\sigma!N\sigma} \xrightarrow{\tau}{}^{*} unit, Val, Res$$

On the Spi Calculus side, the process can evolve like

$$P\sigma = (\overline{M}\langle N\rangle.Q)\sigma \xrightarrow{M\sigma!N\sigma} Q\sigma$$

So the same labeled transition happened in both systems, and the simulation relation S can be shown to still hold for the final states.

In general, the Java `send` method could return before the data have actually been sent to the other party; for example it could return as soon as data are buffered by the underlaying operating system. This is not an issue: until data are forwarded, this asynchronous behavior is simulated by the Spi Calculus traces where the attacker does not use the received data. The asynchronous channel behavior would be an issue with restricted channels, because the Java code could evolve to the next external state, while the Spi Calculus process would be stuck. However, the type system prevents the creation of restricted channels inside sequential processes, thus ruling out this issue.

The initial state of a Java program could be an internal state, if the translated Spi Calculus process P begins with a restriction. However, it can be formally shown that the translation of a restriction process leads the Java program to an external state where the simulation relation S holds. So, even the translation of a restriction process is handled, enabling theorem 2, thus getting to the final result that the Java code simulates the Spi Calculus process from which it has been generated.

Theorem 2 implies the preservation of a large class of security properties from Spi Calculus specifications to Java code. More specifically, safety properties such as the common falvors of secrecy and authentication are preserved. The reasoning that leads to this conclusion can be shown by an example. Let P be a generic sequential Spi Calculus process. Suppose that a trace of the Java program (that is a sequence of input/output operations) generated by $tr_p(P)$ allows an attacker to get access to some private data, thus breaking a secrecy property. This means that it is possible to build an attacker process that executes matching input/output operations and finally computes the secret data using only the received messages and the initially known data. By theorem 2, it follows that the same trace is in the trace set of P too. This means that it is possible to build a Spi Calculus process that behaves exactly as the Java attacker. Formal verification of the protocol to which P belongs would show this attack. Conversely, if no attack is found in the formal protocol specification, then the Java program is safe, because it only executes a subset of the traces that have been verified.

Thanks to the formal definition of the SpiWrapper library given in this paper, formal verification of the generated code can be modularized in an assume-guarantee style. In particular, theorem 2 only deals with the Java code implementing the protocol logic, assuming that all low level details, such as dealing with the JCA or sockets, is correctly implemented. Providing a correct implementation of such details is an orthogonal verification problem that can be handled in isolation.

A bi-simulation relation cannot be proven, because it is not possible to show that any trace of a sequential Spi Calculus process can occur in the generated Java program too. As counter-example, consider a Java program stopping execution because of some low-level errors that are not caught by the Spi Calculus specification (for example because of wrong usage of cryptographic parameters). In these cases the Spi Calculus trace can continue, while the Java program stops. One may argue that if a bi-simulation relation could be proven, more kinds of trace properties in addition to the safety ones could be preserved from the Spi Calculus specification to the Java implementation. Although in principle this is true, it is worth noting that security-related properties that are not safety properties cannot be proven with a Dolev-Yao attacker anyway.

4.1. Verification of a SpiWrapper Implementation

As stated by theorem 2, the generated Java code simulates the Spi Calculus process from which it has been generated, by relying on the formal specification of the custom SpiWrapper library. As a step further, an implementation of part of the SpiWrapper library is now presented, that is shown to be correct with respect to its specification reported in table 3. In order to reason on executions of the Java code implementing the library, the Middleweight Java (MJ) framework [27, 28] is used. Essentially, MJ specifies a small-step operational semantics for a rich subset of sequential Java. States are called *configurations*; a configuration is a four-tuple made of (H, VS, CF, FS), where

H is the heap, mapping object identifiers (oids) to their type and to a field function. A field function is a map from field names to values.

VS is the variable stack, mapping variable names to their type and to the referred oid or to their value.

CF is a closed frame of Java code to be evaluated.

FS is a frame stack, that is the program context in which CF is being evaluated.

