Black-Box Monitoring of Security Protocols, Revision 1

Alfredo Pironti
Politecnico di Torino
Turin, Italy
http://alfredo.pironti.eu/research/

Jan Jürjens
TU Dortmund and Fraunhofer ISST
Dortmund, Germany
http://jurjens.de/jan

***Abstract*—In the challenge of getting provably correct implementations of security protocols, much effort has been recently put into two strategies: model-driven-development to generate new implementations; and verification of the source code of already existing implementations. However, no approach currently deals with legacy implementations for which no source code is available. This paper presents a formal approach to design and implement monitors that stop insecure protocol runs executed by legacy implementations, without the need of their source code. We demonstrate the approach at a case study about monitoring a generic SSL server implementation. Our monitoring approach allowed us to detect a flaw in an SSL client implementation.**

***Keywords*-Formal Methods; Security Protocols; Application Monitoring; Model Driven Development**

I. Introduction

Security protocols are communication protocols that allow to coordinate distributed computation through untrusted channels, while protecting some assets; often cryptographic functions are leveraged in order to achieve these goals. For example, security protocols can be designed for online banking or e-commerce through the Internet, or to set-up trusted, confidential point-to-point communication on top of a network of untrusted relayers. Depending on the application, the desirable security goals vary so much, that different security protocols have been designed for different purposes. Despite the apparent simplicity of security protocols, they are quite difficult to get right, mainly for two reasons: the concurrent nature of the distributed environment, and the presence of an active attacker, that can behave in the worst case. In fact, several flaws have been documented both in the protocol specifications (e.g. [1], [2]), and in their implementations (e.g. [3]). Moreover, the cryptographic functions that security protocols rely on can be flawed too (e.g. [4]), increasing the chances that an attacker can successfully breach the desired security goals.

Increasing the confidence on the security protocol implementations correctness is then a worthwhile challenge, because it is one of the key factors that enable to increase reliability and dependability of software systems. Testing can mitigate the problem, but it can only stress specific scenarios, while the concurrent nature of the system and the active attacker generate a large (often unbounded) number of scenarios, thus making testing coverage usually quite low. Many different formal approaches have been also proposed to tackle the problem, at different abstraction levels. In this paper we concentrate on cryptographic protocols, rather than on cryptographic functions. That is, we assume cryptographic functions to be correct, and concentrate on their usage within the cryptographic protocols; this is often called a Dolev-Yao [5] approach, where perfect encryption is assumed, and the attacker is able to control the medium. Moreover, we concentrate on implementations of security protocol actors, rather than on the high level specifications of such security protocols. That is, we assume (and do not try to prove) that a given protocol specification satisfies its security requirements; instead we want to asses that a given implementation of one protocol's role is correct with respect to its specification, under the presence of a Dolev-Yao attacker. Note that we focus on assessing the correctness of legacy, already deployed implementations, rather than on the development of correct new ones. Indeed, there are several scenarios where a legacy implementation is already running, and cannot be substituted by a new one. For example, the legacy implementation of the security protocol may be strictly coupled with the rest of the information system, rising the cost of the switch; or the legacy implementation may be already certified or trusted by the user; or some policy may simply require not to change the legacy implementation.

Several attempts have already been made to check that a protocol role implementation is correct w.r.t. its specification, and they can be grouped in four main categories.

1) Model Driven Development (MDD)
2) Model Extraction
3) Refinement Types
4) Online Monitoring

The first approach consists in designing and verifying a formal, high-level model of a security protocol (or a security-aware application) and to semi-automatically generate an implementation that is proven to satisfy the security properties satisfied by the formal model [6]–[8]. This approach allows, during the design phase, the developer to concentrate only on the high-level protocol logic, rather than on implementation details; moreover it allows to get implementations of the security protocol that can be proven to be correct w.r.t. the given specification. Nevertheless, it has the drawback of not handling legacy implementations of security protocols: in already deployed software systems, the cost of switching from a legacy implementation to a new one may be high, much higher than coping with the security flaws of the existing implementation.

The second approach starts from the source code of a full blown, existing implementation, and extracts a formal model which is then verified for the desired security properties [9], [10]. This approach is then able to deal with legacy implementations, but it requires their source code to be available. Again, this may not be the case for legacy or proprietary systems. Moreover, it is often necessary to manually annotate the source code, which is an error prone and usually costly task.

The third approach performs a special kind of type checking on the implementation source code, so that it can be proven secure against the desired security properties [11]. It shares the same advantages and drawbacks of the second approach, so the (possibly annotated) source code is needed, which makes this approach difficult to apply on legacy or proprietary implementations.

The fourth approach consists in instrumenting the source code of an existing implementation with assertions that are checked at runtime, so that program execution can be stopped if any assertion fails. Again, this approach implies manual modification of the source code (when the source code is available at all), which is error prone, and requires to substitute the legacy implementation with the instrumented one, which may not always be possible.

In principle, a fifth approach that could handle legacy implementations may also be possible, that is formal verification of assembly code. Unfortunately, this approach is not viable in practice, because it is too difficult to abstract and identify the security relevant details from the flattened assembly code, making verification unfeasible.

In order to deal with legacy implementations of security protocols, this paper presents an approach based on black box monitoring of security protocols implementations. The overall methodology is depicted in figure 1. Given the protocol definition, a specification for one role is manually derived. By using the $a2m$ function defined in this paper, a monitor specification for that protocol role is automatically generated. Then the monitor implementation is obtained by using the MDD framework called spi2java [6]. The monitor application is finally ran together with the monitored protocol role implementation (not shown in the picture). The monitoring is black box, because the source code of the monitored application is not needed, only its observable behaviour (data transmitted over the medium, or *traces*) and locally accessed data are required. This implies that any legacy, already deployed implementation can still be used in production as is, while being monitored. If the monitor detects any deviating behaviour of the monitored legacy application, it can stop protocol execution, and report the error. The correctness of this approach depends on the correctness of the generated monitor for a given protocol role. Our approach leverages formal methods in the derivation of the monitor implementation, so that a provably correct monitor is obtained. Moreover, a soundness property of the monitor is also showed, meaning that, under the Dolev-Yao model, any faulty trace is recognized and stopped by the monitor, and only safe traces are allowed.

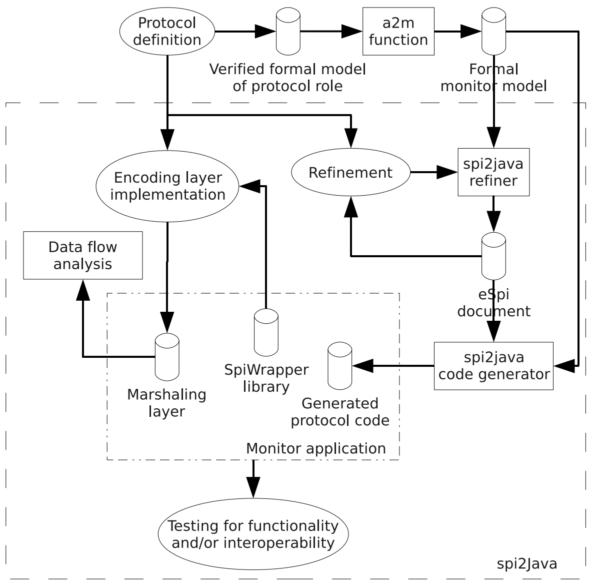

Figure 1. Monitor design and development methodology.

Note that this approach can be exploited during the testing phase too. Indeed, one can run several simulated protocol sessions in a testing environment, and let the monitor check for the correct behaviour of the protocol actors.

The rest of the paper is organized as follows. Section II illustrates the formal background used in the paper. Section III describes the function translating a Spi Calculus protocol agent's specification into a monitor specification for that agent. Then section IV shows a case study of the design, development and field-test of a black-box monitor for SSL server implementations. Finally section V concludes.

II. Formal Background

A security protocol role can usually be specified as a (sequential) process, performing input/output operations on channels, cryptographic operations and checks on received data. Often, security protocols specifications are given in an informal, human readable way, from which implementations are directly developed. Ambiguities in the informal specification can lead to different, incompatible implementations, or even flawed ones.

Since our goal is to deal with legacy implementations, as explained before it is not possible to use formal methods in order to check their correctness with respect to a formal, unambiguous specification, because source code may not be available, or annotations may be required in order to verify it.

Instead, with the approach proposed in this paper, one starts from the formal specification of the protocol role to be monitored, and automatically obtains the formal specification of the monitor for that role. Then, using an MDD approach, the monitor implementation can be semi-automatically generated from its specification.

A. Network Model

Many network models have been proposed in the Dolev-Yao setting. For example, sometimes the network is represented as a separate process [12], offering a service between protocol agents; the attacker is then connected to this network, and has the special abilities to eavesdrop messages, drop and modify them, or forge new ones. In other cases, the attacker is the medium [13], and honest agents can only communicate *through* the attacker. Even more detailed network models have been developed [14], where some nodes may have direct, private secured communication with other nodes, while still also being able to communicate through insecure channels, controlled by the attacker. Moreover, in all of these models, each node can in principle be a composition of concurrent processes, that synchronise by communicating over internal (private) channels. This feature allows to tune the granularity of the representation of each participant: two nodes sharing a private channel may be implemented in many different ways: they could be physically different machines, connected by a dedicated infrastructure; or they could be two processes on the same machine, using inter-process communication; or they could even be two functions of the same program, where the secure channel is implemented by function invocation.

