Formally based semi-automatic implementation of an open security protocol

Formally based semi-automatic implementation of an open security protocol

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

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

Abstract

This paper presents an experiment in which an implementation of the client side of the SSH Transport Layer Protocol (SSH-TLP) was semi-automatically derived according to a model-driven development paradigm that leverages formal methods in order to obtain high correctness assurance. The approach used in the experiment starts with the formalization of the protocol at an abstract level. This model is then formally proved to fulfill the desired secrecy and authentication properties by using the ProVerif prover. Finally, a sound Java implementation is semi-automatically derived from the verified model using an enhanced version of the Spi2Java framework. The resulting implementation correctly interoperates with third party servers, and its execution time is comparable with that of other manually developed Java SSH-TLP client implementations. This case study demonstrates that the adopted model-driven approach is viable even for a real security protocol, despite the complexity of the models needed in order to achieve an interoperable implementation.

Keywords: Model-driven-development, Security protocols, Automatic code generation, Spi2Java

1. Introduction

Security protocols are communication protocols that use cryptographic primitives in order to protect some assets. Despite their apparent simplicity, designing and implementing security protocols correctly is difficult. At least three reasons that justify this complexity can be identified: the distributed nature of the environment, which generates a large (usually unbounded) number of scenarios to be dealt with; the (possibly bad) interactions between different cryptographic primitives; and the presence of an active attacker that is not controlled by the protocol designer or implementer. Regarding the first reason, it

*Corresponding author. Tel: +390110907073; Fax: +390110907099
Email addresses: alfredo.pironti@polito.it (Alfredo Pironti), davide.pozza@polito.it (Davide Pozza), riccardo.sisto@polito.it (Riccardo Sisto)

is worth remarking that, even if experience suggests that attacks on security protocols are normally possible with just a few protocol sessions, it is difficult to know in advance the number of sessions required. More importantly, the unconstrained behavior of the attacker or the potentially honest-but-curious attitude of a protocol participant may be enough to puzzle a human analyzer even when considering only a few sessions, because attention is normally focused on the functional part, i.e. that the protocol does something useful, which makes it very difficult to think about how to break the protocol.

On the protocol design side, formal methods can provide rigorous assurance that a desired property holds for a given protocol model. The property is usually assured to hold under some assumptions about the attacker's capabilities and the behavior of the cryptographic primitives, which form the model according to which the protocol is verified. Several attacker and cryptographic primitive models, as well as several verification techniques, have been proposed in order to formally verify abstract security protocol specifications (e.g. Blanchet, 2001; Blanchet and Pointcheval, 2006; Durante et al., 2003; Escobar et al., 2009; Lowe, 1998; Mödersheim and Viganò, 2009).

On the protocol implementation side, different techniques can be used to enhance confidence regarding implementation correctness. By using techniques such as testing or code reviews, only a limited number of scenarios is considered, which may not be enough, especially when the protocols are deployed in safety or mission critical applications. It is important to note that some subtle implementation errors, such as the omission or the wrong execution of a prescribed check, do not infringe interoperability. They can only be discovered, by testing, if the test patterns include one of the particular protocol scenarios where the implementation should abort a session while, in fact, it does not. Even if these scenarios are very special cases, attackers who know of a vulnerability will trigger it on purpose. Again, in the implementation phase formal methods can provide the necessary assurance level. This is done by formally linking security protocol implementations to their formal specifications, so that an implementation is proved to preserve the same security properties that are guaranteed to hold on its formally verified specification, under the same assumptions.

Model-Driven Development (MDD) is one of the possible approaches for achieving this linkage. MDD is based on designing a model, from which code is later automatically generated. If the model is expressed in a formal language, it can be formally verified, and code can be automatically produced in such a way that the verified properties are preserved in the generated code.

This paper presents an MDD technique for security protocols applied to the SSH Transport Layer Protocol (SSH-TLP) (Ylonen and Lonvick, 2006a,b). First, a formal model is written according to the SSH-TLP RFC, and secrecy and authentication are formally verified to hold under the Dolev and Yao (1983) perfect cryptography assumptions. This model differs from already existing Dolev-Yao models of SSH-TLP (e.g. the ones provided with the ProVerif tool, or by the AVISPA project), because it includes enough detail to let an interoperable Java implementation be derived. Because of such protocol details, verification is more challenging on this model than on the simplified models that are usually

analyzed just to verify the main protocol logic. From this formally verified model, an implementation of an SSH-TLP client is then soundly derived. An SSH-TLP server could have been derived as well, requiring a similar effort, but this would not have added anything fundamentally new to the case study.

The design and development of both model and application are performed with the support of state of the art tools. Formal verification of the model is done by means of the ProVerif tool (Blanchet, 2001), while the implementation code is semi-automatically generated by means of an enhanced version of the Spi2Java code generation framework (Pozza et al., 2004).

The main contribution and motivation of this paper is to present this formally-based model-driven approach for security protocols, and to show its practical viability by handling the full version of a real protocol. Furthermore, the lessons learned about the strengths and current limitations of the approach are reported.

Even if the formal analysis of Dolev-Yao models can only exclude the main protocol logic errors, the model-driven technique presented here provides a higher assurance level for the final product than manual implementation, because a formal proof rules out all such logical errors. Even better assurance could be obtained by using the more precise computational models (e.g. Blanchet, 2008) for verification. However, although some automated tools for analyzing computational models have just appeared, the scalability and automation of these tools have still to be improved before they are ready for production environments. So, the approach presented here, using Dolev-Tao models, can benefit from the maturity and full automation of formal verification tools.

This paper extends the work presented by Pironti and Sisto (2007), where a preliminary, not formally verified and stripped-down version of SSH-TLP was considered. Since then, the Spi2Java framework has been extended, enabling the development of abstract models that include all the main protocol features.

The paper gives an account of the current Spi2Java features and how they have been exploited to develop an implementation of the SSH-TLP protocol. For the sake of brevity, only small parts of the full SSH-TLP formal model are reported in this paper. The full model that was formally verified, and the full source code of the generated client can be found online (Pironti et al., 2011).

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 gives an overview of the MDD workflow that was used to develop the SSH-TLP case study with the Spi2Java framework. Section 4 presents the abstract formal modeling of SSH-TLP, while section 5 shows the results of formal verification of the developed model. Section 6 describes the steps followed for semi-automatic generation of a SSH-TLP client from its specification. Section 7 discusses correctness assurance provided by this model-driven approach, section 8 presents the results of interoperability and performance testing, and section 9 concludes the paper.

2. Related Work

Formally linking security protocol models and their implementations is a relatively recent research field. During recent years, several approaches have

been investigated.

One technique consists of extracting a model from an already existing security protocol implementation source code. The independent works by Bhargavan et al. (2006a); Jürjens (2009); O'Shea (2008) are different examples of this approach. The (over-approximated) extracted model is then verified for the desired security properties. Bhargavan et al. (2006a) prove soundness of the model extraction algorithm under the Dolev-Yao abstraction, so that authenticity and secrecy properties verified on the formal model also hold on the implementation. In principle, this approach has the advantage of allowing existing implementations to be verified without changing the way applications are currently written. However, currently so many constraints are required on the implementation source code that only *ad-hoc* written programs can be verified. This approach has already been tested on real protocol implementations, as documented for example by Bhargavan et al. (2006b, 2008). On the other hand, Jürjens (2009) starts from somewhat arbitrary Java implementations of cryptographic protocols, but formal soundness proofs are lacking, and the extraction process is not fully automatic.

Another available technique is to add refinement types to an existing implementation source code, so that some security properties can be proved directly on the implementation (Bengtson et al., 2011; Bhargavan et al., 2010). This approach has the potential of being applied to real protocols, although the supported source code is a restricted version of an ML-like functional language, as opposed to the imperative or object-oriented languages such as C, C++ or Java often used to develop implementations of security protocols. Another limitation of this approach is that adding refinement types requires expertise not usually available in protocol implementation developers.

Code generation is a further technique, where the developer starts from a formally verified model of a security protocol, and semi-automatically derives an implementation from it. In principle, this approach may require the developer to deal with the formal protocol model, which is usually described in a (domain-specific) formal modeling language. Nevertheless, since the formal model abstracts some low level details away, it is simpler than the full implementation and the developer can concentrate attention only on the protocol logic aspect during the model design phase, and later only on the low level aspects, when deriving the implementation from the model.

The independent works by Bangerter et al. (2008); Bhargavan et al. (2009); Hubbers et al. (2003); Jeon et al. (2005); Kiyomoto et al. (2008); Pironti and Sisto (2007); Pozza et al. (2004); Song et al. (2001); Tobler and Hutchison (2004) are different implementations of the code generation approach. Most of them do not give the user full control over aspects such as message encoding, thus being unable to generate interoperable implementations of open protocols. Another limitation found in most of these works is the lack of soundness proofs about the refinement steps, so that there is no formal link between formal models and the implementations derived from them.

Grandy et al. (2008) give a formal proof of correctness for an interoperable Java Card implementation of the Mondex protocol. Unfortunately, the results

by Grandy et al. (2008) are not general, but specifically tailored to the analyzed Mondex case. This means that the results are not easily reusable, and they could not be directly adopted for the SSH-TLP case study considered.

SecureMDD (Moebius et al., 2009) is a model-driven approach based on UML representation of security protocols. From the UML model both a formal model based on state machines and an interoperable Java Card implementation can be obtained. UML is a powerful modeling language and, compared to spi calculus, it is relatively easier to use. However, UML lacks a formal semantics, and the SecureMDD approach does not provide a formal proof that links together the UML model, the generated formal model based on state machines and the generated Java Card code.

Other work has focused on graphical modeling of security protocols, such as the UML-based SecureUML by Basin et al. (2006) and UMLSec by Jürjens (2005), or the GSPML graphical language (McDermott, 2005) which can be translated into CSP models. These graphical languages can be used to describe security protocols and to verify their properties, but none of these works address code generation from the model.

The only general approach that both admits in principle the development of interoperable implementations of security protocols and provides a formal soundness proof of the refinement steps is Spi2Java (Pironti and Sisto, 2007, 2010; Pozza et al., 2004).