In the original MJ, transitions are not labeled, because all side effects are captured by the subsequent configuration. In this work, transitions that produce input or output of message M on channel c, are labeled with $c!M$ and $c?M$, while all other transitions are labeled with τ.

A relation between Java states, defined as (j, Val, Res), and MJ configurations is defined, so that the two frameworks can be related. Indeed, states and MJ configurations are very similar, with MJ configurations storing more information, which is unneeded for SpiWrapper behavior specification. In fact, j corresponds to the MJ CF, that is the code to be evaluated; Res serves the same purpose as VS, although VS also stores some other information about variable scopes and types. Since the goal is to verify single method behaviors rather than full program behaviors, the FS context is set empty (denoted by []) before the execution of the method, and it will be empty after method executed. A note must be made on the relation between Val and MJ H, which completes the two frameworks relation. On one hand, Val maps oids to the implemented Spi Calculus terms; on the other hand, H maps oids to their internal

state, which is the value of their fields. So, $Val(o) = M$ (read "object o implements Spi Calculus term M") *means* that $H(o) = v$ (read "object o has fields set as described by v") for some v. For example

$$\begin{aligned} Val(v_p) = (M, N) \quad \Leftrightarrow \quad & H(v_p) = \left(PairS, \begin{array}{l} \texttt{left} \rightarrow v_{\mathcal{M}} \\ \texttt{right} \rightarrow v_{\mathcal{N}} \end{array}\right) \wedge \\ & PairS <: Pair \wedge \\ & Val(v_{\mathcal{M}}) = M \wedge Val(v_{\mathcal{N}}) = N \end{aligned} \tag{1}$$

states that "object v_p implements the pair (M, N)" means that object v_p has a subtype of the *Pair* type, and it has exactly two fields, namely `left` and `right`, pointing to two objects $v_{\mathcal{M}}$ and $v_{\mathcal{N}}$, implementing the M and N terms respectively. It is worth noting that the relation between Val and H is implementation dependent.

MJ does not support generic types, and there is no need to add such support, because, as explained before, generic types are implemented by erasure, so they can be disregarded when analyzing run time behavior.

For brevity, only one method of the `Pair` class is shown in this work, and intermediate evaluation steps are not reported. Verification of all methods of the `Pair` class, along with full evaluation step chains can be found in [22]. In the Spi2Java framework the `Pair` class is abstract and extended by another class implementing marshalling functions. For simplicity, since marshalling functions are irrelevant in this context and adding them would not add value to this example, the `Pair` class is considered to be concrete and marshalling functions are neglected.

Figure 5 shows a possible implementation of the `Pair` class that fits in the MJ framework.

Correctness of the constructor is now shown. Looking up table 3, the initial state of the `Pair` constructor invocation is

$$\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{)}, \{(\mathcal{A}, A), (\mathcal{B}, B)\}, \emptyset$$

In the presented implementation, no additional marshalling parameters are required, so the '`,params`' argument can be neglected.

This state corresponds to the initial MJ configuration

$$(\underbrace{\{(\mathcal{A}, (T_A, \mathbb{F}_A)), (\mathcal{B}, (T_B, \mathbb{F}_B))\}}_{H_0}, \underbrace{[]}_{VS_0}, \underbrace{\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{);}}_{CF_0}, \underbrace{[]}_{FS_0})$$

for some types $T_A, T_B <: Message$ and mapping functions $\mathbb{F}_A, \mathbb{F}_B$ such that the $\mathcal{A}$ and $\mathcal{B}$ objects implement the A and B Spi Calculus terms respectively.

It can be shown that this starting configuration leads to the final configuration

$$(H_0 \cup \{(o, \left(Pair, \begin{array}{l} \texttt{left} \rightarrow \mathcal{A} \\ \texttt{right} \rightarrow \mathcal{B} \end{array}\right))\}, VS_0, o, FS_0)$$

Since the right side of (1) is true, this configuration corresponds to the final state

$$o, \{(\mathcal{A}, A), (\mathcal{B}, B), (o, (A, B))\}, \emptyset$$

The same reasoning applies to the other methods of the `Pair` class.