In general, it is not trivial to show that all of these models are equivalent in a Dolev-Yao setting, furthermore different network models and agents granularity justify different positions of the monitor with respect to the monitored agent, affecting the way the monitor is actually implemented. Since one goal of this paper is to make this approach practically useful, it is worth reasoning, at least informally, on the practical issues concerning the monitor position in the network model, in order to deal with some common situations that could be found in practice.

As a motivating example, consider the case where the network is modeled as a graph, where vertexes are the network agents, and directed edges are half-duplex communication channels, either public or private, that respectively mean under the attacker control or not. In such a setting, as hinted before, the implementation of a private channel could range from a physical channel, to a function invocation. In the former case, monitoring the secure channel may be feasible, and with minimal effort; in the latter case, monitoring that

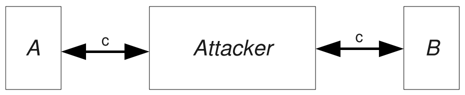

Figure 2. Agents A and B with the attacker.

internal private channel implemented by function invocation may be hardly feasible. However, when monitoring a single agent, it is in general useful to monitor its private channels too, because some secret data may unwillingly flow to other agents through private channels, and then being put on to a public channel by the other agents, which are not monitored: monitoring private channels would avoid such errors. Nevertheless, whether a private channel is easily monitorable or not depends on the granularity of agents representation, and is implementation dependent.

In this paper, we focus on a simple scenario, that is usually found in practice, and thus can be used to faithfully model many security protocols. In this model, that is depicted in figure 2, the attacker is the medium, and every protocol agent can only send data over a single insecure channel c, and private channels between different agents are not allowed. Moreover, agents are sequential and non-recursive. In practice, this model represents the common case where different agents running on different nodes use an untrusted network to exchange data (e.g. a client-server protocol, or a protocol involving trusted third parties); all inter-process communication within a node is kept in its internal state. Although it is believable that this scenario can simulate more complex ones, where multiple public channels are used, or where agents can be multi-threaded or open-ended (recursive), or where different agents share private channels, the formal development of this theory, and where to put the monitor in the enhanced scenarios, is left for future work.

Let us define A as the (correct) model of the agent to be monitored, and M_A as the model of the monitor for agent A.

In order to effectively monitor a protocol role, M_A must be able to gather all messages sent and received by A, and also to access all local data accessed by A. For instance, if A is deciphering some data using its private key, the monitor should have access to that private key, in order to check the content of the plaintext too. It may be argued that giving M_A access to the private keys of A may increase the probability of information leak, or even compromise some non-repudiation properties of the protocol. Although in principle this is true, it must be pointed out that M_A is considered a trusted application, and the use of formal methods for its generation can give high confidence about its correctness.

In order to give M_A access to all exchanged messages and to locally accessed data, it is reasonably assumed that M_A runs in the same environment as A: for example they run on the same operating system with same user privileges.

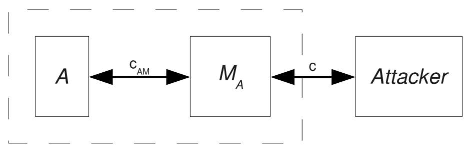

Figure 3. Agent A monitored by M_A and the attacker.

Following the given scenario, when the monitor is not present, A communicates directly with the attacker on channel c, and the attacker will then forward messages to the other parties, still being able to forge, drop or modify messages. When the monitor is present, A communicates with M_A only, through the use of a private channel c_{AM}, not known by the attacker, while M_A is directly connected to the attacker by channel c, as depicted in figure 3, where the dashed box denotes that A and M_A run in the same environment. Note that in A channel c is in fact renamed to c_{AM}. The assumption about c_{AM} not being known by the attacker is reasonable, because we assume that M_A and A run in the same environment.

It is also worth pointing out that monitoring can be performed either "online" or "offline". In the first case, all messages that should be received by A are first checked by M_A, and forwarded to A only if they are safe; similarly, all messages sent out by A are first checked by M_A, and are then relied on the medium. In the second case, all messages sent and received are stored, and then fed to M_A for later inspection. The online paradigm has the advantage of preventing a security property to be violated, by stopping a protocol execution as soon as an unexpected message is detected by the monitor, before the unexpected message is delivered to the intended recipient. For example, this approach prevents that some malicious data are sent to A, or that some information is leaked by A. However online monitoring may introduce some latency that could affect performances, and could even be exploited in timing attacks (which are not captured by the standard Dolev-Yao models). The offline paradigm is able to recognize compromised protocol sessions later, which can still be useful to limit the damage of an attack, helping in its early recognition. For example, if a credit card number is stolen due to an e-commerce protocol attack, and if offline monitoring is run overnight, one can discover the issue at most one day later, thus effectively limiting the time span of the fraud. Offline monitoring has the advantage of not introducing any performance issue. The example in this paper uses the online approach, but the offline approach could have been used as well.

Given a formal protocol specification, a protocol role can be extracted from it, and a monitor specification for that role can be automatically generated.

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

Table I
Spi Calculus terms.

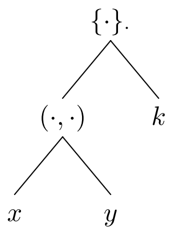

Figure 4. Tree of the $\{(x, y)\}_k$ term.

B. The Spi Calculus

In this paper, the formal models are expressed in Spi Calculus [15]. Briefly, a Spi Calculus specification is a system of concurrent processes that operate on untyped data, called terms. Terms can be exchanged between processes by means of input/output operations. Table I contains the terms defined by the Spi Calculus, while table II shows the processes.

A name n is an atomic value, and a pair (M, N) is a compound term, composed of the terms M and N. The 0 and $suc(M)$ terms represent the value of zero and the logical successor of some term M, respectively. A variable x represents any term, and it can be bound once to the value of another term. A variable that is not bound is free. Names can be regarded as special kinds of variables that can take only atomic terms as values. Then, free variables include free names. The term $\{M\}_N$ represents the encryption of the plaintext M with the symmetric key N, while $H(M)$ represents the result of hashing M. The M^+ and M^- terms represent the public and private part of the keypair M respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively.

Note that for each term, it is possible to build an ordered term tree. For example, the tree for the term $\{(x, y)\}_k$ is depicted in figure 4. By overloading notation, from now on M will represent both a term, and its root node. The $children(M)$ function returns the ordered list of children nodes (of depth 1) for the (root node of) term M. For example, $children(\{(x, y)\}_k)$ returns the list $[(x, y), k]$; if a is a name or variable, $children(a)$ returns $[]$, which denotes

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

Table II
SPI CALCULUS PROCESSES.

the empty list.

Informally, the $\overline{M}\langle N\rangle.P$ process sends message N on channel M, and then behaves like P, while the $M(x).P$ process receives a message from channel M, and then behaves like P, with x bound to the received term in P. A process P can perform an input or output operation iff there is a reacting process Q that is ready to perform the dual output or input operation. Note, however, that processes run within an environment (the Dolev-Yao attacker) that is always ready to perform input or output operations. Composition $P|Q$ means parallel execution of processes P and Q, while replication $!P$ means an unbounded number of instances of P run in parallel. The restriction process $(\nu n)P$ indicates that n is a fresh name (i.e. not previously used, and unknown to the attacker) in P. The match process executes like P, if M equals N, otherwise is stuck. The nil process does nothing. The pair splitting process binds the variables x and y to the components of the pair M, otherwise, if M is not a pair, the process is stuck. The integer case process executes like P if M is 0, else it executes like Q if M is $suc(N)$ and x is bound to N, otherwise the process is stuck. If L is $\{M\}_N$, then the shared-key decryption process executes like P, with x bound to M, else it is stuck, and analogous reasoning holds for the decryption and signature check processes.

The assumption that A is a sequential process, means that composition and replication processes are never used in its specification.

The semantics of the Spi Calculus has been originally expressed for closed processes, by means of a reaction relation $P \to P'$ and a reduction relation $P > P'$ [15]. $P \to P'$ means that P can evolve into P' after a message exchange between two parallel components of P, while $P > P'$ means that P can evolve into P' by performing some other (internal) operation. In this paper, instead, we need expressing the semantics of open processes, i.e. the possible evolutions of an open process P, regardless of its environment. For this purpose, we introduce a classical labeled transition system (LTS). In this system, a τ transition $P \overset{\tau}{\to} P'$ means $P \to P'$ or $P > P'$, i.e. P can evolve into P' without interaction with its environment. Instead, $P \overset{M!N}{\to} P'$ and $P \overset{M?N}{\to} P'$ mean that P can interact with its environment by respectively sending or receiving N on channel M. Formally,

$$\begin{array}{lll} P \overset{M!N}{\to} P' & \text{means} & \exists \overline{y}|\forall Q.\ (P|M(x).Q) \to \\ & & \qquad (\nu\overline{y})(P'|Q[N/x]) \\ P \overset{M?N}{\to} P' & \text{means} & \exists \overline{y}|\forall Q.\ (P|(\nu\overline{y})\overline{M}\langle N\rangle.Q) \to \\ & & \qquad (\nu\overline{y})(P'|Q) \end{array}$$

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

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

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

$$\frac{P \overset{\mathcal{L}}{\to} P'}{(\nu n)P \overset{\mathcal{L}}{\to} (\nu n)P'}$$

where $\mathcal{L}$ ranges over labels.