Up to now, the applicability of Spi2Java to real open protocols has been only partially tested by Pironti and Sisto (2007), where a stripped-down version of the SSH-TLP protocol was implemented. Since then, the framework has been extended and now it accepts a richer input language. In this way, protocol models can be made closer to real protocol implementations, thus improving overall correctness assurance.

3. SSH-TLP Client Application Development with Spi2Java

The workflow that was used to develop an implementation of the SSH-TLP client side with Spi2Java is depicted in figure 1. Starting from the SSH-TLP RFCs, a formal model of the whole protocol was manually developed. This model includes all protocol actors, and is enriched with the definition of the desired security properties. The model was then fed to the ProVerif formal verification tool, to ensure the security goals were met.

Then, the whole protocol model was parsed by the Spi2Java spi calculus parser, to extract the client side model. The Spi2Java refiner was run a first time on the client model, to automatically infer the implementation details that were missing in the spi calculus model. These implementation details were manually refined, in order to match the requirements specified in the SSH-TLP RFCs.

When all the implementation details were filled, the Spi2Java Java code generator was used to automatically generate the code implementing the protocol logic. This protocol logic is linked against a Spi2Java provided library, called

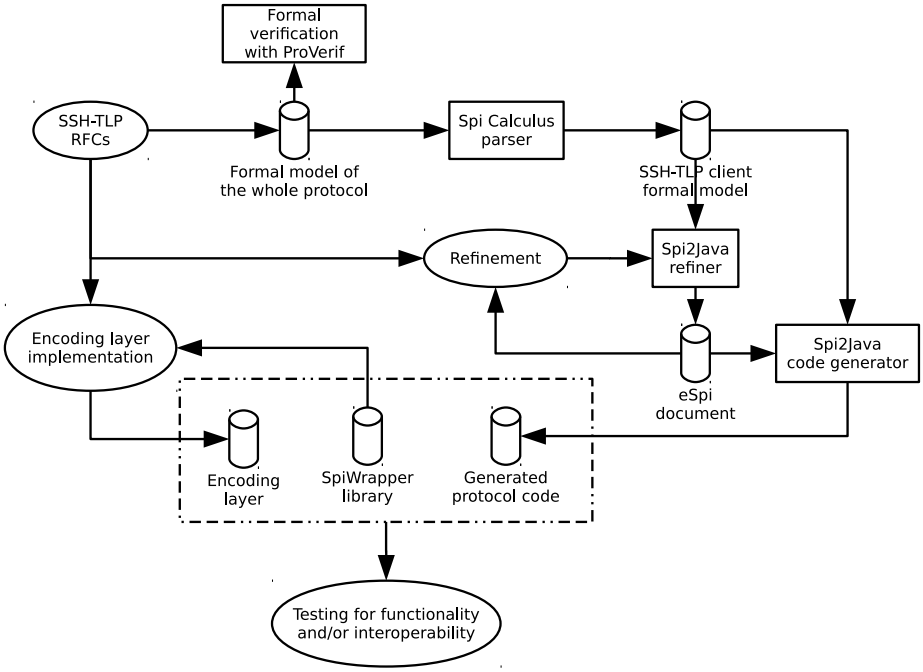

Figure 1: SSH-TLP application development workflow with Spi2Java.

SpiWrapper, and a library containing data marshaling functions. The latter library was manually developed, and is customized to fit the SSH-TLP binary packet representation specified in the RFCs.

The application obtained was finally tested for interoperability with third party servers, and for reliability against maliciously crafted sessions of the protocol.

4. Formal Model

4.1. Formal language

In the Spi2Java framework, formal models of security protocols are specified by an extension of the spi calculus language (Abadi and Gordon, 1998). With respect to the original definition of spi calculus, the language accepted by Spi2Java now includes else branches, additional operations and some syntactic sugar. Moreover, Spi2Java is extensible in order to handle more cryptographic primitives. The input language of Spi2Java can be considered as a subset of the input language accepted by the verification tool ProVerif, albeit with a syntax that is not exactly the same. A simple syntax translation can automatically transform a spi calculus specification accepted by Spi2Java into a corresponding ProVerif input file. The part of the syntax accepted by Spi2Java that has been used in the case study is shown in tables 1 and 2.

Table 1: Term syntax of spi calculus (the part used for the case study).

σ, ρ, τ ::=	terms
m	name
x	variable
(σ, ρ)	pair
$\mathrm{H}(\sigma)$	hashing
$DHPub(\sigma)$	DH public part
$DHKey(\sigma, \rho)$	DH shared key
$\sigma\sim$	shared-key
$\{\sigma\}\rho$	shared-key encryption
$\sigma+$	public key
$\sigma-$	private key
$\{[\sigma]\}\rho$	public-key encryption
$[\{\sigma\}]\rho$	private-key encryption (signature)

Briefly, a spi calculus specification is a set of process definitions $P, Q, ...$, written using the syntax in table 2. Process definitions make use of terms which are abstract representations of data, written using the syntax in table 1.

In typical client-server protocols, like SSH-TLP, several client and server instances may run concurrently, participating in protocol sessions. This is usually modeled for formal verification by defining a top level process $Inst$ that instantiates an unbounded number of client and server processes, thus representing any scenario involving concurrent protocol sessions. For example, if C and S are the client and server processes respectively, then the $Inst$ process can be defined as

$$Inst \triangleq !C \mid !S$$

where the parallel composition expression $P \mid Q$ means parallel execution of processes P and Q, and the replication expression $!P$ means an unbounded number of instances of P running concurrently.

In the Dolev and Yao (1983) modeling approach, all data are represented symbolically as terms of an algebra and all operations on data, including cryptographic primitives, are represented as algebraic operators that are applied over terms in order to build new terms. For example, in the spi calculus, given terms σ and ρ, term $\{\sigma\}\rho$ represents the encryption of σ under symmetric key ρ, while $H(\sigma)$ represents the result of computing a cryptographic hash function on σ. Other operators are available for other kinds of encryption, for pairing messages into structured messages, and for building shared keys from their key material or for extracting public and private keys from key pairs. The $DHPub()$ and $DHKey()$ are the operators that represent Diffie-Hellman (DH) public part generation and DH shared secret derivation respectively. During formal verification with ProVerif, the DH equation $DHKey(x, DHPub(y)) = DHKey(y, DHPub(x))$ (Blanchet et al., 2008) will be taken into account to cor-

Table 2: Process syntax of spi calculus (the part used for the case study).

$P, Q ::=$	process behavior expressions
$\sigma\langle\rho\rangle.P$	output
$\sigma(x).P$	input
$P \mid Q$	parallel composition
$!P$	replication
$(@\, m)\ P$	restriction
0	nil
$let\ x\ =\ \sigma\ in\ P$	assignment
$[\sigma\ is\ \rho]\ P$	match (equality check)
$[\sigma\ is\ \rho]\ (P)\ else\ (Q)$	
$let\ (x, y)\ =\ \sigma\ in\ P$	pair splitting
$let\ (x, y)\ =\ \sigma\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ \{x\}\rho\ in\ P$	shared-key decryption
$case\ \sigma\ of\ \{x\}\rho\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ \{[x]\}\rho\ in\ P$	private-key decryption
$case\ \sigma\ of\ \{[x]\}\rho\ in\ (P)\ else\ (Q)$	
$case\ \sigma\ of\ [\{x\}]\rho\ in\ P$	public-key decryption
$case\ \sigma\ of\ [\{x\}]\rho\ in\ (P)\ else\ (Q)$	
$check\ \sigma\ of\ \tau\ with\ \rho\ in\ P$	signature check
$check\ \sigma\ of\ \tau\ with\ \rho\ in\ (P)\ else\ (Q)$	

rectly model and verify the DH key exchange. The primitive terms are names, i.e. symbolic representations of unstructured plain data, and variables.

All names are assumed by default to be publicly known. However, a process can make new restricted names, representing private data initially not known by the attacker. Creation of a new private name m is represented in spi calculus by the process $(@\, m)\ P$ which is like P, but with the additional specification that m is a private name in P. A private name can be used, for example, to model a secret key initially not known by the attacker or a nonce, i.e. a randomly generated large number.

The spi calculus processes communicate over channels. Process $\sigma\langle\rho\rangle.P$ sends message ρ over channel σ and then behaves like process P; process $\sigma(x).P$ receives a message on channel σ, stores it in variable x, then behaves like process P, where x is bound to the received message. Like any other term, a channel can be either public, so that the attacker has access to it, or private. For instance, in the SSH-TLP model presented here, clients and servers communicate over a public channel `cAB`, and each client and server process has its own private channel `ks`, used to exchange data with its local key store.

In spi calculus, a Dolev-Yao attacker is implicitly modeled as an environment process that can eavesdrop, delete, inject or alter any message over public channels. On the contrary, none of these actions can be performed on private

channels. For each message it reads, the attacker increases its knowledge, which can then be used by the attacker to forge new messages or to alter existing ones in order to break the protocol. No other assumptions are made on the attacker, so that any behavior (including the worst one) is considered during formal verification.

Associated with term construction operations (the ones shown in Table 1), there are corresponding inverse operations (e.g. decryption or pair splitting shown in Table 2) also known as destructors. For example, $case\ \sigma\ of\ \{x\}_\rho.P$ decrypts term σ using symmetric key ρ. If decryption is successful, the result of decryption is assigned to variable x and the process continues as specified by process P. If decryption fails, the process gets stuck. An optional $else$ branch in the construct specifies an alternative behavior to be adopted when decryption is unsuccessful, instead of getting stuck. Similar constructs are available for different kinds of encryption. An additional construct, not available in the original spi calculus, is signature check. $check\ \sigma\ of\ \rho\ with\ \tau\ in\ P$ checks whether σ is a valid signature of message ρ using public key τ. If this is the case, then process P is executed; otherwise the process gets stuck, or it behaves like process Q if an $else\ Q$ branch follows.

