Formally-Based Black-Box Monitoring of Security Protocols*

Alfredo Pironti[1] and Jan Jürjens[2]

[1] Politecnico di Torino
Turin, Italy
http://alfredo.pironti.eu/research
[2] TU Dortmund and Fraunhofer ISST
Dortmund, Germany
http://jurjens.de/jan

Abstract. In the challenge of ensuring the correct behaviour of legacy implementations of security protocols, a formally-based approach is presented to design and implement monitors that stop insecure protocol runs executed by such legacy implementations, without the need of their source code. We validate the approach at a case study about monitoring several SSL legacy implementations. Recently, a security bug has been found in the widely deployed OpenSSL client; our case study shows that our monitor correctly stops the protocol runs otherwise allowed by the faulty OpenSSL client. Moreover, our monitoring approach allowed us to detect a new flaw in another open source SSL client implementation.

1 Introduction

Despite being very concise, cryptographic protocols are quite difficult to get right, because of the concurrent nature of the distributed environment and the presence of an active, non-deterministic attacker. Increasing the confidence in the correctness of security protocol implementations is thus important for the dependability of software systems. In general exhaustive testing is infeasible, and for a motivated attacker one remaining vulnerability may be enough to successfully attack a system. In this paper, we focus in particular on assessing the correctness of legacy implementations, rather than on the development of correct new implementations. Indeed, it is often the case in practice that a legacy implementation is already in use which cannot be substituted by a new one: for example, when the legacy implementation is strictly coupled with the rest of the information system, making a switch very costly.

In this context, our proposed approach is based on black-box monitoring of legacy security protocols implementations. Using the Dolev-Yao [6] model, we assume cryptographic functions to be correct, and concentrate on their usage within the cryptographic protocols. Moreover, we concentrate on implementations of security protocol actors, rather than on the high level specifications of

* This research was partially supported by the EU project SecureChange (ICT-FET-231101).

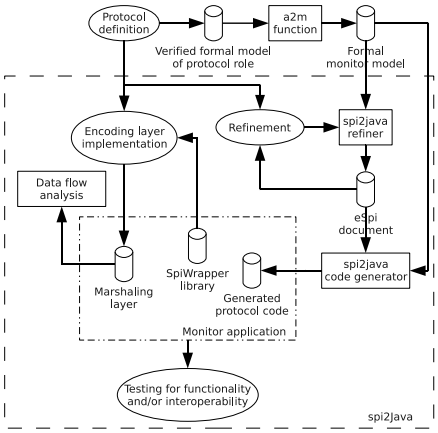


Fig. 1: Monitor design and development methodology.

such security protocols. That is, we assume that a given protocol specification is secure (which can be proven using existing tools); instead, by monitoring it, we want to asses that a given implementation of one protocol's role is correct with respect to its specification, and it is resilient to Dolev-Yao attacks.

The overall methodology is depicted in figure 1. Given the protocol definition, a specification for one agent is manually derived. By using the "agent to monitor" ($a2m$) function introduced in this paper, a monitor specification for that protocol role is automatically generated. Then the monitor implementation is obtained by using the model driven development framework called spi2java [13], represented by the dashed box in the figure. The spi2java internals will be discussed later on in the paper. The monitor application is finally ran together with the monitored protocol role implementation (not shown in the picture).

A monitor implementation differs from a fresh implementation of a security protocol, because it does not execute protocol sessions on behalf of its users. The monitor instead observes protocol sessions started by the legacy implementations, in order to recognize and stop incorrect sessions, in circumstances where the legacy implementations cannot be replaced.

For performance trade-offs, monitoring can be performed either "online" or "offline". In the first case, all messages are first checked by the monitor, and then forwarded to the intended recipient only if they are safe. In the second case, all messages exchanged by the monitored application are logged, and then fed to the monitor for later inspection. The online paradigm prevents a security property to be violated, because protocol executions are stopped as soon as an unexpected message is detected by the monitor, before it reaches the intended recipient. However, online monitoring may introduce some latency. The offline paradigm does not introduce any latency and is still useful to recognize compromised protocol sessions later, which can limit the damage of an attack. 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 limiting the time span of the fraud.

In this paper, the main goal of monitors is to detect, stop and report incorrect protocol runs. Monitors are not designed for example to assist one in forensic diagnosis after an attack has been found.

The monitoring is "black-box" in that 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. Thus any legacy implementation can still be used in production as is, while being monitored. The correctness of this approach depends on the correctness of the generated monitor. Our approach leverages formal methods in the derivation of the monitor implementation, so that a trustworthy monitor is obtained.