Moreover, in this paper we need to represent Spi Calculus states in a slightly different way. According to the notation introduced in [15] and extended here for open processes, a transition can be in general written as

$$P \overset{\mathcal{L}}{\to} P'\sigma'$$

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

$$P_1\sigma \overset{\mathcal{L}}{\to} P_1'\sigma\sigma'$$

and a state can be divided into two components: a process expression followed by a variable substitution.

III. THE MONITOR GENERATION FUNCTION

The $a2m$ function is formally defined in this paper: it translates a sequential protocol role specification into a monitor specification for that role. Before introducing the function definition, the concepts of *known* and *reconstructed* terms are given. For any Spi Calculus state, a term T is said to be *known* by the monitor through variable $\_T$, iff $\_T$ is bound to T. This can happen either because the implementation of M_A has access to the agent's memory location where T is stored; or because T can be read from a communication channel, and M_A stores T in variable $\_T$. A compound term T (that is not a name or a variable) is said to be *reconstructed*, if all the terms in the $children(T)$ list are known or reconstructed.

Let us clarify these concepts by an example. Consider this made up specification for A:

$$(\nu n)\overline{c}\langle H(n)\rangle.\overline{c}\langle n\rangle.\mathbf{0}$$

where in the initial state, both n and $H(n)$ are assumed to be not known by M_A, for example because accessing the memory area where their values are stored by A would require too much effort. When $H(n)$ is sent over channel c, the monitor stores this term in the $\_H(n)$ variable, so $H(n)$ becomes known through $\_H(n)$, and M_A does not perform any check on it. However, when n is sent over c, the monitor stores it in $\_n$, so n becomes known through $\_n$, and $H(n)$ can be reconstructed as $H(\_n)$. Now $\_H(n)$ can be matched against the reconstructed value $H(\_n)$, to ensure data coherence. Note that the monitor implementation presented in this paper does not enforce that nonces are actually different for each protocol run. To enable this, the monitor should track all previously used values, in order to ensure that no value is used twice. However, this check could be highly costly, both in time and resources, especially in the online mode. In order to drop this check, it is needed to assume that the random value generator in the monitored agent is correctly implemented.

Formally, let $SpiTerm$ be the set of Spi Calculus terms, then $known : SpiTerm \rightarrow SpiTerm$ is a function mapping each known term to the variable where it is stored. In general, more than one variable could be bound to the same term, making $known$ not a function. However, the monitor generator function is defined such that $known$ is a function. Note that $known$ is only defined on the known terms, that is $dom(known)$ only contains those terms for which there exist a variable binding. The $reconstructed : SpiTerm \rightarrow SpiTerm$ function returns, when possible, a term that "reconstructs" the given term by using known children and, if they are not available, by using the reconstructed ones. The formal definition is trivial but quite verbose, while an example can better help in understanding its purpose: if $known(M) = \_M$ and $known(H(M)) = \_H(M)$, then $reconstructed((H(M), M))$ returns $(\_H(M), \_M)$; note that it is not the case that $reconstructed((H(M), M))$ returns $(H(\_M), \_M)$, because known children are preferred, when available. The domain of $reconstructed$ is formally defined as

$$dom(reconstructed) = \{T : SpiTerm \mid children(T) \neq \emptyset \\ \wedge \forall\, t \in children(T) \cdot t \in dom(known) \cup dom(reconstructed)\}$$

where the $\in$ symbol is overloaded to lists, meaning that the given term appears at least once in the list. The $children(T) \neq \emptyset$ predicate avoids that names or variables are in $dom(reconstructed)$, while the universally quantified predicate ensures that each child of the root node of term T is known or reconstructed. Again, from the definition it follows that terms that cannot be reconstructed are not in $dom(reconstructed)$. Note that, as terms become known, the $reconstructed$ function is updated too. In the example given above, if M was not known, then it was not possible to reconstruct $(H(M), M)$; however, if later M became known (for example, because it was sent over a channel), then $(H(M), M)$ would become reconstructed.

The $choose : SpiTerm \rightarrow SpiTerm$ function returns the known or, if it is not available, the reconstructed term for the given term, and is defined as

```
function $choose(T)$
    if $T \in dom(known)$ then
        return $known(T)$
    else if $T \in dom(reconstructed)$ then
        return $reconstructed(T)$
```

with $dom(choose) = dom(known) \cup dom(reconstructed)$.

Let Spi be the set of Spi Calculus processes, then $a2m : Spi \rightarrow Spi$ is the function that translates a Spi Calculus agent specification into the Spi Calculus specification of the corresponding monitor. Besides the aforecited functions, the $a2m$ function also uses a list Q. If Q and Q' are two lists, then $Q|Q'$ denotes the concatenation of the two. In order to keep the notation simple, each function does not take Q, $known$ and $reconstructed$ as input parameters, instead they are updated by the functions side-effects. The $a2m$ function is formally defined in figure 5. For brevity, in this and the following function definitions, asymmetric encryption is not showed; it is handled like symmetric encryption, when the appropriate key is known, or like hashing, otherwise. In general, the function operates based on the type of process. For each process, if all the free terms are known or reconstructed, then the process is translated into the monitor process, otherwise an error is thrown, because some terms cannot be monitored. By applying some basic static analysis techniques, it can be showed that if all free names and variables are assumed to be known (which is reasonable, as they are constants), then this error is never thrown, and the monitor specification is correctly returned. Some significant cases are explained. If the process P to be translated is a match $[M\ is\ N]P'$ process, then the $[choose(M)\ is\ choose(N)]$ process is returned, and the function is recursively invoked on P'. Note that the $choose$ function is used to resolve the actual names of M and N, because they may be mapped to some known or reconstructed term. If P is $let\ (x, y) = M\ in\ P'$, then it is translated into $let\ (x, y) = choose(M)\ in\ a2m(P')$. Note that x and y are bound in $a2m(P')$, and $known$ is updated accordingly, so that in later accesses to x or y, they will be

```
function $a2m(P)$
    if $P$ is $[M\ is\ N]P'$ then
        if $M, N \in dom(choose)$ then
            return $[choose(M)\ is\ choose(N)]\ a2m(P')$
    else if $P$ is $let\ (x, y) = M\ in\ P'$ then
        if $M \in dom(choose)$ then
            $known := known \cup \{x \to x, y \to y\}$
            return $let\ (x, y) = choose(M)\ in\ a2m(P')$
    else if $P$ is $\mathbf{0}$ then
        return $emptyQ()\ \mathbf{0}$
    else if $P$ is $case\ M\ of\ 0 : P'\ suc(x) : P''$ then
        if $M \in dom(choose)$ then
            $br1 := a2m(P')$
            $known := known \cup \{x \to x\}$
            $br2 := a2m(P'')$
            return $case\ choose(M)\ of\ 0 : br1\ suc(x) : br2$
    else if $P$ is $case\ L\ of\ \{x\}_N\ in\ P'$ then
        if $L, N \in dom(choose)$ then
            $known := known \cup \{x \to x\}$
            return $case\ choose(L)\ of\ \{x\}_{choose(N)}\ in\ a2m(P')$
    else if $P$ is $c(x).P'$ then
        $known := known \cup \{x \to x\}$
        $Q := Q|[x]$
        return $c(x).a2m(P')$
    else if $P$ is $\overline{c}\langle M\rangle.P'$ then
        return $emptyQ()\ c_{AM}(\_M)\ check(M, \_M)\ \overline{c}\langle\_M\rangle\ a2m(P')$
    else if $P$ is $(\nu n)P'$ then
        return $a2m(P')$
    error Required data not monitored; monitor not sound.
```

Figure 5. From agent specification to monitor specification.

resolved as $known(x) = x$ and $known(y) = y$. In the base case where P is the **0** process, the content of Q is forwarded to A by the $emptyQ$ function (more on this later), then the **0** process is returned. If P is $c(x).P'$, then the monitor receives message x on behalf of the agent, buffers x in Q, and then behaves following what is done by the agent; x is bound by the input process, and the $known$ function is updated accordingly. The received message stored in x is not forwarded to A immediately, because this could lead A to receive some malicious data, that could for example enable some denial of service attack. Instead, the received data are buffered, and will be forwarded to A only when necessary: that is when the process should end (**0** case), or when some output data from A are expected: A should receive all buffered data, because they may be needed to compute the value to be sent. Often P' will contain some checks on the received data, and they will be implemented by the monitor as well. Note that if two ore more inputs are made by A before an output or the process end, they will all be buffered, and only forwarded in the right order when necessary. If P is $\overline{c}\langle M\rangle.P'$, then, as explained before, all buffered data are forwarded to A, so that it is able to compute M; then M is received by M_A on the private channel, and stored in the $\_M$ variable. Before the monitor forwards $\_M$ to the attacker, all possible checks on $\_M$ are done, to check that it matches with M. These checks are introduced by the $check$ function, explained in details below; what checks can be performed on the received data depends on what terms are known or reconstructed. Finally, any restriction process $(\nu n)P'$ is ignored, because the monitor shall never create fresh data, rather it should $know$ what fresh data has been generated by A. If the monitor can read the value of n from the agent's state, then $known(n) = n$ since the initial state; otherwise, $n \notin dom(known)$ at the beginning, however, n (or some term that depends on it, like $H(n)$) may become known later, if it is sent over a channel.