The algebraic properties of the term algebra represent the ideal properties of cryptographic operations. For example, an ideal property of symmetric cryptography is that a plaintext can be recovered from the ciphertext only if the encryption key and the ciphertext are available. Accordingly, in the term algebra the only way to compute σ from $\{\sigma\}_\rho$ is by decrypting $\{\sigma\}_\rho$ with ρ. All protocol participants and the attacker have access to the algebraic version of the cryptographic primitives. In particular, this gives reasonable power to the attacker, which can decrypt exchanged messages (if and only if it has the required keys) and can create new cryptographic terms in order to possibly break the protocol.

Extending the language with new operations on terms and their corresponding destructors is straightforward. The mapping into the input language of ProVerif can be easily extended accordingly, because ProVerif allows definition of constructors and destructors in the input file itself.

4.2. Modeling the SSH-TLP protocol in spi calculus

SSH-TLP is informally specified by Ylonen and Lonvick (2006a,b). For the sake of clarity, a typical SSH-TLP scenario is provided in figure 2. In the first two messages, client and server start a protocol session by exchanging their identifiers ID_C and ID_S respectively. Then, with messages three and four, client and server negotiate session algorithms by exchanging two random nonces (c_cookie and s_cookie) and their lists of supported algorithms ($c\_algorithms$ and $s\_algorithms$), from which the agreed ones will be selected. Finally, with the fifth message, the client sends its DH public key e to the server, and the server replies with the last message, providing its public key $PubKey\_s$ used for digital signature, its DH public key f, and the signed *final hash*, upon which client and server perform authentication agreement. At the end of the protocol,

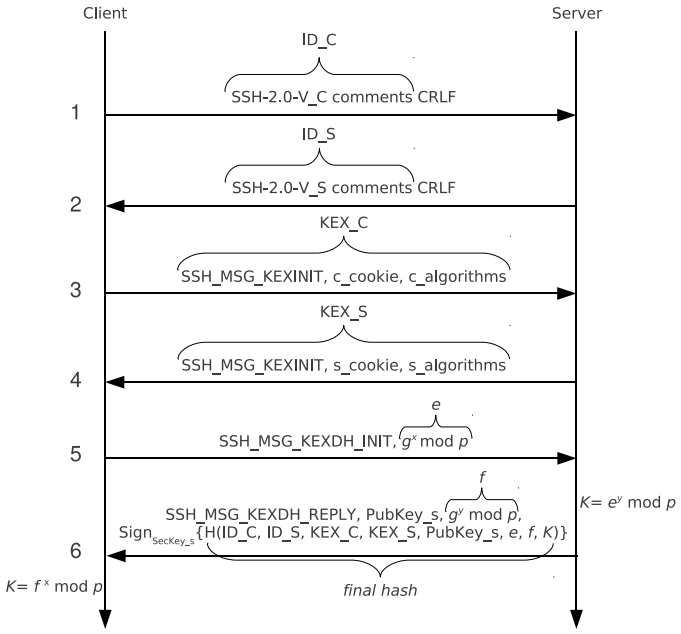

Figure 2: SSH Transport Layer Protocol typical scenario.

both client and server can derive a shared DH session secret K, which is later used to generate several session keys.

In order to enable formal verification, a spi calculus model of the full SSH-TLP protocol, including client, server, the key stores, and the $Inst$ process, was manually written. This model includes significant protocol features that were neither considered by Pironti and Sisto (2007) nor in other existing Dolev-Yao models of SSH-TLP:

- Algorithm negotiation is modeled;
- Cryptographic parameters related to the agreed algorithms are taken into account in both cryptographic primitives and in data upon which authentication agreement is performed;
- Interaction between each protocol agent and its key store is modeled;
- Error states of the protocol are considered, and emission of proper error messages is modeled.

Modeling such features is essential in order to later derive a working and interoperable Java implementation of the protocol that closely corresponds to the formal model. Furthermore, including these details in the model that is formally verified leads to a proof of correctness of the real protocol as opposed to the proofs of correctness of simplified versions of this protocol that previously appeared in the literature.

Optional aspects have not been included in the model, thus considering the smallest protocol configuration compliant with the standard. Only one mandatory aspect of the standard protocol logic was left out of the model, namely the management of the SSH-TLP `first_kex_packet_follows` flag. Handling this flag, which was introduced as a way to optimize implementation performance, is rather complex and this option is hardly ever used in real implementations.

While the protocol logic is fully modeled (albeit with the exception of the `first_kex_packet_follows` flag), data marshaling functions that are used to convert data to/from serial binary streams before transmission or before applying cryptographic operations are not included at all in the formal model. Indeed, according to the Spi2Java development approach, they will be added later, after formally verifying the model, in a separate refinement step. In this way, formal verification does not become too complex. At the same time, formally proving the relevant security properties (i.e. secrecy and authentication under Dolev-Yao assumptions) on the model that does not include these functions has been proved sufficient to conclude that the same properties hold when such functions are added, provided the added functions satisfy some general assumptions such as injectivity and non-leakage of secret data (Pironti and Sisto, 2008; Pironti, 2010). These assumptions are normally fulfilled by the code implementing such functions, and this can be verified in isolation by state-of-the-art information flow tools and model checking tools for sequential code.

In its full length, the model of SSH-TLP discussed here amounts to 254 lines. For the sake of brevity, and since simpler versions of the model have appeared in the literature, only the significant aforementioned additional aspects of the client-side model are discussed in detail in the next subsections. The complete protocol specification can be found online (Pironti et al., 2011).

4.3. Algorithm Negotiation

With messages 3 and 4, client and server exchange their lists of supported algorithms, from which the session-algorithms are negotiated. Client and server must agree on ten different algorithms. For each algorithm that must be agreed upon, both client and server propose a list of supported algorithms, ordered by preference. Each algorithm is finally agreed by choosing the first algorithm in the corresponding client list that is also available in the corresponding server list.

In the preliminary version presented by Pironti and Sisto (2007) algorithm negotiation was neglected. It was modeled instead that the client would accept the first algorithm proposed by the server, even if this algorithm was not even supported by the client. Here, algorithm negotiation is modeled, so that the generated application is fully RFC-compliant.

Modeling negotiation explicitly in spi calculus is possible but leads to models that are hard to verify, because the lists of supported algorithms are unbounded, and verification tools like ProVerif fail to terminate when faced with open-ended data structures. In order to avoid this issue, and to limit the complexity of the model, the negotiation procedure for each one of the ten algorithms was abstractly modeled by a one-way function that operates on the relevant pair of

algorithm lists (the client list and the server list). For example, negotiation of the key exchange algorithm based on the corresponding client and server lists of supported key exchange algorithms is modeled by the following expression:

```
let agreed_kex_algorithm =
   H((c_kex_algorithms, s_kex_algorithms)) in
```

where `c_kex_algorithms` and `s_kex_algorithms` are respectively the client and server lists of supported key exchange algorithms, and `agreed_kex_algorithm` is the negotiated key exchange algorithm. Using a hash function to model the algorithm decision procedure reflects the fact that negotiation "transforms" the pair of algorithm lists into the agreed algorithm and from the output of the procedure no information can be derived about the original pair of algorithm lists, except they both contained the agreed algorithm. Modeling this with a hash function may seem too constraining for the attacker's knowledge, because, according to this model, the attacker could not infer anything about the originating pair of lists from the agreed item. However, this is not an issue for the SSH-TLP, and this also holds for many other protocols, because the full algorithms lists are not secret. Another issue of this simplified modeling approach is that two different negotiations that should give the same result actually give different results. The inability to recognize equality of the results of negotiation may lead in principle to unsound verification, because an equality test that fails in the model could succeed in reality, leading to real behaviors that are not considered by formal verification.

In principle, new operators could be added to spi calculus and Spi2Java, with their semantics formally defined in the ProVerif model, and their implementation provided as part of the SpiWrapper library. However, on the one hand this process would be rather complex and delicate for the average user of Spi2Java, because wrongly specifying or implementing a new operator could be a source of unsoundness. On the other hand, the granularity at which such new operators can be formally defined in ProVerif is limited; for instance, it would be hard to properly model custom properties as was done in the Mondex case study (Grandy et al., 2008). Indeed, there exists a trade-off between the generality of the approach, and its applicability to specific protocols.

Even with this slight limitation, the model is closer to reality than the one presented by Pironti and Sisto (2007).

4.4. Cryptographic Parameters Handling

When a protocol implementation invokes cryptographic operations it must use proper algorithms and parameters (e.g. 3DES or AES, with specific key length) for each of them, instead of the symbolic encryption functions. This kind of information is often abstracted away from Dolev-Yao models but it is of course relevant for implementation. It is crucial that these algorithms and parameters can be set independently for each cryptographic operation, and that they can either be specified at compile-time, or be resolved at run-time. For instance, there are protocols, like SSH-TLP, where the encryption of incoming

and outgoing data is performed independently, each using its own algorithms and keys. Furthermore, such algorithms and corresponding key lengths are selected at run-time according to the result of negotiation.

Spi2Java lets the user add this information in the refinement phase by specifying it in a separate XML document, called the eSpi document. Essentially, for each spi calculus term representing a cryptographic operation, the user can specify its algorithm and parameters, either statically (so that they are resolved at compile-time) or by referring to another spi calculus term that will contain the value of the algorithm or parameter at run-time.

Run-time resolution of negotiated cryptographic algorithms is straightforward, because in the model the computation of such negotiation is represented by a term holding the agreed algorithm. So, in the eSpi document, it is sufficient for the user to refer to that term in any cryptographic operation that must use the negotiated algorithm.

Instead, the parameters associated to a negotiated algorithm are not explicitly represented in the model by default, nevertheless they might depend on the negotiated algorithm. For example, the length of a digest may depend on which digest algorithm is selected. In such cases, a term, which holds the value of the parameter depending on the negotiated algorithm, must be explicitly included in the model so that it can be later referenced in the eSpi document.

For example, let us consider the negotiation of the DH key exchange algorithm in SSH-TLP.