Note that this approach can be exploited during the testing phase as well: One can run an arbitrary number of simulated protocol sessions in a testing environment, and use the monitor to check for the correct behaviour.

In order to validate the proposed approach, a monitor for the SSL protocol is presented. The generated monitor stops incorrect sessions that could, for example, exploit a recently found flaw in the OpenSSL implementation.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 illustrates the formal background used in the paper. Section 4 describes the function translating a Spi Calculus protocol agent's specification into a monitor specification for that agent. Then section 5 shows the SSL protocol case study. Finally section 6 concludes.

For brevity, this paper mainly concentrates on the description of the proposed approach and on its validation by means of a real-life size case study. An extended version of this paper that includes all the formal definitions and the source code of the presented case study and other case studies can be found in [12].

2 Related Work

Several attempts have been made to check that a protocol role implementation is correct w.r.t. its specification which can be grouped in four main categories: (1) Model Driven Development (MDD); (2) Static Code Verification; (3) Refinement Types; (4) Online Monitoring and Intrusion Detection Systems (IDSs).

The first approach consists of designing and verifying a formal, high-level model of a security protocol and to semi-automatically derive an implementation that satisfies the same security properties of the formal model [8,9,13]. However, it has the drawback of not handling legacy implementations of security protocols.

The second approach starts from the source code of an existing implementation, and extracts a formal model which is verified for the desired security properties [5, 11]. In principle, this approach can deal with legacy implementations, but their source code must be available, which is not always the case.

The third approach proves security properties of an implementation by means of a special kind of type checking on its source code [4]. Working on the source code, it shares the same advantages and drawbacks of the second approach.

The fourth approach comes in two versions. With online monitoring, the source code of an existing implementation is instrumented with assertions: program execution is stopped if any assertion fails at runtime [3]. Besides requiring the source code, the legacy implementation must be substituted with the instrumented one, which may not always be the case. IDSs are systems that monitor network traffic and compare it against known attack patterns, or expected average network traffic. By working on averages, in order not to miss real attacks, IDSs often report false positive warnings. In order to reduce them, sometimes the source code of the monitored implementation is also instrumented [10], sharing the same advantages and drawback of online monitoring.

Another branch of research focused on security wrappers for legacy implementations. In [1], a formal approach that uses security wrappers as firewalls with an encrypting tunnel is described. Any communication that crosses some security boundary is automatically and transparently encrypted by the wrappers. That is, the wrappers *add* security to a distributed legacy system. In our approach, the monitor *enforces* the security already present in the system. Technically, our approach derives a monitor based on the security requirements already present in the legacy system, instead of adding a boilerplate layer of security.

Analogously, in [7] wrappers are used, among other things, to transparently encrypt local files accessed by library calls. However, distributed environments are not taken into account. Finally, in [17] wrappers are used to harden software libraries. However, cryptography and distributed systems are not considered, and the approach is test-driven, rather than formally based.

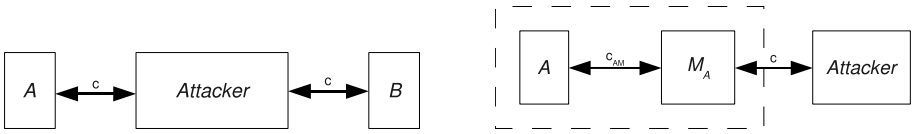
3 Formal Background

3.1 Network Model

Many network models have been proposed in the Dolev-Yao setting. For example, sometimes the network is represented as a separate process [16]; the attacker is connected to this network, and can eavesdrop, drop and modify messages, or forge new ones. In other cases, the attacker is the medium [15], and honest agents can only communicate *through* the attacker. Even more detailed network models have been developed [19], 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.

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. In this paper, we focus on a simple scenario that is usually found in practice, and is depicted in figure 2(a): the attacker is the medium, and every protocol agent communicates over a single insecure channel c, and private channels are not allowed. Moreover, agents are sequential and non-recursive.

Let us define A as the (correct) model of the agent to be monitored, and M_A as the model of its monitor. When the monitor is present, A communicates



(a) Agents A and B with the attacker.

(b) Agent A monitored by M_A and the attacker.

Fig. 2: The network model.

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

(a) Spi Calculus terms.

$P, Q, R ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$P \mid Q$	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

(b) Spi Calculus processes.

Table 1: Spi Calculus grammar.

with M_A only, through the use of a private channel c_{AM}, while M_A is directly connected to the attacker by channel c, as depicted in figure 2(b). The dashed box denotes that A and M_A run in the same environment, for example they run on the same system with same privileges. Note that in A channel c is in fact renamed to c_{AM}.