The auxiliary function $emptyQ$ outputs the buffered variables over the private channel, so that buffered data are actually forwarded to the monitored agent. Its formal definition is

```
function $emptyQ$
    if $Q$ is $[]$ then
        return
    else
        read $Q$ as $[x]|Q'$
        $Q := Q'$
        return $\overline{c_{AM}}\langle x\rangle.\ emptyQ()$
```

The $check : SpiTerm \times SpiTerm \to Spi$ takes the expected term E as first parameter, and the variable bound to the received term R as second parameter. The function performs all possible checks between them. In particular, if E is already known or reconstructed, then it is checked whether the variable already bound to E matches R. In case E is not known or reconstructed, the $explode$ function is invoked that, if possible, tries to de-construct R and check its subterm, otherwise, by calling the $backcheck$ function, E is set to be known by R. Finally, as mapping a new known term enables new terms to be constructed, the $backcheck$ function also adds all those checks that are made now possible because of the new constructed terms. The formal definition is given in figure 6. Some relevant cases of the $explode$ function are commented; note that, when the function is invoked, $E \notin dom(choose)$, and in particular $E \notin dom(known)$ is true; also note that the top level **if** switches are made on E, and not on R, because E is the "expected term" that has a known structure, while R is a variable that should have the same structure, which is the reason why R is de-constructed when possible. If E is a name or a variable, then R cannot be de-constructed, so, by the $backcheck$ function, E is set to be known through R, and all newly enabled checks are added. For example, after E is known through R, $H(E)$ is reconstructed by $H(R)$; in case $H(E)$ was previously known, say through $\_H(E)$, the $backcheck$ function will ensure that $[\_H(E)\ is\ H(R)]$. If E is the pair (E', E''), then it should it possible to de-construct R by using the pair splitting process; if the pair splitting fails, then R was not a pair, thus storing a wrong term. After R is split into R' and R'', its parts are recursively checked against their expected terms. Note that in this case E is *not* set as known through R. Indeed, if R

```
function $check(E, R)$
    if $E \in dom(choose)$ then
        return $[R\ is\ choose(E)]$
    else
        $explode(E, R)$
function $explode(E, R)$
    if $E$ is name or variable then
        return $backcheck(E, R)$
    else if $E$ is $(E', E'')$ then
        return $let\ (R', R'') = R\ in\ check(E', R')\ check(E'', R'')$
    else if $E$ is $\{E'\}_K$ then
        if $K \in dom(choose)$ then
            return $case\ R\ of\ \{R'\}_{choose(K)}\ in\ check(E', R')$
        else
            return $backcheck(E, R)$
    else if $E$ is $H(E')$ then
        return $backcheck(E, R)$
function $backcheck(E, R)$
    $formerlyKnown := \{E' \in SpiTerm | E' \in dom(known) \wedge E' \notin dom(reconstructed)\}$
    $known := known \cup \{E \to R\}$   ▷ Note that $reconstructed$ gets updated too
    for all $E' \in formerlyKnown \cap dom(reconstructed)$ do
        $res := res\ [known(E')\ is\ reconstructed(E')]$
    return $res$
```

Figure 6. Auxiliary functions: $check$, $explode$ and $backcheck$.

can be split into R' and R'', then it is always possible to reconstruct it by using its parts; if E was set known through R, then the $backcheck$ function could possibly introduce later the $[R\ is\ (R', R'')]$ check, which is redundant (making the monitor implementation inefficient). If E is the hashing $H(E')$, then there is no way to de-construct it, so the $backcheck$ function is invoked, like in the name or variable case. Finally, if E is $\{E'\}_K$, then two cases are possible. If K is known or reconstructed, then it is possible to try to de-construct the encryption, getting the plaintext R', that is then checked against its expected term E'; like in the pair splitting case, E is not set to by known trough R. Else, when K is not known or reconstructed, it is not possible to de-construct the encryption, so, like in the hashing case, the $backcheck$ function is invoked.

Finally, the definition of M_A is

$$M_A \triangleq preChecks()\ a2m(A)$$

with Q being the empty list [] in the initial state, and $preChecks$ a consistency function that checks that all terms which are both known and reconstructed in the initial state are the same. Formally

```
function $preChecks$
    for all $T \in SpiTerm\, |\, T \in dom(known) \wedge T \in dom(reconstructed)$ do
        $res := res\ [known(T)\ is\ reconstructed(T)]$
    return $res$
```

Also note that if all free variables and free names in A are known, then the error state of $a2m$ is unlikely to be reached. The error state could still be reached for example if an unknown fresh value is checked against some data,

```
1a: A(M,k) :=            1m: MA(k,_H(M)) :=
2a:   cAM<{M}k>.         2m:   cAM(_{M}k).
                         3m:   case _{M}k of {_M}k in
                         4m:   [_H(M) is H(_M)]
                         5m:   c<_{M}k>.
3a:   cAM(x).            6m:   c(x).
4a:   [x is H(M)]        7m:   [x is _H(M)]
5a:   0                  8m:   cAM<x>.
                         9m:   0
```

Figure 7. Example specification of agent A along with its derived monitor M_A.

immediately after its creation: $(\nu n)[n\ is\ M]$ or $(\nu n)[n\ is\ n]$; however these two cases are rather pointless because the first check would always fail, and the second one always pass (and can thus be removed), by definition.

In order to better understand how the $a2m$ function actually works, an example is now shown. Note that the focus of this example is on the way the $a2m$ function operates on a given agent, rather than on monitoring a security protocol. A real-life monitoring example dealing with the SSL protocol is provided in section IV.

Figure 7 shows a specification A for an agent, and its derived monitor specification M_A. Here, an ASCII syntax of Spi Calculus is used: the 'ν' symbol is replaced by the '@' symbol, and the overline in the output process is omitted (input and output processes can still be distinguished by the different brackets), moreover comments are enclosed between $/\star$ and $\star/$. At line 1a the agent A process is declared: it has two free variables, a message M and a symmetric key k. At line 2a A sends the encryption of M with key k over the communication channel (which is c_{AM} when being monitored, but would be c in the original specification). Then, at line 3a it receives a message that is stored into variable x, and, at line 4a, the received message is checked to be equal to the hashing of M: if this is the case, the process correctly terminates. At line 1m, the monitor M_A is declared: to make this example significant, it is assumed that in the initial state $known(k) = k$ and $known(H(M)) = \_H(M)$, that is the monitor knows the key k used by A through the variable k, but does not know M (for example because those data cannot be accessed); instead the monitor has access to the location where $H(M)$ is stored, and this value is bound to the variable $\_H(M)$ in the monitor. The $preChecks$ function does not add any match process, since there is no term being both known and reconstructed in the initial state. When line 2a is translated by $a2m$, lines 2m–5m are produced. By looking at the $a2m$ definition, first the $emptyQ$ function is invoked, which returns an empty statement, since the list Q is empty in the initial state. Then the data sent by A are received by the monitor at line 2m, and stored in variable $\_\{M\}_k$. Afterwards, the $check(\{M\}_k, \_\{M\}_k)$ function is invoked. Since $\{M\}_k$

is not known (by hypothesis) or reconstructed (because M is not known or reconstructed), the $explode(\{M\}_k, \_\{M\}_k)$ function is invoked, to possibly dissect the received value. Since $\{M\}_k$ is an encryption, and k is known, the decryption case process is output at line 3m, binding $\_M$ to the value of the plaintext, that should be M. Now the $check(M, \_M)$ function is invoked. Since M is not known or reconstructed, the $explode(M, \_M)$ is called in turn. Since M is a name, $\_M$ cannot be dissected; instead, the $backcheck(M, \_M)$ is called, to let M be known through $\_M$. Note that, before the $known(M) = \_M$ relation is defined in $backcheck$, $H(M)$ is known through $\_H(M)$, but it is not reconstructed, so $H(M) \in formerlyKnown$ holds. After the assignment, note that $reconstructed(H(M)) = H(\_M)$, so $H(M)$ is a term that was previously only known through $\_H(M)$, but that can be now be reconstructed by $H(\_M)$. This idea is captured by $formerlyKnown \cap dom(reconstructed) = \{H(M)\}$, which leads to the match process output at line 4m, checking that the previously known term for $H(M)$ and the current reconstructed term are the same. Now, we are back in the $a2m$ function, that puts the output process in line 5m, forwarding received data to the attacker, if all checks passed. Then, line 3a is translated into line 6m. Note that $known(x) = x$ will hold, and that Q is now $[x]$, because x has been received and buffered by the monitor. Line 4a is then translated into line 7m, where $choose(x) = x$ and $choose(H(M)) = \_H(M)$. Note that testing x against $\_H(M)$ or $H(\_M)$ would actually be the same, since they have already been checked to match. Finally, line 5a is translated into lines 8m and 9m. First, all buffered data (x in this case) are forwarded to A, then the monitor correctly ends.

IV. An SSL Server Monitor Example

It is now shown how the monitoring approach presented in this paper can be practically applied to existing server side implementations of a widely used protocol, such as SSL [16]. As stated above, the online monitoring paradigm will be used in this example.