The SSH-TLP RFC prescribes that different DH groups must be supported for key exchange. When a key exchange algorithm is negotiated, in fact the DH group to be used is negotiated. In turn, agreement on a DH group implies the usage of specific values for the parameters p, a large safe prime, g, a generator for a subgroup, and q, the order of the subgroup. In the proposed spi calculus model, such cryptographic parameters are explicitly represented by computing a function over the term storing the negotiated DH group (called `agreed_kex_algorithm`), and a marker identifying the cryptographic parameter to be extracted. Once again this function is modeled as a hash function, which is safe for secrecy since all the terms involved are not secret (they are already known by the attacker anyway). An issue similar to the one explained for algorithm negotiation could arise in principle here too, but in this case it is less likely to occur, because the parameters for different algorithms are generally different.

In practice, the spi calculus model where p, g and q are derived from the agreed DH key exchange algorithm can be written as

```
let p = H(agreed_kex_algorithm,pParam) in
let g = H(agreed_kex_algorithm,gParam) in
let q = H(agreed_kex_algorithm,qParam) in
```

where `pParam`, `gParam` and `qParam` are the constant markers identifying which parameter to extract, and the variables `p`, `g` and `q` store the parameters so derived. During the refinement phase, the latter three variables can be referenced as run-time resolved parameters for the DH cryptographic operations.

This same modeling pattern is equally applied to the parameters that depend on the other negotiated algorithms.

4.5. Key Store Access

After receiving message 6 from the server, the client should check that the received server public key `PubKey_s` matches the trusted one. Note that in fact more than one trusted public key can be associated with the same server, namely one key for each supported signature algorithm (e.g. RSA, DSA) and for each IP address.

In the model, the check on the received public key is done like a typical implementation would do it, i.e. by first retrieving from a local key store the trusted key for the negotiated signature algorithm associated with the connected server, and then by matching the two keys.

Interaction of a process P with the key store is modeled in Spi2Java by pairs of output/input (request/response) message exchanges between P and a process representing the key store, over a private channel. This model is flexible enough to allow both formal verification and derivation of an implementation. First, P sends the key store a request, that is a message of the form $(Operand, Data)$, where *Operand* specifies the requested operation, and *Data* are the application data specific to the given operand. In turn, the key store emits its response by sending an outcome back, whose form depends on the requested operation.

Two operands are defined for the key store, namely `CHECK_KEY` and `GET_KEY`. For both operands, application data is a key alias. `CHECK_KEY` is used to ask the key store whether a key associated with the given alias exists, while `GET_KEY` is used to actually retrieve the key. By distinguishing these two operations, key-retrieval errors (e.g. key not found in the key store) can be properly handled.

In our SSH-TLP client model, the key alias for a server key is a list made up of the server identification string, the negotiated signature algorithm and the public communication channel, which represents the server IP address information.

As an example, in the SSH-TLP client model, the server trusted key is retrieved via the following spi calculus excerpt:

```
ks<(GET_KEY,(ID_S,signKeyType,cAB))>.
ks(stored_PubKey_s).
```

where the application data are composed of `ID_S` (the server identification string), `signKeyType` (the agreed signature algorithm), and `cAB` (the public communication channel).

On the server side, the same approach is used to let the server retrieve its own private key from its key store, depending on the agreed signature algorithm.

4.6. Error Handling

In general, error conditions are handled by sending the appropriate SSH-TLP error message to the other party, through the `cAB` channel, and by reporting the

error to the user via a private channel (the `cState` channel), which is also used to report the end of a successful protocol run.

Such error conditions are usually detected because the `else` branch of a failed operation is taken. For example, on the client side, if the signature check fails, the `else` branch is taken where an "SSH DISCONNECT" message, with cause "Key Exchange Failed", is sent to the server over `cAB`, and then the error is also reported over `cState`.

5. Formal Verification of the SSH-TLP Model

The formal model developed as shown in the previous section can be verified using the ProVerif theorem prover (Blanchet, 2001). Being a theorem prover, ProVerif can handle an unbounded number of protocol sessions. At the same time, it offers full automation and can often report counter-examples. In addition, both ProVerif and Spi2Java accept (variations of) the same input language, thus making their integration rather straightforward. Actually, Spi2Java comes with a tool, called *Spi2Proverif*, that translates specifications from the Spi2Java syntax to the syntax accepted by ProVerif.

When translating to ProVerif syntax, Spi2Proverif adds selected information from the eSpi document, in order to make the analysis more precise. For example, let us consider the following final hash that appears in the client spi calculus model:

```
H((ID_C, ID_S, KEX_C, KEX_S, PubKey_s, DHPub(x), f, DHKey(x,f)))
```

The eSpi document includes for this term the following element (reported here with minor modifications to improve readability):

```
<term id="399"
      name="H((ID_C, ID_S, KEX_C, KEX_S,
               PubKey_s, DHPub(x), f, DHKey(x,f)))"
      type="Cryptographic Hashing">
  <codify>CryptoHashingSR</codify>
  <parameters>
    <param name="algorithm" type="var">DHHash</param>
    <param name="provider" type="const">SUN</param>
  </parameters>
</term>
```

The `id` attribute uniquely identifies the term in the XML document, while `name` is its human readable form. The `type` attribute specifies the Java type assigned to the term, and the `codify` element specifies the name of the Java class that implements the marshaling layer. The `parameters` element instead is the part where cryptographic algorithms and parameters are specified. For this term, the `algorithm` parameter is of variable (`var`) type, meaning it will be resolved at run-time: `DHHash` is the spi calculus term whose content (e.g. SHA-1, or SHA-256) will be the negotiated algorithm for the final hash. Conversely, the

`provider` parameter, indicating the Java Cryptographic Architecture (JCA) provider to be used, is of constant (`const`) type, meaning its value is assigned at compile-time. This value is `SUN` in this example.

Using this information taken from the eSpi document, Spi2Proverif translates the final hash into the following term:

```
H((ID_C, ID_S, KEX_C, KEX_S, PubKey_s, DHPub(x), f, DHKey(x,f)
  ), DHHash)
```

This is a hash function taking two parameters: the first one is the data to be hashed, while the second one specifies the hashing algorithm according to the eSpi document. This model captures the ideal property that hash values obtained using different algorithms are unrelated. It can be shown that this model is equivalent to the more common pattern where different hash functions are used for different algorithms. Indeed, suppose there exist two hash functions $Hsha1(\cdot)$ and $Hsha256(\cdot)$ that implement the SHA-1 and SHA-256 algorithms respectively, and that are used to hash terms m and n. Since they are different symbolic functions (and no equational theories link them), $Hsha1(m) \neq Hsha256(n)$ for any value of m and n. In the same way, suppose there exist two symbolically different constants $Sha1$ and $Sha256$, then $H(m, Sha1) \neq H(n, Sha256)$ for any value of m and n, because the two hashes will always differ for their second argument.

Note that this approach allows each cryptographic operation to use independent and custom algorithms and parameters, either negotiated at run-time or known at compile-time. As already remarked, marshaling functions that map data to/from serial binary streams are not represented in the model that is analyzed by ProVerif, and correctness of their implementation is assumed.

The algebraic properties of cryptographic operations are included as equations in the generated ProVerif model. For example, ProVerif supports DH modular exponentiation by the two functions `DHPub` and `DHKey` used in the spi calculus model. These functions are modeled in ProVerif by the equation $DHKey(x, DHPub(y)) = DHKey(y, DHPub(x))$ (Blanchet et al., 2008).

The original spi calculus model only includes protocol instances and actors specifications, while the intended security properties that should be satisfied by the model have to be specified separately. In the approach described here, security properties have been specified as security queries in ProVerif syntax and added to the ProVerif model generated by Spi2Proverif. An equivalent option would be to allow the addition of properties in ProVerif style directly in the spi calculus model file with automatic translation to ProVerif along with the model.

When automatically translated into the ProVerif syntax by the Spi2Proverif tool, the SSH-TLP model discussed in the previous section numbers 431 lines. It was formally verified for secrecy of the DH shared secret and for injective server authentication by agreement on session data. Verification of all the security properties took less than 2.5s, on a Linux host with an Intel Core2 Quad Q9450 CPU running at 2.66GHz, with 8GB of RAM.

The secrecy-related queries are quite simple. Let `x` be the client DH secret, and `y` be the server DH secret. On the client side, the shared DH secret is computed as `DHKey(x,DHPub(y))` (where the client only knows `DHPub(y)`, and not `y`). So, the ProVerif query

```
query attacker:DHKey(x,DHPub(y)).
```

asks ProVerif to check whether the shared DH secret, as computed by the client, remains secret indeed. In order to speed up verification, two secrecy assumptions are made, namely that both `x` and `y` are not known to the attacker. ProVerif verifies the secrecy assumptions on the model, before proceeding with the formal verification of the properties.

On the server side, the shared DH secret is computed as `DHKey(y,DHPub(x))` (again, the server only knows `DHPub(x)`, and not `x`). Technically, the client-side query above is enough to prove secrecy because, due to the equational system, also the server leaking its own copy of the shared DH secret would be recognized. Nevertheless, it does not require much effort (nor a significant amount of verification time) to add a server-side secrecy query.

The server authentication property is expressed in ProVerif by means of injective agreement on relevant session data (Blanchet, 2002). Briefly, injective agreement means that each time client C believes it has finished a protocol session with server S agreeing on some data M, server S started a protocol session with client C, agreeing on the same data M. The work by Lowe (1997) gives more details on injective agreement.

Two agreement events are defined: `beginServerEnd(x)`, emitted by the server when it believes it has started a session with the client, agreeing on data `x`; and `endServerEnd(x)`, emitted by the client when it believes it has finished a session with the server, agreeing on some data `x`. Then, injective agreement is expressed by the following ProVerif query:

```
query evinj:endServerEnd(x) ==> evinj:beginServerEnd(x).
```

In SSH-TLP, the server emits the `beginServerEnd` event just before sending message 6 of the typical scenario in figure 2. Since cryptographic algorithms and parameters are explicitly represented in the model and the negotiation algorithm is explicitly modeled too, it is sufficient to only use the final hash as the `beginServerEnd` event argument. Indeed, the final hash depends on server and client identities, on all exchanged messages, on the shared DH secret and on the agreed algorithms (added by Spi2Proverif). The client instead emits the `endServerEnd` event on the same final hash after receiving message 6 of the typical scenario of figure 2, and only if the server signature has been successfully checked.