3.2 The Spi Calculus

In this paper, the formal models are expressed in Spi Calculus [2]. Spi Calculus is amenable for our approach because it is a domain specific language tailored at expressing the behaviour of single security protocol agents, where checks on received data must be explicitly specified. Thus, from the Spi Calculus specifications of protocol agents, the $a2m$ function can derive precise and complete specifications of their monitors.

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 1(a) contains the terms defined by the Spi Calculus, while table 1(b) 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. If

a variable or a name is not bound, then it is free. The $M^{\sim}$ term represents a symmetric key built from key material M, and $\{M\}_N$ represents the encryption of the plaintext M with the symmetric key N, while $H(M)$ represents the result of hashing M. The M^+ and M^- terms represent the public and private part of the keypair M respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively.

Informally, the $\overline{M}\langle N\rangle.P$ process sends message N on channel M, and then behaves like P, 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 are never used in its specification.

4 The Monitor Generation Function

The $a2m$ function translates a sequential protocol role specification into a monitor specification for that role; formally, $M_A \triangleq a2m(A)$. For brevity, $a2m$ is only informally presented here, by means of a running example. Formal definitions can be found in [12].

Before introducing the function, 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 its subterms are known or reconstructed. For example, suppose M is known through $\_M$ and $H(N)$ is known through $\_H(N)$. It is the case that $(H(N), M)$ is reconstructed by $(\_H(N), \_M)$. Note that, as terms become known, other terms may become reconstructed too. In the example given above, if M was not known, then it was not possible to reconstruct $(H(N), M)$;

```
1a: A(M,k) :=
2a:  cAM<{M}k>.

3a:  cAM(x).
4a:  [x is H(M)]
5a:  0
```

(a) Agent A specification.

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

(b) Monitor specification derived from agent A one.

Fig. 3: Example specification of agent A along with its derived monitor M_A.

however, if later M became known (for example, because it was sent over a channel), then $(H(N), M)$ would become reconstructed.

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. Especially in the online mode, this overhead may not be acceptable. In order to drop this check, it is needed to assume that the random value generator in the monitored agent is correctly implemented. Also note that there may be cases where the monitor has not enough information to properly check protocol execution. These cases are recognised by the $a2m$ function, so that an error state is reached, and no unsound monitor is generated.

The $a2m$ function behaviour is now described by means of a running example. Agent A sends some data M encrypted by the key k to the other party, and expects to receive the hash of the plaintext, that is $H(M)$. Note that the example focuses on the way the $a2m$ function operates, rather than on monitoring a security protocol, so no security claim is meant to be made on this protocol. Figure 3(a) shows the specification for agent A, and figure 3(b) 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).

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. 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 the key k used by A is known by the monitor (through the variable k), while M is not known (for example, because the monitor cannot access those data); however $H(M)$ is known through $\_H(M)$, that is the monitor has access to the memory location where $H(M)$ is stored, and this value is bound to the variable $\_H(M)$ in the monitor.

When line 2a is translated by $a2m$, lines 2m–5m are produced. The data sent by A are received by the monitor at line 2m, and stored in variable $\_\{M\}_k$. Afterwards, some checks on the received value are added by the $a2m$ function. In general, each time a new message is received from the monitored application, it or its parts are checked against their expected (known or reconstructed) values. In this case, since $\{M\}_k$ is not known (by hypothesis) or reconstructed (because M is not known or reconstructed), it cannot be directly compared against the known or reconstructed value, so it is exploded into its components. As $\{M\}_k$ is an encryption and k is known, the decryption case process is generated at line 3m, binding $\_M$ to the value of the plaintext, that should be M. Since M is not known or reconstructed, and it is a name, $\_M$ cannot be dissected any more; instead, M becomes known through $\_M$, in other words, the term stored in $\_M$ is assumed by the monitor to be the correct term for M. Note that, before M was known through $\_M$, $H(M)$ was known through $\_H(M)$, but it was not reconstructed. After the assignment of $\_M$, $H(M)$ becomes reconstructed by $H(\_M)$ too. The match process at line 4m ensures that known and reconstructed values for the same term are actually the same.

After all the possible checks are performed on the received data, they are forwarded to the attacker at line 5m. Then, line 3a is translated into line 6m. When translating an input process, the monitor receives message x from the attacker on behalf of the agent and buffers it; x is said to be known through x itself. Then the monitor behaves according to what is done by the agent (usually checks on the received data, as it is the case in the running example). 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.

Line 4a is then translated into line 7m, and 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.