A. Monitor Specification

As a first step, a formal Spi Calculus specification of the SSL protocol is needed. In particular, since server side implementations are going to be monitored in this example, only the server protocol role specification is reported here. The ASCII syntax of Spi Calculus is used in this example. Also note that the "Shared Key" term $M^{\sim}$ (ASCII `M~`) is added, in order to allow symmetric key construction from raw key material; it follows that, like for the asymmetric decryption, a symmetric decryption only succeeds if the appropriate symmetric key was used. This minor variation is irrelevant w.r.t. the formal approach presented above. Moreover, to make the specification more readable, the following syntactic sugar is used: lists of terms like (A, B, C) are allowed, and they are translated into left associated nested pairs, like $((A, B), C)$; a `rename n = M in P` process is introduced, that renames the term M to n and then behaves like P (during evaluation, this process is discarded, and n is just syntactically substituted by M).

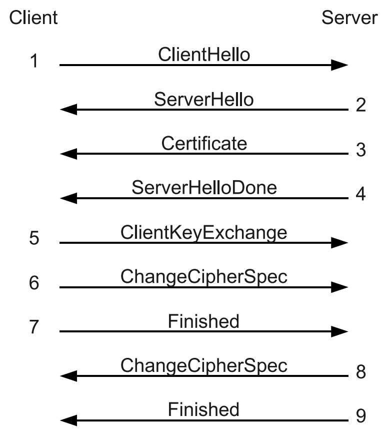

Figure 8. Typical SSL scenario.

In its full shape, SSL is a rather complex protocol, where many interaction scenarios between client and server are possible, and different sets of cryptographic algorithms (called *cipher suites*) can be negotiated. For simplicity, this example considers only one scenario of the SSL protocol version 3.0, and it is assumed that the same cipher suite is always negotiated. Despite these simplifications, we believe that the considered SSL fragment is still significant, and that adding full SSL support would increase the example complexity more than its significance.

The SSL scenario considered in this example is depicted informally in figure 8. The first message is the ClientHello, and is sent from the client to the server. It contains the highest protocol version supported by the client, a random value, a session ID for session resuming, and the set of supported cipher suites and compression methods. In order to allow the execution of the scenario considered in this example, the client should send at least 3.0 as the highest supported protocol version, and it should send 0 as session ID, so that no session resuming will be performed. Moreover, the client should at least support the always-negotiated cipher suite, namely `SSL_RSA_WITH_3DES_EDE_CBC_SHA`, with no compression enabled. In the second message the server replies by sending its ServerHello message, that contains the chosen protocol version, a random value, a fresh session ID and the chosen cipher suite and compression method. Again, in the chosen scenario, the server should choose protocol version 3.0, and should always select the `SSL_RSA_WITH_3DES_EDE_CBC_SHA` cipher suite (if it is supported by the client, else the protocol session should be shut down) with no compression enabled. Then the server sends the Certificate message to the client, that contains a certificate chain for the server's public key, used by the client to authenticate the server. The fourth message is the ServerHelloDone. It contains no data, but signals the client that the server ended its negotiation part, so the client can move to the next protocol stage. The client replies with the ClientKeyExchange message, that contains the Premaster Secret (PMS) encrypted by the server public key, which is got from the server's certificate sent in the third message. The PMS is a client chosen random sequence of bytes that will be used by both parties to derive some shared secrets used for symmetric encryption of application data. By applying an SSL custom hashing function to the PMS and the client and server random data, both client and server can compute the same Master Secret (MS). The bytes of the MS are then extended (again by using a custom SSL hashing algorithm) to as many byte as required by the negotiated ciphersuite, obtaining the Key Material (KM). Finally, different subsets of bytes of the KM are used as shared secrets (e.g. encryption keys or initialization vectors IVs). The sixth message is the ChangeCipherSpec. This message contains no data, but signals the server that the client will start using the negotiated cryptographic algorithm from the next message on. The client then sends its Finished message, that contains the final hash, that hashes all relevant session information: all exchanged messages (excluding the ChangeCipherSpec ones) and the MS are included in the final hash, plus some constant data identifying the protocol role (client or server) that sent the final hash. In fact, the Finished message includes two versions of the same final hash, one using the MD5 algorithm, and one using the SHA-1 algorithm. Note that Message Authentication Code (MAC) and encryption are applied to the Finished message sent by the client, as the client already sent its ChangeCipherSpec message. The sever is expected to verify that the MAC is valid for the client's Finished message, and to check that the locally computed final hash matches the final hash sent by the client. If this is the case, then the server sends its ChangeCipherSpec message to client, and its Finished message, that comes with MAC and encryption too. After these nine messages are exchanged, the protocol handshake is complete, and next messages will contain application data, along with MAC and encrypted with the negotiated algorithms.

A possible server Spi Calculus specification is shown in figure 9. In order to verify any security property on this specification, the full SSL specification, including the client and protocol sessions instantiations should be required. However, the scope of this paper is not to prove security properties of an SSL specification; this goal has already been achieved, for example by the AVISPA project [17]. Here it is assumed instead that the specification of the server is correct, and thus secure, so that the monitoring approach can be shown.

At line 1S, the server process is declared; for brevity, input parameters are omitted, each free name is an input parameter. At line 2S the server receives the ClientHello `c_hello` message, and at line 3S this message is split into its constituent parts: the client version `c_version`, the client random value `c_rand`, the ID of a possible previous protocol session `c_SID` to be resumed, the client supported cipher suites `c_ciph_suite` and the client supported compression methods `c_comp_method`. From line 4S to 7S the received values are checked against the expected ones. The protocol version must match "3.0", represented by the Spi Calculus constant term `THREE_DOT_ZERO`. Since in this example session resume is not supported, the ID of a possible previous protocol session must be zero; moreover, since in this example the same cipher suite is always going to be negotiated, at lines 6S and 7S it is checked that the client supported encryption algorithms and compression methods are the expected ones. When all checks are done, at lines 8S and 9S the server generates its random value `s_rand` and a fresh session ID `SID`. At line 10S the ServerHello message `S_HELLO` is created, and at line 11S it is sent to the client, along with the constant server Certificate message `S_CERT` and the constant ServerHelloDone message `S_HELLO_DONE`. Note that in SSL, each message belongs to one of several upper layer protocols, and each upper layer message is encapsulated into a message of a lower layer protocol, called the Record Layer protocol. Different messages of the same upper layer protocol may be encapsulated into a single Record Layer message. For example, the ServerHello, server Certificate and ServerHelloDone messages all belong to the same upper layer protocol, and thus they *may* be encapsulated into the same Record Layer message, or they could be encapsulated into different Record Layer messages: this is implementation dependent. Thanks to the marshalling/unmarshalling features of the spi2java framework, which is the MDD framework used to generate the monitor implementation and is presented in section IV-B, the Spi Calculus specification can abstract this kind of details away, and they will be fully dealt in the monitor implementation, in a provably secure way. At line 12S the ClientKeyExchange message is received, and at line 13S the message header and message content are split. Both the whole message including the header (`encrypted_PMS_msg`) and the message content (`encrypted_PMS`) must be represented in Spi Calculus, because the former is used to compute the final hash, which includes all exchanged data including headers, while the latter is needed to get the PMS. Indeed, at line 14S the server gets the PMS by decrypting `encrypted_PMS` with its private key `s_PriKey`. At line 15S the client's ChangeCipherSpec message is received. Since this message should have a constant value, at line 16S the received value is matched against the expected one. Then, at line 17S, the server receives the client's finished message

```
 1S Server() :=
 2S   c(c_hello).
 3S   let (c_version,c_rand,c_SID,c_ciph_suite,c_comp_method) = c_hello in
 4S   [ c_version is THREE_DOT_ZERO ]
 5S   [ c_SID is ZERO ]
 6S   [ c_ciph_suite is SSL_RSA_WITH_3DES_EDE_CBC_SHA ]
 7S   [ c_comp_method is comp_NULL ]
 8S   (@s_rand)
 9S   (@SID)
10S   rename S_HELLO = (THREE_DOT_ZERO,s_rand,SID,SSL_RSA_WITH_3DES_EDE_CBC_SHA,comp_NULL) in
11S   c<S_HELLO,S_CERT,S_HELLO_DONE>.
12S   c(encrypted_PMS_msg).
13S   let (ePMSHead,encrypted_PMS) = encrypted_PMS_msg in
14S   case encrypted_PMS of {[PMS]}s_PriKey in
15S   c(c_ChgCipherSpec).
16S   [ c_ChgCipherSpec is CHG_CIPH_SPEC ]
17S   c(c_encrypted_Finish).
18S   rename MS = H(PMS,c_rand,s_rand) in
19S   rename KM = H(MS,c_rand,s_rand) in
20S   rename dummy1 = H(KM,C_WRITE_IV) in
21S   rename dummy2 = H(KM,S_WRITE_IV) in
22S   case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
23S   let (c_Finish,c_MAC) = c_Finish_and_MAC in
24S   [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
25S   let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
26S   [ final_Hash_MD5 is
27S     H((c_hello,S_HELLO,S_CERT,S_HELLO_DONE,encrypted_PMS_msg),C_ROLE,MS,MD5) ]
28S   [ final_Hash_SHA is
29S     H((c_hello,S_HELLO,S_CERT,S_HELLO_DONE,encrypted_PMS_msg),C_ROLE,MS,SHA) ]
30S   c<CHG_CIPH_SPEC>.
31S   rename DATA = (c_hello,S_HELLO,S_CERT,S_HELLO_DONE,encrypted_PMS_msg,c_Finish) in
32S   rename S_FINISH = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
33S   (@pad)
34S   c<{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH),pad}(KM,S_WRITE_KEY)~>.
35S   0
```

Figure 9. A possible Spi Calculus specification of an SSL server.