5. Conclusions

This paper defines a provably correct refinement from Spi Calculus specifications into Java code implementations. Using this refinement, implementations can be automatically generated from specifications, while preserving the security properties verified on the specifications. A type system that allows static types to be assigned to the untyped Spi Calculus terms has been initially defined. The same types are used in the Java language for representing the Spi Calculus terms. Finally, a translation function from well-formed sequential Spi Calculus processes to Java code has been formally defined, so that it is possible to reason on the relation between the Spi Calculus source processes and the generated Java code. The first result is that the translation of a well-formed Spi Calculus process always leads to the generation of well-formed Java code, that is code that compiles. Some features, like protocol return parameters

```
package it.polito.spi2java.spiWrapper;

public class Pair extends Message {
  protected Message left;
  protected Message right;

  public Pair(Message left,
              Message right) {
    super();
    this.left = left;
    this.right = right;
  }

  public Message getLeft() {
    return this.left;
  }

  public Message getRight() {
    return this.right;
  }

  public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Pair) {
      Pair otherPair = (Pair) obj;
      boolean leftOK = this.getLeft()
                .equals(otherPair.getLeft());
      result = leftOK && this.getRight()
                .equals(otherPair.getRight());
    }
    return result;
  }
}
```

Figure 5: A possible implementation of the `Pair` class.

and interoperability of the generated application, which is achieved by letting the user implement the marshalling functions, are also taken into account by the translation function.

As a further step, the generated Java code is proven to be a correct refinement of the Spi Calculus specification in a modular way. First it is shown that the generated Java code implementing the Spi Calculus protocol logic is correct, by assuming correctness of an underlying custom Java library, called SpiWrapper. In order to achieve this result, the formal definition of the intended behavior of the SpiWrapper library has been formalized. Then, it has been shown that the intended behavior of this library can be related to the formal semantics of the Spi Calculus, so that the generated Java code, by properly using the SpiWrapper library, can simulate the Spi Calculus specification.

Finally, it has been shown how an implementation of a class belonging to the SpiWrapper library can be verified correct with respect to the intended behavior. Correctness is proven by evaluating within the MJ framework the Java code that implements the class, and by relating that framework with the one presented here. This result increases confidence about the correctness of the whole system, because only correctness of standard libraries, like the JCA, is assumed, while the custom SpiWrapper library can be proven correct.

The translation function that has been formalized in this paper has been implemented in the spi2java tool, which has been used to successfully generate interoperable implementations of real cryptographic protocols. For example,

in [12] a description of using spi2java for the implementation of the SSH transport protocol can be found.

It is believable that the extension of the results shown in this paper to other (statically typed or not) programming languages is straightforward. Moreover, the approach presented here can be practically used for model-driven-development of provably correct security protocol implementations. With respect to manual development of such applications, the proposed approach allows the developer to concentrate first only on the protocol logic, during formal specification, and later on implementation details, during code generation. However, performances of the generated code may not be always optimized, and manually modifying the code to increase its performances could compromise its security properties. Moreover, only new implementations of protocols can be obtained with model-driven-development, implying some switching costs to substitute the legacy implementation with the newly generated one.

There are still open issues that would complete and improve this work. For instance, more classes belonging to the SpiWrapper library could be proven correct by using the same approach. Another possibility is to link this work to existing proposals [29] of cryptographic libraries that offer provably correct implementations of abstract cryptographic primitives like the ones used in the Spi Calculus.

References

[1] M. Abadi, A. D. Gordon, A calculus for cryptographic protocols the spi calculus, Research Report 149 (1998).

[2] L. Durante, R. Sisto, A. Valenzano, Automatic testing equivalence verification of spi calculus specifications, ACM Transactions on Software Engineering and Methodology 12 (2) (2003) 222–284.

[3] B. Blanchet, An Efficient Cryptographic Protocol Verifier Based on Prolog Rules, in: IEEE Computer Security Foundations Workshop, 2001, pp. 82–96.

[4] M. Abadi, B. Blanchet, C. Fournet, Just fast keying in the pi calculus, ACM Transactions on Information and System Security 10 (3) (2007) 1–59.