6. Model Refinement and Implementation Generation

Starting from a verified protocol model like the one discussed in the previous section, the Spi2Java tools can be used to semi-automatically derive a Java

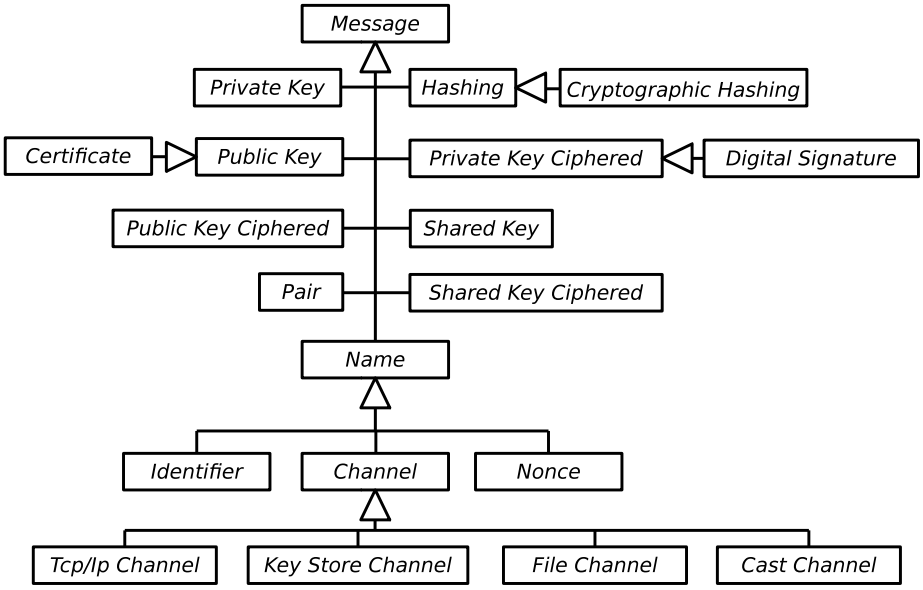

Figure 3: A significant subset of the default type hierarchy.

implementation of one or more roles of the protocol. Each role implementation is derived separately, using only that role's process specification. Auxiliary processes, such as the ones that model key stores, are implicitly implemented by a Java library provided by the Spi2Java framework, called SpiWrapper. Then, no code generation is needed for them.

For each spi calculus term belonging to the abstract model, three kinds of refinement information must be added in order to derive the Java implementation: a type; a set of cryptographic algorithms and parameters; and an encoding layer.

This section explains how the refinement and implementation steps were undertaken for the client-side of the SSH-TLP protocol, starting from the formal model discussed in section 4.

6.1. Assigning Types and Cryptographic Algorithms and Parameters

The Spi2Java framework offers an extensible type hierarchy, where "Message" is the top type, and subtypes are defined, based on term usage. For example, the "Pair" type is defined to represent a term used as a pair of data items in the spi calculus specification. A significant subset of the default type hierarchy is shown in figure 3. Each type is implemented by a Java class belonging to the SpiWrapper library. For example, the Pair class of the SpiWrapper library offers the `getLeft()` and `getRight()` methods, allowing the spi calculus pair splitting process to be implemented in Java.

Starting from the spi calculus specification, Spi2Java initially generates a default eSpi document that assigns default types, cryptographic algorithms,

parameters, and encoding layers to each term. Default types are inferred from term usage. When no useful information can be inferred, the most general type (Message) is assigned. Default cryptographic algorithms and parameters are pre-defined and a general default marshaling layer based on Java serialization is available in the library and assigned by default to each term. This initial eSpi document can be used for fast preliminary prototyping and early error detection: using this automatically generated document and the spi calculus specification, Spi2Java can generate a Java implementation without user interaction, which can be executed to test the logic of the protocol before proceeding with the final refinement. Of course, this preliminary prototype is not yet an interoperable implementation, because the default choices do not match the ones defined by the SSH standard for the protocol, but protocol roles generated in this way can interoperate with each other.

In the SSH-TLP client model, most data types were automatically inferred by Spi2Java and manual type refinement was necessary just for a couple of cases. The first adjustment required was the downcast to "Nonce" of some fresh names implicitly used as nonces but not automatically recognized as such by Spi2Java. The second adjustment required was to downcast channels from the general "Channel" type to more specific types. For example, `cAB` was cast to "TCP/IP Channel", while the private channel for interaction with the key store `ks` was cast to "Key Store Channel".

Fixed cryptographic algorithms and related parameters were also manually set according to the SSH-TLP RFC.

In the model, as explained in section 4, the algorithm negotiation procedure was abstractly modeled by means of one-way functions. Now, a real implementation of that procedure must be provided. Thanks to the extensible type system, two sub-types of the "Hashing" type have been defined, namely "Extract Alg" and "Extract Param". The former type is the function that computes the result of negotiation for one algorithm. Its argument is a pair of lists: the client-preferred list of algorithms, and the server-preferred list of algorithms. The result of the function is the agreed algorithm.

Likewise, "Extract Param" includes the function that computes the cryptographic parameters from the negotiated algorithms. Its argument is a pair composed of an (agreed) algorithm and an integer marker. The result of the function is the value of the cryptographic parameter identified by the marker, to be used with the given algorithm.

Note that this modeling strategy is generic and modular: provided the algorithm negotiation procedure can be modeled by a one-way function in the first place (discussed in section 4), switching the algorithm negotiation procedure amounts to only updating the implementation of the "Extract Alg" and "Extract Param" types, while the abstract model remains unchanged.

6.2. Encoding Layer

The last kind of refinement information that must be provided in order to get an interoperable implementation concerns the encoding layer, i.e. the layer

```
1: /* check sign_47 of H([...]) with PubKey_s_49 in */
   [...] Declaration of required variables [...]
2: if (sign_47.check(final_hash, PubKey_s_49,
3:                       signKeyType_43.getText(),
4:                       signHash_41.getText(),
5:                       "SUN; SunRsaSign")) {
```

Figure 4: Java code excerpt implementing the spi calculus signature check.

that manages marshaling operations. In the current Spi2Java version, this layer is not automatically generated, so it must be provided by the user.

For the case study presented here, an encoding layer implementing the conversions between Java types and the SSH-TLP binary packet protocol (Ylonen and Lonvick, 2006b) and SSH-TLP specific types representations (Ylonen and Lonvick, 2006a) was implemented. The implementation work was quite straightforward, although the required development time was not negligible compared to the modeling or refinement steps.

Once the encoding layer was available, the eSpi document was updated. Each term that required the custom SSH-TLP encoding layer had the corresponding `codify` element modified to point to the proper Java class implementing the SSH-TLP encoding and decoding for that term. Note that some terms that are never sent over channels (such as, for example, some constants or the channels themselves) do not need to use any encoding layer, and the default one can be safely left.

6.3. Implementation Generation and Execution

When the spi calculus model and the eSpi document containing refinement information are ready, the Spi2Java code generator can be used to automatically generate the Java code implementing the protocol logic, and the whole application skeleton. The translation function from spi calculus to Java was proved sound under a Dolev-Yao attacker by Pironti and Sisto (2010).

In order to show what the generated protocol code looks like, the Java excerpt corresponding to the signature check in the client spi calculus model is reported in figure 4. Line 1 contains an automatically generated comment that improves readability of the generated code, by reporting the original translated spi calculus process (term identifiers are appended a numerical suffix to make them unique). Note that the Java `final_hash` identifier appearing at line 2 is a shorthand for a longer variable name actually used by the code generator, which is obtained by mangling the `H([...])` spi calculus term.

The Java `sign_47` variable appearing at line 2 stores the server digital signature and has type "Digital Signature". This type offers the method

```
boolean check(Message msg, PublicKey pubK,
          String algorithm, String digest, String provider)
```

that returns `true` if the signature in the object is a valid signature of message `msg` under key `pubK`, and returns `false` otherwise. The `algorithm`, `digest` and `provider` formal parameters are Java strings containing the name of the cryptographic algorithms and parameters to be used when performing the signature check. The first two of them are to be resolved at run-time, so in the actual parameters the `getText()` method is invoked, which returns the negotiated algorithm. The last parameter is known at compile time, so its value is directly set by the code generator.

The generated code is surrounded by some template context code, which handles exceptions and disposable resources such as channels. The context also collects into a map the protocol data that should be returned after a successful protocol execution: for example, the agreed shared secret is to be returned after a successful run of SSH-TLP. The skeleton of the generated method looks like

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

The `@InputParams@` placeholder is substituted by the free variables and free names of the translated spi calculus process. `@DisposableResourcesDeclaration@` is substituted by the Java declaration of the variables holding references to the disposable objects. `@GeneratedSpiImpl@` is substituted by the generated Java code implementing the core spi calculus process being implemented, and the `@CloseDisposableResources@` placeholder is substituted by the code closing disposable resources that shall not be returned by the caller, that is whose reference is not stored within the `_return` map. Alternatively, the disposable resources could be handled by the *try-with-resources* Java statement introduced with Java version 7.

The `SpiWrapperException` class extends the standard Java `Exception` class, and captures all subtypes of exceptions that can be thrown by the SpiWrapper library. Since the generated code only uses the SpiWrapper library, it is sufficient to catch this exception to handle execution errors such as failed decryptions or equality tests (for which no else branch has been specified). Other runtime exceptions or errors could be thrown: as the applications should not normally try to catch such exceptions, the skeleton code just lets the Java Virtual Machine handle them.

Note that the caller of the generated method distinguishes between successful execution of the protocol and failure by the returned value: `null` means the protocol run failed, while a (possibly empty) map of returned values implies successful execution.