`c_encrypted_finish`, that contains the encryption of client's final hash, along with its MAC. At lines 18S and 19S the MS and the KM are renamed, so that they can be conveniently used later. The MS is generated by hashing the PMS with the client and server random values; the implementation will ensure that the hashing algorithm is the one prescribed by the SSL specification. Same reasoning applies for the KM. At lines 20S and 21S instead, the rename process is used to "declare" the client and server IVs respectively: these terms are never used in the specification, however, they must be explicitly represented, because they will be used as cryptographic parameters of some encryption operations; more on this in section IV-B when showing the usage of the spi2java framework to generate the monitor implementation. Note that the bytes composing the IVs are just a subset of the bytes composing the KM; in Spi Calculus they are represented as hashings of the KM and a marker, indicating which bytes of the KM should be used. This means that if a real attacker gets to know an IV, it actually gets to know at least part of the KM; in Spi Calculus this is not the case, because the hash function is assumed to be not invertible. In fact, such property cannot be represented in a plain Dolev-Yao model. Fortunately, this issue can be easily overcome during formal analysis of the protocol, by not only verifying that the KM remains secret, but also that the IVs remain secret too. Then, at line 22S the client's Finished message is decrypted, and the plaintext is expected to be the client's final hash along with its MAC. Like for the IVs, the decryption key used `(KM,C_WRITE_KEY)~` is obtained by creating a shared key, starting from the key material `KM` and a marker `C_WRITE_KEY` that indicates which portion of the KM to use (in this case the bytes corresponding to the client write key). At line 23S the plaintext is split into the final hash `c_Finish` and its MAC `c_MAC`. Then, at line 24S, `c_MAC` is matched against the locally reconstructed MAC, obtained by hashing the client MAC secret `(KM,C_MAC_SEC)~` with the value to be authenticated, that is `c_Finish`. Again, the implementation will use the hashing algorithm prescribed by the SSL specification. Once the MAC is verified, at line 25S the client's final hash is split into the MD5 and the SHA-1 versions, then, at lines 26S–29S both versions are checked against the locally reconstructed values. Note that Spi Calculus does not natively support different hashing algorithms: the hashing term $H(M)$ is just a non invertible function, and the underlying cryptographic algorithm is abstracted away. In general this is acceptable, as the spi2java framework allows to associate a specific cryptographic algorithm to each

Spi Calculus hashing term. However, in this case we need to distinguish at the Spi Calculus level between the final hash made with MD5, and the final hash made with SHA-1, both hashing the same data. To solve this issue, the hashing algorithm is explicitly represented among the hashed data, so to make the two hasings syntactically different (lines 27S and 29S). In the implementation, the `MD5` and `SHA` markers will contain empty data, so that the final hashes will be computed only on the right data; still, it will be possible to distinguish the two hashes, so to assign to each of them the correct hashing algorithm. After the final hashes are checked, at line 30S the server sends its ChangeCipherSpec message, and at lines 31S–34S, it sends its encrypted final hash, along with its MAC. At line 31S, the data to be hashed are renamed to `DATA`, then at line 32S the server's final hash is computed, by creating a pair containing the MD5 and SHA-1 versions of the hash. At line 33S some random padding required to align the plaintext length to the block cipher size is created and at line 34S the server's finished message is sent: it is composed of the server's final hash, its MAC and the padding, all of this encrypted by the server write key. In principle, the padding could be abstracted away from the Spi Calculus specification, and let cryptographic functions take care of randomly generating it when needed. However, as it will be clearer later, not including the padding at the specification level, would make it impossible for the monitor to check the server final hash, so in this case it is needed to model padding at the Spi Calculus level. Finally, the server specification terminates at line 35S.

In figure 10, the $a2m$ function described in section III is applied on the server specification, in order to obtain the online monitor specification for the server role. It is assumed that the monitor has access to the server private key, which is then *known*, while it is not able to read the freshly generated server random value `s_rand`, the session ID `SID` and the random padding which are then not known nor reconstructed at generation time. These assumptions are reasonable and easily met in practice. Indeed, server private keys are usually stored in files; if the monitor runs with the same privileges of the server, then it can access to the server private key, thus satisfying the assumption. Instead, it will be usually much harder to get the server generated random values directly from the server implementation, because these values will only be stored in memory, in a possibly unpredictable location, which is also implementation dependent. Since the server random value and the session ID are sent in clear over the wire anyway, and the padding can be discarded, it is fine not to put any effort in reading them from the program memory, but letting them become known as the protocol session is executed.

Note that each free variable in the server specification is constant data, so it is reasonable to assume that they are all known to the monitor. That is, for each free variable x in the server specification, $known(x) = x$ holds. In order to make the automatically generated monitor specification more readable, some rename processes have been added, and bound variables have been given significant names.

At line 2M the monitor receives on the public channel `c` the ClientHello message `c_hello` on behalf of the server. By looking at the $a2m$ function definition, after the input process is translated, $known(c\_hello) = c\_hello$ holds, and c_hello is added to the queue of buffered messages. Then, from line 3M to 7M the translation continues replicating in the monitor all the checks that are performed in the server. The two restriction processes at lines 8S and 9S in the server specification are dropped, but for convenience they are just commented out at lines 8M and 9M of the monitor specification. Then, the output process at line 11S is translated: first all buffered messages (`c_hello` in this case) are relayed to the server over the private channel `c_int` (line 10M), then the ServerHello, Certificate and ServerHelloDone messages are received from the server, and stored in the `s_hello_s_cert_s_hello_done` variable (line 11M). Afterwards, the latter variable is checked against the term to which it should be bound, namely `(S_HELLO,S_CERT,S_HELLO_DONE)`. Since the `s_rand` and `SID` names composing the `S_HELLO` term are not known in the monitor, it is not possible to reconstruct the latter term, making it impossible to reconstruct the whole `(S_HELLO,S_CERT,S_HELLO_DONE)` term. For this reason, the `s_hello_s_cert_s_hello_done` variable is exploded. Following the $explode$ definition, at line 12M the variable is split into its parts, that should match the ServerHello, Certificate and ServerHelloDone messages, then the $check$ function is invoked on each of the split variables. The `s_hello` variable is checked from line 13M to 17M. Note that variables that should be bound to constant terms are matched against them, while `s_rand` and `SID` are just bound and not checked, since they are bound to the fresh names generated by the server. As `s_rand` and `SID` become known, the `(S_HELLO,S_CERT,S_HELLO_DONE)` term becomes reconstructed, but it would be now useless to check the reconstructed term against its expected term, because the match would certainly pass. Indeed, the $backcheck$ function is properly defined, so as to avoid the introduction of these redundant checks. The `s_cert` and `s_hello_done` variables are then checked from line 18M to 21M, and at line 22M the received `s_hello_s_cert_s_hello_done` message is finally forwarded over the public channel. The $a2m$ function continues translating line 12S, so at line 23M the ClientKeyExchange message is received and buffered by the monitor, and translation continues. Indeed, from line 23M to 40M, all checks made by the server from line 12S to 29S are plainly replicated into the monitor specification, and all input operations are replicated and their content buffered in the queue. Note that the decryption

```
Monitor_S() :=
  c(c_hello).
  let (c_version, c_rand, c_SID, c_ciph_suite, c_comp_method) = c_hello in
  [ c_version is THREE_DOT_ZERO ]
  [ c_SID is ZERO ]
  [ c_ciph_suite is SSL_RSA_WITH_3DES_EDE_CBC_SHA ]
  [ c_comp_method is comp_NULL ]
  /* (@s_rand) */
  /* (@SID) */
  c_int<c_hello>.
  c_int(s_hello_s_cert_s_hello_done).
  let (s_hello,s_cert,s_hello_done) = s_hello_s_cert_s_hello_done in
  /* checks for s_hello */
  let (x_THREE_DOT_ZERO, s_rand, SID, x_SSL_RSA_WITH_3DES_EDE_CBC_SHA, x_comp_NULL) = s_hello in
  [ x_THREE_DOT_ZERO is THREE_DOT_ZERO ]
  [ x_SSL_RSA_WITH_3DES_EDE_CBC_SHA is SSL_RSA_WITH_3DES_EDE_CBC_SHA ]
  [ x_comp_NULL is comp_NULL ]
  /* checks for s_cert */
  [ s_cert is S_CERT]
  /* checks for s_hello_done */
  [ s_hello_done is S_HELLO_DONE ]
  c<s_hello_s_cert_s_hello_done>.
  c(encrypted_PMS_msg).
  let (ePMSHead,encrypted_PMS) = encrypted_PMS_msg in
  case encrypted_PMS of {[PMS]}s_PriKey in
  c(c_ChgCipherSpec).
  [ c_ChgCipherSpec is CHG_CIPHER_SPEC ]
  c(c_encrypted_Finish).
  rename MS = H(PMS,c_rand,s_rand) in
  rename KM = H(MS,c_rand,s_rand) in
  rename dummy1 = H(KM,C_WRITE_IV) in
  rename dummy2 = H(KM,S_WRITE_IV) in
  case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
  let (c_Finish,c_MAC) = c_Finish_and_MAC in
  [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
  let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
  [ final_Hash_MD5 is
    H((c_hello,s_hello_s_cert_s_hello_done,encrypted_PMS_msg),C_ROLE,MS,MD5) ]
  [ final_Hash_SHA is
    H((c_hello,s_hello_s_cert_s_hello_done,encrypted_PMS_msg),C_ROLE,MS,SHA) ]
  c_int<encrypted_PMS_msg>.
  c_int<c_ChgCipherSpec>.
  c_int<c_encrypted_Finish>.
  c_int(s_ChgCipherSpec).
  [ s_ChgCipherSpec is CHG_CIPHER_SPEC ]
  c<s_ChgCipherSpec>.
  /* (@pad) */
  c_int(s_encrypted_Finish).
  case s_encrypted_Finish of {s_Finish_and_MAC_and_pad}(KM,S_WRITE_KEY)~ in
  let (s_Finish_and_MAC, pad) = s_Finish_and_MAC_and_pad in
  rename DATA = (c_hello,s_hello_s_cert_s_hello_done,encrypted_PMS_msg,c_Finish) in
  rename local_s_finish = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
  rename local_s_MAC = H((KM,S_MAC_SEC)~,local_s_finish) in
  [ s_Finish_and_MAC is (local_s_finish,local_s_MAC) ]
  c<s_encrypted_Finish>.
  0
```