[5] M. Abadi, B. Blanchet, Computer-Assisted Verification of a Protocol for Certified Email, Science of Computer Programming 58 (1–2) (2005) 3–27.

[6] D. Dolev, A. C.-C. Yao, On the security of public key protocols, IEEE Transactions on Information Theory 29 (2) (1983) 198–207.

[7] K. Bhargavan, C. Fournet, A. D. Gordon, S. Tse, Verified interoperable implementations of security protocols, in: Computer Security Foundations Workshop, 2006, pp. 139–152.

[8] J. Jürjens, Verification of low-level crypto-protocol implementations using automated theorem proving, in: Formal Methods and Models for Co-Design, 2005, pp. 89–98.

[9] J. Goubault-Larrecq, F. Parrennes, Cryptographic protocol analysis on real C code, in: Verification, Model Checking, and Abstract Interpretation, 2005, pp. 363–379.

[10] K. Bhargavan, C. Fournet, A. D. Gordon, Verified reference implementations of WS-security protocols, in: Web Services and Formal Methods, 2006, pp. 88–106.

[11] D. Pozza, R. Sisto, L. Durante, Spi2java: Automatic cryptographic protocol java code generation from spi calculus, in: International Conference on Advanced Information Networking and Applications, 2004, pp. 400–405.

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

[13] B. Tobler, A. Hutchison, Generating network security protocol implementations from formal specifications, in: Certification and Security in Inter-Organizational E-Services, Toulouse, France, 2004.

[14] A. Pironti, R. Sisto, Formally sound refinement of Spi Calculus protocol specifications into Java code, in: IEEE High Assurance Systems Engineering Symposium, 2008, pp. 241–250.

[15] C.-W. Jeon, I.-G. Kim, J.-Y. Choi, Automatic generation of the C# code for security protocols verified with casper/FDR, in: International Conference on Advanced Information Networking and Applications, 2005, pp. 507–510.

[16] E. Hubbers, M. Oostdijk, E. Poll, Implementing a formally verifiable security protocol in java card, in: Security in Pervasive Computing, 2003, pp. 213–226.

[17] S. Bajaj et al., Web services policy 1.2 - framework (WS-policy), W3C Recommendation (2006).

[18] S. Bajaj et al., Web services policy 1.2 - attachment (WS-policyattachment), W3C Recommendation (2006).

[19] K. Bhargavan, C. Fournet, A. D. Gordon, G. O'Shea, An advisor for web services security policies, in: Workshop on Secure web services, 2005, pp. 1–9.

[20] A. Nadalin, C. Kaler, P. Hallam-Baker, R. Monzillo, OASIS web services security: SOAP message security 1.1 (WS-security 2004) (2006).

[21] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, parts I and II, Information and Computation 100 (1) (1992) 1–77.

[22] A. Pironti, R. Sisto, Correctness-preserving translation from Spi Calculus to Java, revision 3, Tech. rep., Politecnico di Torino (Jun. 2009).
URL `http://staff.polito.it/riccardo.sisto/reports/translation3.pdf`

[23] A. Igarashi, N. Kobayashi, A generic type system for the pi-calculus, Electronic Notes in Theoretical Computer Science 311 (1-3) (2004) 121–163.

[24] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[25] A. Pironti, R. Sisto, Soundness conditions for message encoding abstractions in formal security protocol models, in: International Conference on Availability, Reliability and Security, 2008, pp. 72–79.

[26] G. Schellhorn, ASM refinement and generalizations of forward simulation in data refinement: a comparison, Theoretical Computer Science 336 (2-3) (2005) 403–435.

[27] G. Bierman, M. Parkinson, A. Pitts, MJ: An imperative core calculus for Java and Java with effects, Tech. Rep. 563, Cambridge University Computer Laboratory (2003).
[28] U. Sammapun, R. Sharykin, M. DeLap, M. Kim, S. Zdancewic, Formalizing Java-MaC, Electronic Notes in Theoretical Computer Science 89 (2) (2003) 171–190.
[29] M. Backes, B. Pfitzmann, M. Waidner, A composable cryptographic library with nested operations, in: ACM Conference on Computer and Communications Security, 2003, pp. 220–230.