In addition to the generated function that implements a protocol session, the generated code includes the skeleton of a client or multi-threaded server

application (depending on the configuration of the code generator) where the call to the protocol session function is included. This skeleton may be taken as the basis for developing the final application or as an example for integrating a call to the protocol function into an existing application. The input arguments to pass to the function when it is invoked to start a new protocol session must be provided by the programmer, and are not included in the skeleton, because they cannot be automatically inferred from the model. It should be noted that, the possibility to set these parameters is essential in order to enable the application to flexibly configure each protocol session. For example, in the SSH-TLP client that was developed, this is exploited in each session to ask the user the server address and port to connect to.

7. Considerations about correctness

The MDD approach that has been presented here enhances confidence about the correctness of the implementation developed, compared to the usual level that can be obtained by manual development. The abstract model captures protocol logic thoroughly. Its formal verification, by proving that the desired security properties are actually fulfilled, rules out the main errors that could be made when writing protocol logic.

The problem of preserving the correctness properties proved on the model down to the generated Java implementation was addressed by Pironti and Sisto (2010). In that work, the translation function from spi calculus to Java used by Spi2Java was formally defined and a pencil-and-paper proof of its soundness was provided. Essentially, soundness was shown by proving that the generated Java program is a refinement of the original spi calculus model. This means that all possible behaviors of the generated Java program are a subset of the behaviors described in the spi calculus model. Thus, if the spi calculus model has been verified as secure (that is, all its behaviors are safe), then the Java program is implied to be secure too, because it will exhibit a subset of the verified behaviors. More precisely, a weak simulation relation between the generated Java program and the original spi calculus model was proved to exist by Pironti and Sisto (2010), showing that each step that a generated Java program can execute is mapped to a step that the original spi calculus model can execute, which implies the aforementioned refinement relation.

Java exceptions are fully handled by this refinement relation, because they do not break any safety property (such as secrecy or authentication) that holds on the verified model, and is implied in the generated code. Assume a formally verified spi calculus model, and its generated Java code. When an exception is thrown in Java, control possibly passes to the `catch` block (where the map of returned values is set to `null`) and ultimately execution of the protocol run is stopped at the point where the exception was thrown. Intuitively, since the generated Java code is a refinement of the spi calculus model, it means that the Java behavior up to the point before the exception was thrown was mapped onto a corresponding safe behavior in spi calculus. When the exception is thrown, essentially Java execution stops (as far as the visible behavior of the security

protocol is concerned), which is always a safe behavior. More formal details about soundness of exception handling can be found in the work by Pironti and Sisto (2010).

This soundness result relies on the correctness of the SpiWrapper library. Pironti and Sisto (2010) give a rigorous model of the intended semantics of this library, and shows preliminary work on how to verify that a possible implementation of this library fulfills the specified semantics. For example, a paper-and-pencil proof that the Pair class implementation provided with the Spi2Java framework is formally correct with respect to its intended semantics is given.

In this library, equality of objects is always tested via the `equals` method, and not by reference equivalence. This is required because 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 Java object for the M term, and one different Java object for the received x term. It may be the case, however, that x is assigned the same value as 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 the value is the same in the two objects. 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 discover if their representing singleton is already instantiated or not.

Problems like pointer aliasing in Java or cyclic pointer structures are avoided instead by letting each object that belongs to the SpiWrapper library classes to be initialized exactly once, atomically at object creation time (in practice they are declared as `final`, all fields of the classes are final, and no setter methods are available). This constraint on the SpiWrapper objects does not limit expressiveness at all, because spi calculus terms obey to the same semantics.

What this approach does not guarantee is the correctness of the encoding layer, made up of the marshaling functions used to translate data from/to their serial binary representations. Since these functions are developed manually, they could be incorrect. However, it has been shown by Pironti and Sisto (2008); Pironti (2010) that any implementation of such functions that fulfills some functional correctness properties cannot introduce other security flaws.

These marshaling functions can be divided into two categories: i) the functions encoding data to be sent/received over the network, and ii) the functions preparing data to be used in cryptographic primitives (e.g. a key derivation function that builds an AES key out of some raw key material). Pironti (2010) showed that a difference in criticality exists between functions of the two types.

Functions of type i) only deal with data that are already secured, and ready to be sent over the network (they just need to be translated into a different format). These data transformation functions could even be implemented by an

untrusted entity, without affecting the protocol security. In fact, no particular properties are required of these functions, except an information flow stating that they should not access any protocol secret data item that is not available to the attacker. In this way, neither can secret data be directly leaked by these functions, nor can they be wrongly used to decrypt some already encrypted data.

Functions of type ii) are instead more critical, because they will possibly handle security relevant data. For example, consider a wrong key derivation function that always generates the same key, regardless of the key material. For such function implementations to be safe much stricter conditions are required. The weakest conditions found by Pironti (2010) state that the mathematical definition of such functions should be injective, and that the implementations should correctly implement the mathematical definition. Furthermore, in order to avoid interactions between different functions that might share the same encodings, it is necessary that all protocol actors agree on the same encoding algorithms. However, this last requirement is often checked as part of the security protocol formal verification task.

It is important to note that while errors in the encoding layer can likely be caught by extensive testing, the same is not true for the protocol logic, because of the huge number of scenarios that concurrent protocol sessions and unconstrained attacker behavior may introduce.

Another delicate point is the development of further algorithms extending the SpiWrapper class hierarchy. For example, the SSH-TLP case study required such an extension to the Hash class to handle algorithm negotiation as hash functions. A wrong implementation of such algorithms could lead to a Java implementation that is overall unsound. On the positive side, the work by Pironti and Sisto (2010) rigorously define the behavior that any implementation of such interface should have. So, any extension to the SpiWrapper library is safe, as long as it can be proved that it fulfills the given interface. Unfortunately, although formal verification of sequential, non-networked Java code has been tackled in the literature (e.g. Beckert et al. (2007); Marché et al. (2004); Barthe et al. (2007)), the current version of the Spi2Java framework does not offer a straightforward way to prove correctness of arbitrary implementations of the SpiWrapper library.

8. Experimental Results

8.1. Interoperability with Third Party Servers and Reliability

The generated SSH-TLP client was tested against seven third party server implementations; five kinds of sessions were executed with each server, totalizing 35 experiments. Since the negotiation procedure is driven by client-preferred algorithms, in each kind of session the client lists of preferred algorithms were properly configured, so that a different algorithm would be agreed.

Table 3 shows, for each kind of session, the significant lists of preferred algorithms that the client sent to the server.

Table 3: Lists of preferred algorithms.

Kind of session	Signature	DH group	Final Hash
1	RSA; DSA	1; 14	SHA-1
2	RSA; DSA	14; 1	SHA-1
3	DSA; RSA	1; 14	SHA-1
4	RSA; DSA	1	SHA-1
5	RSA; DSA	14	SHA-1

Table 4: Tested servers.

Server Name	Server ID string	OS	Comments
dropbear 0.52	dropbear_0.52	Linux	No DH G14
freeSSHD 1.2.6	WeOnlyDo 2.1.3	Windows	All correct
KpyM 1.18	cryptlib	Windows	No DH G14 – RSA only
lsh 2.0.4	lshd-2.0.4 lsh - a GNU ssh	Linux	RSA only
MobaSSH 1.12	OpenSSH_5.1	Windows	All correct
OpenSSH 5.5	OpenSSH_5.5p1	Linux	All correct
WinSSHD 5.15	1.04 FlowSsh: WinSSHD 5.15	Windows	All correct

In sessions of kinds 1, 2, and 3, the client sent to the server a list with all the algorithms that the SSH-TLP requires to be supported by the actors. If the server supported at least one of the algorithms proposed by the client, then the negotiation algorithm was expected to terminate with success. When the server supported both RSA and DSA keys, sessions of kinds 1, 2 and 3 ended well. With servers only supporting RSA keys, all the three kinds of sessions were expected to end correctly, but always with the final agreement on the RSA signature scheme.

In sessions of kinds 4 and 5, for the "DH group", the client sent the server a list with only one group. The negotiation algorithm was expected to fail if the server did not exactly support the client requested group, otherwise the session was expected to end well.

The third party servers used for testing are reported, with some comments, in table 4. The "Server Name" column reports the advertised name of the server application, while the "Server ID string" column reports the identification string (ID_S) sent by the server. The "OS" column specifies the operating system which the server ran on.

Under the "Comments" column, servers with the comment "All correct" support all required algorithms, and all kinds of sessions ended as expected. In particular, the negotiated algorithms were always the preferred client algorithms.

Servers with the comment "RSA only" only support RSA keys. In this case, sessions of kind 3 correctly terminated by agreeing on the RSA signature scheme.

Servers with the comment "No DH G14" only support one of the two re-

Table 5: Measures of the generated client application.

Package	TLOC	MLOC	Ca
spiWrapper	3768	2180	102
spiWrapperSSH	2564	1136	5
sshClient	648	565	0

TLOC: Total Lines of Code: non-blank and non-comment lines in a class.
MLOC: Method Lines of Code: non-blank and non-comment lines inside method bodies of a class.
Ca: Afferent Coupling: number of classes outside a package that depend on classes inside the package.

quested DH groups, namely group 1. With these servers, sessions of kinds 1, 3 and 4 ended as expected. Sessions of kind 2 ended correctly, being the DH group 1 agreed. Sessions of kind 5 correctly failed instead, because it was impossible to agree on a DH group.

The experiments illustrated here show in fact that the generated client can execute in real environments, because it correctly interoperates with third party implementation servers.

Moreover, the client was tested against the SSHredder[1] suite, composed of more than 660 incorrect protocol sessions. Each incorrect session was correctly rejected by the client, which confirms the reliability of the code developed with the proposed methodology.

8.2. Code Metrics

Some measures of the client code generated with Spi2Java are reported in table 5. The spiWrapper package contains the SpiWrapper library, implementing the low-level cryptographic and network operations. The spiWrapperSSH package contains the manual implementation of the SSH-TLP specific encoding layer, and the sshClient package contains the automatically generated protocol logic, a default main application class and some customizable stub code to be run after a successful protocol run. It can be argued that *TLOC* and *MLOC* metrics highly depend on coding style. This is true; however, all the packages were written with the same coding style and rules. Because of this, it can be assumed that, in this particular context, both *TLOC* and *MLOC* are significant.