Figure 10. A possible Spi Calculus specification of a monitor for an SSL server.

at line 25M is possible, because we reasonably assumed that `s_PrivKey` is known; otherwise, it would have been impossible to perform that decryption, and thus to generate a sound monitor specification. When translation reaches line 30S, first the buffered messages are delivered to the server (lines 41M–43M), then the server ChangeCipherSpec message is received, checked and relayed on the public channel by the monitor (lines 44M–46M). The freshly created padding at line 33S is dropped (commented out at line 47M), then the server Finished message is received, checked, and relayed on the public channel at lines 48M–55M. It is essential that the random padding used to align the plaintext length to the block cipher size for encryption is explicitly represented in the server specification. Indeed, if the padding was not represented, the server Finished message would have had the form `{S_FINISH,H((KM,S_MAC_SEC);S_FINISH)}` `(KM,S_WRITE_KEY)~`, and it would have been in $dom(reconstructed)$. For this reason, after line 48M the monitor would have matched the `s_encrypted_Finish` variable against the reconstructed term. Although this would be correct at the specification level, the monitor implementation could never reconstruct the same encrypted Finished message, because it could not guess the server chosen random padding for that encryption. That is, the monitor is able to reconstruct the encrypted Finished message, modulo the server chosen random padding. By letting the padding be explicitly represented, the `{S_FINISH,H((KM,S_MAC_SEC);S_FINISH),pad}` `(KM,S_WRITE_KEY)~` term becomes not reconstructed (because `pad` is not known nor reconstructed), so the encrypted Finished message is dissected, and all of its parts, except for the padding which is discarded, are checked. One may argue that the same reasoning should be then applied to the client Finished message too. Although this could be possible, it is not needed. The server specification already prescribes to check the content of the client Finished message (and not, for example, to check it against a reconstructed term), and the monitor replicates the server checks; padding will be handled (and discarded) by the cryptographic function implementation. After the server Finished message is relayed over the public channel at line 55M, line 35S is translated. Since there are no buffered messages in the queue, the monitor terminates at line 56M.

B. Monitor Implementation

Once a Spi Calculus specification is available for the monitor of the SSL server role, the MDD approach can be used to generate a monitor implementation from its specification.[1] Note that the server specification is generic, because it is not bound to any specific server implementation. As such, the monitor specification is generic too, and its implementation will be able to black box monitor different SSL server implementations.

In order to generate the monitor implementation the spi2java framework [6] is used. Briefly, spi2java is a set of tools that allow to start from the formal Spi Calculus specification of a security protocol, and to get a semi-automatically generated interoperable Java implementation of the protocol actors. Note that, in the first place, spi2java was designed to allow MDD of security protocol actors, rather than of monitors. In this paper, we originally use spi2java to address a new problem: the generation of a monitor. In fact, the work in this paper overcomes one of the main limitation of the standard MDD approach: the impossibility to deal with legacy implementations. Indeed, by using spi2java to generate one implementation of a black box monitor for different legacy implementations of the same protocol role, an MDD approach is used to monitor correctness of several legacy implementations.

In order to generate an executable Java implementation of a Spi Calculus specification, some details that are not caught by the Spi Calculus specification must be added. That is, the Spi Calculus specification must be refined, before it can be translated into a Java application. In particular:

1) Spi Calculus is an untyped language, while Java is statically typed: for this reason any Spi Calculus term must be assigned a type before it can be translated into a Java object;
2) Spi Calculus cryptographic functions (for example a hashing $H(M)$) do not contain any information about the cryptographic algorithms to be used (e.g. MD5 or SHA-1 for hashing, or RSA or DSA for encryption), or their cryptographic parameters (such as IVs or block cipher size). One algorithm with its parameters must be chosen for each cryptographic term before generating the Java implementation;
3) Spi Calculus has no concept of data marshalling, which is necessary to achieve interoperability of the generated Java application.

As shown in figure 1, the spi2java framework assists the developer during the refinement and code generation steps. Once the verified Spi Calculus specification of the monitor is available, the spi2java refiner is used to automatically infer some refinement information from the specification: for example, if a name is used to perform input/output operations, it will be assigned the type “Channel”; or each $\{M\}_{K\sim}$ term will be assigned the type “Shared Key Ciphered”, with DES as default algorithm and some default constant IV. All inferred information is stored into an eSpi (extended Spi Calculus) document, which is coupled with the Spi Calculus specification. The spi2java refiner always outupts an eSpi document that can be directly used to generate an implementation, thus enabling agile prototyp-

[1]The SSL server monitor implementation generated in this work is available at: http://www.dai-arc.polito.it/dai-arc/manual/people/pironti/papers/monitoring/SSLMonitorImpl.tar.gz

ing. Nevertheless, the generated implementation will not be interoperable, because default cryptographic algorithms and marshalling functions are used. For this reason the developer can manually refine the generated eSpi document, for example by specifying that a "Channel" will actually be a "TCP/IP Channel", or by indicating the specific encryption algorithm and the IV for the term $\{M\}_{K}\ _{\sim}$, as prescribed by the protocol being implemented. The manually refined eSpi document is passed back to the spi2java refiner, that checks its coherence against the original Spi Calculus specification, and possibly infers new information from the user given one. This iterative refinement step can be repeated until the developer is satisfied with the obtained eSpi document, but usually only one iteration is enough.

Once the eSpi document is ready, it is passed along with the original Spi Calculus specification to the spi2java code generator, that automatically outputs the Java code implementing the given specification. The generated code only implements the "protocol logic", that is the code that simulates the Spi Calculus specification by coordinating input/output operations, cryptographic primitives and checks on received data. Dealing with Java sockets or the Java Cryptographic Architecture (JCA) is delegated to the SpiWrapper library, which is part of the spi2java framework. Each type that can be assigned to a Spi Calculus term in the eSpi document, is matched by a corresponding SpiWrapper class, that implements its behaviour. The SpiWrapper library allows to get a compact and readable generated code, that can be easily mapped back to the Spi Calculus specification. For example, line 2M of the monitor specification in figure 10 is translated as

```
/* c_0(c_hello_1). */
Pair c_hello_1 = (Pair)
  c_0.receive(new PairRecvClHello());
```

(each Spi Calculus term gets a suffix appended to make sure there is a unique identifier for that term). To improve readability, the spi2java code generator puts the translated Spi Calculus process as a Java comment. In this example, the Java variable `c_0` has type `TcpIpChannel`, which is a Java class included in the SpiWrapper library implementing a Spi Calculus channel using TCP/IP as transport layer. This class offers the `receive` method that allows the Spi Calculus input process to be easily implemented, by internally dealing with the Java sockets. The `c_hello_0` Java variable has type `Pair`, which implements the Spi Calculus pair. The `Pair` class offers the `getLeft` and `getRight` methods, allowing a straightforward implementation of the pair splitting process. The spi2java translation function is proven sound in [18].

In order to get interoperable implementations, the SpiWrapper library classes only deal with the *internal* representation of data. By extending the SpiWrapper classes, the developer can provide custom marshalling functions that transform the internal representation of data into the external one. If some information flow properties are satisfied, e.g. the marshalling functions cannot access secret data, it is possible to show [19] that any developer provided marshalling function implementation cannot compromise secrecy and authentication properties, under the Dolev-Yao attacker. In the eSpi document, the developer can specify, for each term of the Spi Calculus specification, the class that implements the marshalling functions. In order to enable agile prototyping, a default marshalling layer that uses Java serialization is provided.