The automatically generated code (sshClient package), which mainly consists of the protocol logic implementation, is small compared to the rest of the application. Note, however, that the protocol logic is the most security critical part because it coordinates cryptographic and input/output operations, algorithm negotiation, key store interactions and handling of error conditions.

The library reusable code (spiWrapper package) accounts for more than half of the application, and the manually written code implementing the encoding layer (spiWrapperSSH package) completes the application. In practice this means that when designing and developing a real application with the proposed

[1] http://www.rapid7.com/sshredder.zip

Table 6: Qualitative aspects of SSH-TLP client applications.

Application Name	Multi Threaded	Message loop/ Imperative
Spi2Java Client	No	Imperative
Jsch	No	Imperative
J2SSH	Yes	Mixed
Jaramiko	Yes	Message loop

methodology, a significant amount of time is required to develop the encoding layer. Nevertheless, the implementation work is quite straightforward for this layer, and the developers need only concentrate on the specific aspect of encoding, which can make development more efficient.

The *Ca* metric (afferent coupling) measures code dependencies. It is defined as the number of classes outside a package that depend on classes inside the package. As expected, the "top-level" package sshClient is not required by any other package, because it only contains the protocol logic, coordinating the functions offered by the underlying libraries. The encoding layer package (spiWrapperSSH) is used by the classes within the sshClient package, and the spiWrapper library package is required by all other packages, since the protocol logic requires it to perform the cryptographic and network operations, and the encoding layer requires it to compute the internal representation of data. The analysis of this metric leads to the conclusion that the proposed methodology also helps to create structured applications, improving application maintainability and code reuse. In particular, since most of the code is a reusable library, this library can be extensively tested and verified in order to increase the assurance level of the code developed by this methodology.

8.3. Comparison with Third Party Clients

The Spi2Java generated client implementation was compared with three third party SSH-TLP clients manually developed in Java, namely Jsch, J2SSH and Jaramiko. The comparison was made considering qualitative and quantitative aspects of the code, and performance.

The qualitative aspects that were compared are shown in table 6: whether the implementation is multi-threaded or not; and whether the protocol logic implementation is written using a message-loop or imperative paradigm.

As it will be shown when presenting the results about performance, a multi-threaded approach does not improve execution performance (and it would make formal verification harder). Indeed, threading is mostly used as an overall timeout feature, to mitigate denial of service (DoS) attacks: if the protocol logic thread does not join within the specified timeout, it gets killed and an error is reported. The Spi2Java framework does not offer the same overall timeout feature, however each channel input/output operation can have a timeout.

Using an imperative paradigm to handle the SSH-TLP logic is viable, because the protocol consists of few message exchanges with few branching, and

error handling is completed by sending one error message. Furthermore, the imperative paradigm has the advantage of implicitly defining the protocol state machine, so reducing the possibility of accepting an otherwise valid message in the wrong state. Conversely, a message loop paradigm is more flexible and scales up better for more complex protocol logics. However, it requires careful design and implementation of the protocol state machine. Although manual code inspection of the Jaramiko application did not reveal any security flaw, the state machine implementation is not evident in the code, which increases the risk of security flaws.

A quantitative analysis of the implementation codes under comparison was also performed, in order to evaluate the impact of the Spi2Java development method on code length. Regarding this analysis, it is important to note that accuracy cannot be very high, because each application has different code structure and style, which may impact on code length. However, for the final aim of this analysis, which was just to show that the code developed with Spi2Java is not very different in length than the manually written codes, high accuracy is not needed and the noise introduced by different code styles is acceptable.

Gathering uniform and significant quantitative metrics for all the implementation codes turned out to be non-trivial: on the one hand, the manually developed applications contain the implementation of the full SSH protocol stack (which includes the SSH-TLP), so it was necessary to identify and exclude the unrelated code; on the other hand, none of the third party implementations supports the RFC-required DH group 14 key exchange method, which instead is included in the code developed with Spi2Java.

Because of the difficulty to identify relevant code in a uniform way in the applications, the following procedure has been used for each implementation (the one developed with Spi2Java and the third party ones): (i) profiling of a typical SSH-TLP session (with the OpenSSL server, using group 1 and RSA) to obtain the set of Java classes actually used in that session; (ii) gathering of the selected metrics (TLOC and MLOC) for these Java classes. This procedure slightly underestimates overall code length because Java interfaces, exception classes and error message handling classes are not counted, but this is done uniformly on all the implementations.

The gathered measures are reported in table 7. As the marshalling code is not easy to distinguish from the rest of the code in the third party implementations, the table just distinguishes the whole code and the code that implements the protocol logic.

All the applications share a comparable overall size. Jsch is slightly smaller in terms of TLOC, but comparable in terms of MLOC, which is justified by different coding styles. Although in the Spi2Java generated application only half of the code was manually developed (the remainder being either a reusable library or automatically generated), it does not follow that development time with Spi2Java was half of a typical manually developed application. Indeed, spi calculus model design, refinement and formal verification compensate the effort, but also give stronger security guarantees with respect to manual development.

Table 7: Measures of third party client applications.

Application Name	TLOC	MLOC	TLOC of protocol logic	MLOC of protocol logic
Spi2Java Client	5883	3221	382	342
Jsch	4220	3224	1543	1324
J2SSH	5843	3397	2571	1526
Jaramiko	5333	3183	1381	921

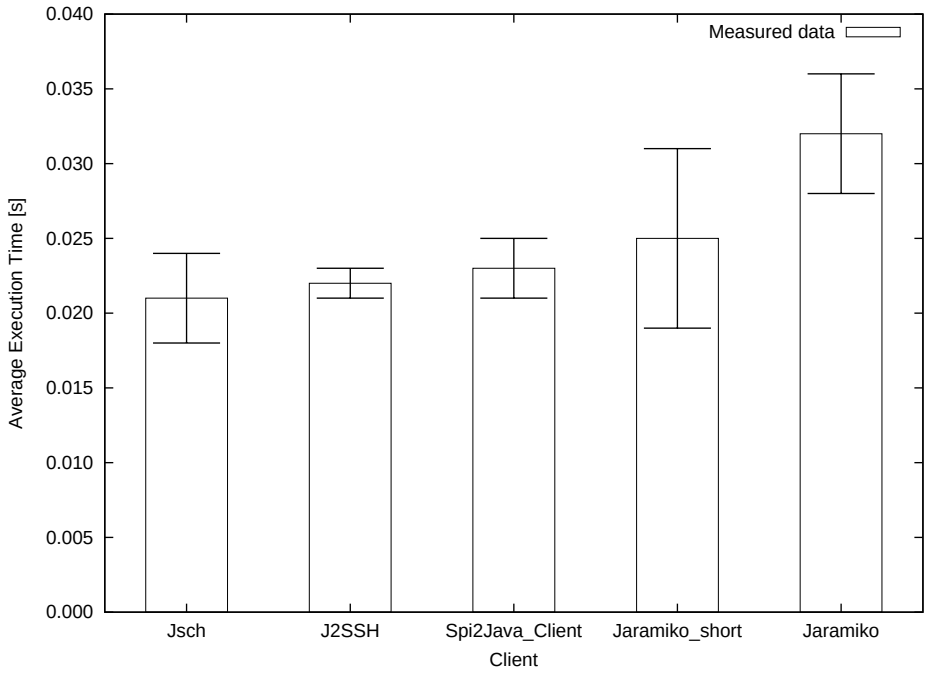

Figure 5: Average execution times of different SSH-TLP clients.

The analysis of the TLOC and MLOC metrics for the parts of code that were identified as protocol logic shows that, compared to the Spi2Java generated implementation, the manually developed ones are larger, and so possibly more complex to analyze for their security properties. It is worth to note that this difference is partly due to the fact that the separation between protocol logic and rest of the application is not as strict in third party implementations as it is in code developed with Spi2Java. For example, code inspection revealed that some aspects related to data encoding are mixed with protocol logic in third party implementations.

Finally, execution times of the generated SSH-TLP client and of the selected third party SSH-TLP clients were measured. For each client, 1000 runs of the protocol were executed with the OpenSSH 5.1p1 server. Average execution times and average deviations are reported in figure 5. All the runs were executed

on the aforementioned Linux machine under constant system load and over the localhost, in order to avoid random network delays; the same cryptographic algorithms were always negotiated, namely DH group 1 with RSA server key. In order to eliminate the latency introduced by the Java class loader, for each client the first run was not considered, and all subsequent runs were all executed in the same virtual machine instance. For all clients, the measured time starts at the beginning of the protocol, including TCP handshake and parsing of the key store file, and stops after checking the digital signature over the final hash and the server key.

All the clients, including the one developed in this paper, exhibited comparable execution times, with the exception of the Jaramiko one. In fact, it turned out that the DH modular exponentiation operation is the most computationally intensive one, making all other operations (including other cryptographic operations such as signature check) negligible. In particular, execution time depends on the size, in bits, of the chosen DH secret (`x` in the spi calculus model). Since all clients except the Jaramiko one use the standard JCA implementation to choose the DH secret, it follows that they all executed in comparable time. The Jaramiko client instead uses a larger DH secret on average, making it slower than other clients. This is further confirmed by the fact that after patching the Jaramiko client to generate DH secrets similar in length to the ones generated by the standard JCA implementation (the Jaramiko_short client in the chart), it executed in times comparable to the ones of the other clients.

It is then possible to conclude that the semi-automatically generated SSH-TLP client presented in this paper can in fact replace other manually developed implementations without significant performance penalties.

9. Conclusion

The case study illustrated in this paper gives evidence of the viability of the Spi2Java-based MDD approach for security protocols, with a real and widely used open protocol at hand. By following the MDD methodology implied by the Spi2Java framework, an abstract model of the SSH-TLP was first developed. This model was then automatically proved to fulfill the expected secrecy and server authentication properties by means of the ProVerif tool. Finally, the client-side part of the model was semi-automatically and soundly refined by means of Spi2Java into a Java application.

This development chain leads to an implementation that is formally guaranteed to follow a protocol logic that fulfills the expected security properties under the Dolev-Yao assumptions. This yields an overall higher assurance that the protocol logic has been implemented without errors, in comparison to what can be obtained by traditional manual development.

In this paper it was shown experimentally that this added value does not come with the cost of compromising other relevant implementation features.

The interoperability of the client-side application that was semi-automatically derived from the model was experimentally verified by performing interoperability testing against several third party server implementations.

The reliability and robustness of the generated application were confirmed by the correct handling and rejection of over 660 maliciously crafted protocol sessions.

The execution time of the model-driven implementation is comparable with that of other third party Java SSH-TLP clients implemented manually. Indeed, most of the implementation complexity lies in the computation of cryptographic primitives, rather than in the protocol logic. This means that the derived application can practically replace other implementations.

Finally, the measures on the size and coupling of the implementation code show that the generated application is quite modular. The protocol logic, the low-level cryptographic and network operations, and the encoding layer are encapsulated in different packages, each containing automatically generated code, library code, and manually developed code respectively. Although the manually developed code accounts for almost half of the whole application code, it only deals with the encoding layer, while the most critical code (the protocol logic implementation) is either automatically generated or part of the SpiWrapper library, shipped with the Spi2Java framework.

In order to achieve an interoperable Java implementation, the developed abstract model takes all the mandatory protocol features into account, including algorithm negotiation and error handling. Moreover, the abstract model explicitly represents implementation-related aspects such as the interaction with a local key store, which are generally left out when formal security protocol models are just used to verify the main protocol logic. To our knowledge, this is the first verified formal specification of SSH-TLP fully modeling all these features. Nevertheless, the model was still left abstract enough, allowing different implementation choices. For example, the key store interaction is generically modeled by a request/response paradigm, allowing applications to be backed by different key store implementations. It is plausible that the same modeling strategies can be reused in other protocol models too.

As far as concerns the development effort required, our experience was that, for an experienced user, writing the formal abstract model in spi calculus has development costs similar to writing the protocol logic in a programming language. Nevertheless, we expect that this step would require more effort for an average developer not already familiar with spi calculus.

In all cases, an extra cost is accrued in the Spi2Java MDD approach for formally verifying the model using ProVerif. The time taken by this verification step was relevant in our experience, because ProVerif seems to be rather sensitive to how the model is written, especially when local channels are extensively used. Because of this problem, several variants of the model were tried out before finding one that could be automatically verified by ProVerif.

A current limitation of the Spi2Java MDD approach is that the encoding layer has to be manually developed. This takes a significant amount of time in the whole application development, and the manually written code is currently assumed, and not verified, to correctly implement the specified encoding layer. Moreover, this code is potentially subject to code vulnerabilities, which could be avoided by automatically generating this code too.

Hence, as a future work, automatic generation of the encoding layer, starting from a mathematical model of encoding and decoding functions, could both reduce development effort and increase the overall assurance level of the generated application, by providing a fully verified implementation. To some extent, this is already possible: some security protocols, such as Kerberos, use ASN.1 encodings, for which automatic code generation for encoding and decoding functions already exists.

Another limitation of the Spi2Java MDD approach is the need for the developer to know the spi calculus modeling language. This problem could be alleviated by adopting either a graphical modeling language, or textual languages more similar to the programming languages commonly adopted by programmers.

Acknowledgments

We would like to thank Bruno Blanchet for his valuable support in the verification phase of our case study.

References

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

B. Blanchet, D. Pointcheval, Automated Security Proofs with Sequences of Games, in: International Cryptology Conference, LNCS 4117, Springer, 537–554, 2006.

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.

S. Escobar, C. Meadows, J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, in: Foundations of Security Analysis and Design, LNCS 5705, Springer, 1–50, 2009.

G. Lowe, Casper: A Compiler for the Analysis of Security Protocols, Journal of Computer Security 6 (1-2) (1998) 53–84.

S. Mödersheim, L. Viganò, The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols, in: Foundations of Security Analysis and Design, LNCS 5705, Springer, 166–194, 2009.

T. Ylonen, C. Lonvick, The Secure Shell (SSH) Protocol Architecture, RFC 4251 (Proposed Standard), URL `http://www.ietf.org/rfc/rfc4251.txt`, 2006a.

T. Ylonen, C. Lonvick, The Secure Shell (SSH) Transport Layer Protocol, RFC 4253 (Proposed Standard), URL `http://www.ietf.org/rfc/rfc4253.txt`, 2006b.

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

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, IEEE Computer Society, 400–405, 2004.

B. Blanchet, A Computationally Sound Mechanized Prover for Security Protocols, IEEE Transactions on Dependable and Secure Computing 5 (4) (2008) 193–207.

A. Pironti, R. Sisto, An Experiment in Interoperable Cryptographic Protocol Implementation Using Automatic Code Generation, in: IEEE Symposium on Computers and Communications, IEEE Computer Society, 839–844, 2007.

A. Pironti, D. Pozza, R. Sisto, Resources about Spi2Java and the SSH-TLP Case Study., Online, URL `http://spi2java.polito.it`, 2011.

K. Bhargavan, C. Fournet, A. D. Gordon, S. Tse, Verified Interoperable Implementations of Security Protocols, in: Computer Security Foundations Workshop, IEEE Computer Society, 139–152, 2006a.

J. Jürjens, Automated Security Verification for Crypto Protocol Implementations: Verifying the Jessie Project, Electron. Notes Theor. Comput. Sci. 250 (2009) 123–136, ISSN 1571-0661, doi:10.1016/j.entcs.2009.08.009.

N. O'Shea, Using Elyjah to Analyse Java Implementations of Cryptographic Protocols, in: Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, 2008.

K. Bhargavan, C. Fournet, A. D. Gordon, Verified Reference Implementations of WS-Security Protocols, in: Web Services and Formal Methods, LNCS 4184, Springer, 88–106, 2006b.

K. Bhargavan, R. Corin, C. Fournet, E. Zălinescu, Cryptographically verified implementations for TLS, in: Computer and Communications Security, ACM, 459–468, 2008.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, S. Maffeis, Refinement types for secure implementations, ACM Transactions on Programming Languages and Systems 33 (2011) 1–45.

K. Bhargavan, C. Fournet, A. D. Gordon, Modular verification of security protocol code by typing, in: Symposium on Principles of Programming Languages, ACM, 445–456, 2010.

E. Bangerter, J. Camenisch, S. Krenn, A.-R. Sadeghi, T. Schneider, Automatic Generation of Sound Zero-Knowledge Protocols, Cryptology ePrint Archive, Report 2008/471, 2008.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, J. J. Leifer, Cryptographic Protocol Synthesis and Verification for Multiparty Sessions, in: Computer Security Foundations Symposium, IEEE Computer Society, 124–140, 2009.

E. Hubbers, M. Oostdijk, E. Poll, Implementing a Formally Verifiable Security Protocol in Java Card, in: Security in Pervasive Computing, LNCS 2802, Springer, 213–226, 2003.

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, IEEE Computer Society, 507–510, 2005.

S. Kiyomoto, H. Ota, T. Tanaka, A Security Protocol Compiler Generating C Source Codes, in: International Conference on Information Security and Assurance, IEEE Computer Society, 20–25, 2008.

D. X. Song, A. Perrig, D. Phan, AGVI - Automatic Generation, Verification, and Implementation of Security Protocols, in: International Conference on Computer Aided Verification, Springer, 241–245, 2001.

B. Tobler, A. Hutchison, Generating Network Security Protocol Implementations from Formal Specifications, in: Certification and Security in Inter-Organizational E-Services, Springer, Toulouse, France, 2004.

H. Grandy, M. Bischof, K. Stenzel, G. Schellhorn, W. Reif, Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code, in: Formal Methods, LNCS 5014, Springer, 165–180, 2008.

N. Moebius, K. Stenzel, H. Grandy, W. Reif, SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications, in: International Conference on Availability, Reliability and Security, IEEE Computer Society, 841–846, 2009.

D. Basin, J. Doser, T. Lodderstedt, Model Driven Security: from UML Models to Access Control Infrastructures, ACM Transactions on Software Engineering and Methodology 15 (1) (2006) 39–91.

J. Jürjens, Secure Systems Development with UML, Springer, 2005.

J. McDermott, Visual security protocol modeling, in: Workshop on New Security Paradigms, ACM, 97–109, 2005.

A. Pironti, R. Sisto, Provably correct Java implementations of Spi Calculus security protocols specifications, Computers & Security 29 (3) (2010) 302–314.

M. Abadi, A. D. Gordon, A Calculus for Cryptographic Protocols: The Spi Calculus, Research Report 149, 1998.

B. Blanchet, M. Abadi, C. Fournet, Automated Verification of Selected Equivalences for Security Protocols, Journal of Logic and Algebraic Programming 75 (1) (2008) 3–51.

A. Pironti, R. Sisto, Soundness Conditions for Message Encoding Abstractions in Formal Security Protocol Models, in: Availability, Reliability and Security, IEEE Computer Society, 72–79, 2008.

A. Pironti, Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications, Ph.D. thesis, Politecnico di Torino (Italy), URL `http://alfredo.pironti.eu/research/sites/default/files/Pironti_Dissertation.pdf`, 2010.

B. Blanchet, From Secrecy to Authenticity in Security Protocols, in: International Static Analysis Symposium, LNCS 2477, Springer, 342–359, 2002.

G. Lowe, A Hierarchy of Authentication Specifications, in: Computer Security Foundations Workshop, IEEE Computer Society, 31–43, 1997.

B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, LNCS 4334, Springer, 2007.

C. Marché, C. Paulin-Mohring, X. Urbain, The Krakatoa tool for certification of Java/Java Card programs annotated in JML, Journal of Logic and Algebraic Programming 58 (1-2) (2004) 89 – 106.

G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, A. Requet, JACK: a tool for validation of security and behaviour of Java applications, in: International Symposium on Formal Methods for Components and Objects, LNCS 5382, Springer, 152–174, 2007.