In the SSL monitor case study, a two-tier marshalling layer has been implemented. Tier 1 handles the Record Layer protocol of SSL, while tier 2 handles the upper layer protocols. As hinted above, this solution makes the Spi Calculus specification independent of how messages are encapsulated, because they are handled in the marshalling layer, thus allowing to monitor a broader range of scenarios. When receiving a message from another agent, tier 1 parses one Record Layer message from the input stream, and its contained upper layer protocol messages are made available to tier 2. The latter implements the real marshalling functions, for example converting US-ASCII strings to and from Java String objects. When sending a message, tier 2 creates the upper layer message from the internal representation of data. Tier 1 buffers several upper layer messages, and encapsulates them into Record Layer messages, that are forwarded to other agents. Note that the marshalling layer functions only check that the packet format is correct. No control on the payload is needed: it will be checked by the automatically generated protocol logic.

The SSL protocol defines custom hashing algorithms, for instance to compute the MS from the PMS, or to compute the MAC value. In order to handle this, the corresponding Spi Calculus terms are hashing (e.g. lines 29M or 35M), and custom types, such as "SSL Hashing" or "SSL MAC", have been assigned to them in the eSpi specification. For each of these custom types, a corresponding SpiWrapper class has been created, implementing the custom algorithm. Another option could have been extending an existing JCA provider to support the custom algorithms. Then, one could have assigned the "Hashing" type to the Spi Calculus terms, passing the custom algorithm as cryptographic parameter, instead of the standard MD5 or SHA-1 algorithms.

Finally, it is worth pointing out some details about the IVs used by cryptographic operations (and declared at lines 31M and 32M). Each term of the form $\{M\}_{K^{\sim}}$ is typed "Shared Key Ciphered". This allows to specify the cryptographic algorithm (such as DES, 3DES, AES) and the IV. However, while in this example the cipher suite is known at compile time, the IV changes for each protocol run. The spi2java framework allows to resolve cryptographic algorithms and parameters both at compile time or at run time. If the parameter is to be resolved at compile time, the value of the parameter must be provided (e.g. AES for the symmetric encryption algorithm, or a constant value for the IV). If the parameter is to be resolved at run time, the identifier of another term of the Spi Calculus specification must be provided: the parameter value will be obtained by the content of the referred term, during execution. In the SSL case study, this feature is used for the IVs. For example, the $\{$`c_Finish_and_MAC`$\}$`(KM,C_WRITE_KEY)`$\tilde{}$ term uses the `H(KM,C_WRITE_IV)` term as IV. Technically, this feature enables support for cipher suite negotiation. However, as stated above, this would increase the specification complexity more than it would increase its significance, and is left for future work.

C. Experimental Results

This section shows and comments how the generated monitor described above has been used with different SSL server implementations, along with some performance considerations.

The monitor has been coupled in turn with three different SSL server implementations, namely OpenSSL[2] version 0.9.8g, GnuTLS[3] version 2.4.2 and JESSIE[4] version 1.0.1. OpenSSL and GnuTLS are implemented in C, while JESSIE is written in Java.

Since the online monitoring paradigm is used in this case study, the monitor is accepting connections on the standard SSL port (443), while the real server is started on another port (4433). Each time a client connects to the monitor, the latter opens a connection to the real server, starting data checking and forwarding, as explained above.

It is worth noting that switching the server implementation is straightforward. It is enough to shut down the running server implementation, and to start the other one. As long as the servers use the same private key, no action on the monitor is required; otherwise, it is enough to restart the monitor too, to let it use the new private key.

In order to generate protocol sessions, three SSL clients have been used with each server; namely the OpenSSL, GnuTLS, and JESSIE clients. During experiments, the monitor helped in spotting a bug in the JESSIE client: This client always sends packet of the SSL 3.1 version (better known as TLS 1.0), regardless of the negotiated version, that is SSL 3.0 in our scenario. The monitor correctly rejected all JESSIE client sessions, warning about the wrong protocol version.

When the OpenSSL or GnuTLS clients are used, the monitor correctly operates with all the three servers. In particular, safe sessions are successfully handled; conversely, when exchanged data are manually corrupted, they are recognized by the monitor and the session is aborted: corrupted data are never forwarded to the intended recipient.

[2]Available at: http://www.openssl.org/

[3]Available at: http://www.gnu.org/software/gnutls/

[4]Available at: http://www.nongnu.org/jessie/

Client	Server	No M [s]	M [s]	Oh [s]	Oh [%]
OpenSSL	OpenSSL	0.032	0.113	0.081	253.125
GnuTLS	OpenSSL	0.108	0.132	0.024	22.253
OpenSSL	GnuTLS	0.073	0.128	0.056	76.552
GnuTLS	GnuTLS	0.109	0.120	0.011	10.313
OpenSSL	JESSIE	0.158	0.172	0.014	8.986
GnuTLS	JESSIE	0.144	0.148	0.004	2.788

Table III
AVERAGE EXECUTION TIMES FOR PROTOCOL RUNS WITH AND WITHOUT MONITORING ENABLED.

In order to estimate the impact on performances of the online monitoring approach, execution times of correctly ended protocol sessions with and without the monitor have been measured. Communication between client, server and monitor happened over local sockets, so that no random network delays could be introduced; moreover system load was constant during test execution. Table III shows the average execution times for different client-server pairs, with and without monitor enabled. For each client-server pair, the average execution times have been computed over ten protocol runs. Column "No M" (No Monitoring) reports the average execution time, in seconds, without monitoring enabled, that is when the client directly connect to the server on port 4433. Column "M" (Monitoring) reports the average execution time, in seconds, with monitoring enabled, that is when the client connects on port 443 to the monitor, and the monitor forwards data to and from the server. The "Oh" columns show the overhead introduced by the monitor, in seconds and in percentage respectively. In four cases out of six, the monitor overhead is under 25 milliseconds, with a minimum of 4 milliseconds. In the other two cases, the overhead is between 50 and 80 milliseconds. From a practical point of view, this means that in a real distributed network, a client could hardly tell whether a monitor is present or not, since network times are orders of magnitude higher. Whether this overhead is acceptable on the server side depends on the number of sessions per seconds that must be handled. If the overhead is not acceptable, the offline monitoring paradigm can still be used.

V. CONCLUSION

The paper shows a practical methodology to design, develop and deploy monitors for legacy implementations of security protocols, without the need to modify the legacy implementations or to analyse their source code. To our knowledge, this is the first work that allows legacy implementations of security protocol agents to be black-box monitored.

The contribution of this paper includes the formal definition of a function that, given the specification of a security protocol actor, automatically generates the specification of a monitor for that actor, that stops incorrect sessions, and avoids malicious data to reach the monitored agent, and avoids agent's data to be unwillingly leaked to other parties. From the obtained monitor specification, an MDD approach is used to generate a monitor implementation; for this purpose, the spi2java framework has been originally reused, and some of its parts enhanced.

Finally, the practical usefulness of the proposed methodology has been shown, by implementing a monitor for the server role of the widely used SSL protocol. The same generated monitor implementation is able to monitor several different SSL server implementations against different clients, in a black-box way. The only needed information is the private key used by the server, in order to check message contents. By reporting session errors, the monitor even hinted us in finding a bug in an SSL client implementation. The overhead introduced by the monitor to check and forward messages is usually negligible. If the overhead is not acceptable, this paper also proposes an "offline" monitoring strategy that has no overhead and can still be useful to timely discover protocol attacks.

As future work, a general result about soundness of the monitor specification generating function would be useful. The soundness property should show that the generated monitor specification actually forwards only (and all) the protocol sessions that would be accepted by the agent's verified specification. This means that any wrong protocol agent implementation is safely monitored. Together with soundness proofs given by the spi2java framework, this would allow to get a sound monitor implementation, from the monitored agent's specification, down to the monitor implementation level.

Moreover, the monitor of the SSL case study could be tested against more SSL server implementations, and its code optimized, trying not to lose its soundness properties.

References

[1] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol," *Information Processing Letters*, vol. 56, no. 3, pp. 131–133, 1995.

[2] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *USENIX Workshop on Electronic Commerce (EC-96)*, 1996, pp. 29–40.

[3] OpenSSL Security Team, "Incorrect checks for malformed signatures," Jan. 2009. [Online]. Available: http://www.openssl.org/news/secadv_20090107.txt

[4] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *CRYPTO*, 2005, pp. 17–36.

[5] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

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

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

[8] E. Hubbers, M. Oostdijk, and E. Poll, "Implementing a formally verifiable security protocol in Java Card," in *Security in Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 2802, 2003, pp. 213–226.

[9] J. Jürjens and M. Yampolskiy, "Code security analysis with assertions," in *ASE*, 2005, pp. 392–395.

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

[11] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *Computer Security Foundations Symposium, IEEE*, vol. 0, pp. 17–32, 2008.

[12] S. Schneider, "Security properties and CSP," in *IEEE Symposium on Security and Privacy*, 1996, pp. 174–187.

[13] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[14] V. L. Voydock and S. T. Kent, "Security mechanisms in high-level network protocols," *ACM Computing Surveys*, vol. 15, no. 2, pp. 135–171, 1983.

[15] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The Spi Calculus," Research Report 149, 1998.

[16] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol version 1.2," RFC 5246 (Proposed Standard), Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[17] L. Viganò, "Automated security protocol analysis with the AVISPA tool," *Electronic Notes on Theoretical Computer Science*, vol. 155, pp. 61–86, 2006.

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

[19] ——, "Soundness conditions for message encoding abstractions in formal security protocol models," in *Availability, Reliability and Security*, 2008, pp. 72–79.