5 An SSL Server Monitor Example

5.1 Monitor Specification

As shown in figure 1, in order to get the monitor specification, a Spi Calculus specification of the server role for the SSL protocol is needed. The full SSL protocol is rather complex: many scenarios are possible, and different sets of cryptographic algorithms (called *ciphersuites*) can be negotiated. For simplicity, this example considers only one scenario of the SSL protocol version 3.0, where 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.

In this paper, the chosen scenario requires the server to use a DSA certificate for authentication and data signature. Although RSA certificates are more com-

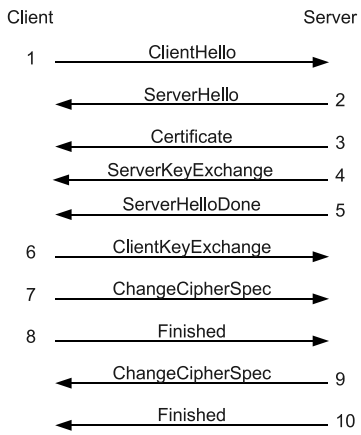


Fig. 4: Typical SSL scenario.

mon in SSL, using a DSA certificate allowed us to stress a bug in the OpenSSL implementation, showing that the monitor can actively drop malicious sessions that would be otherwise accepted as genuine by the flawed OpenSSL implementation. The more common RSA scenario has been validated through a dedicated case-study. However, it is not reported here for brevity; it can be found in [12].

The SSL scenario considered in this example is depicted informally in figure 4, while figure 5 shows a possible Spi Calculus specification of a server for the chosen scenario. The ASCII syntax of Spi Calculus is used in figure 5. Also, in order to model the Diffie-Hellman (DH) key exchange, the $EXP(L, M, N)$ term is added, which expresses the modular exponentiation $L^M \mod N$, along with the equation $EXP(EXP(g, a, p), b, p) = EXP(EXP(g, b, p), a, p)$.

To make the specification more readable, lists of terms like (A, B, C) are added as syntactic sugar, and they are translated into left associated nested pairs, like $((A, B), C)$; a `rename n = M in P` process is introduced too, that renames the term M to n and then behaves like P.

The first message is the ClientHello, 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 the server specification, the ClientHello message is received and split into its parts at line 2S. In the chosen scenario, 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_DHE_DSS_3DES_EDE_CBC_SHA`, with no compression enabled. All these constraints are checked at lines 3S–4S.

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. The server random value

```
 1S Server() :=
 2S   c(c_hello).  let (c_version,c_rand,c_SID,c_ciph_suite,c_comp_method) = c_hello in
 3S   [ c_version is THREE_DOT_ZERO ]  [ c_SID is ZERO ]
 4S   [ c_ciph_suite is SSL_DHE_DSS_3DES_EDE_CBC_SHA ]  [ c_comp_method is comp_NULL ]
 5S   (@s_rand)  (@SID)
 6S   rename S_HELLO = (THREE_DOT_ZERO,s_rand,SID,SSL_DHE_DSS_3DES_EDE_CBC_SHA,comp_NULL) in
 7S   (@DH_s_pri)  rename DH_s_pub = EXP(DH_Q,DH_s_pri,DH_P) in
 8S   rename S_KEX = ((DH_P,DH_Q,DH_s_pub),[{H(c_rand,s_rand,(DH_P,DH_Q,DH_s_pub))}]s_PriKey) in
 9S   c<S_HELLO,S_CERT,S_KEX,S_HELLO_DONE>.
10S   c(c_kex).  let (c_kexHead,DH_c_pub) = c_kex in  rename PMS = EXP(DH_c_pub,DH_s_pri,DH_P) in
11S   rename MS = H(PMS,c_rand,s_rand) in  rename KM = H(MS,c_rand,s_rand) in
12S   rename c_w_IV = H(KM,C_WRITE_IV) in  rename s_w_IV = H(KM,S_WRITE_IV) in
13S   c(c_ChgCipherSpec).  [ c_ChgCipherSpec is CHG_CIPH_SPEC ]
14S   c(c_encrypted_Finish).  case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
15S   let (c_Finish,c_MAC) = c_Finish_and_MAC in  [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
16S   let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
17S   [ final_Hash_MD5 is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,MD5) ]
18S   [ final_Hash_SHA is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,SHA) ]
19S   c<CHG_CIPH_SPEC>.
20S   rename DATA = (c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex,c_Finish) in
21S   rename S_FINISH = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
22S   (@pad)  c<{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH),pad}(KM,S_WRITE_KEY)~>.
23S   0
```

Fig. 5: A possible Spi Calculus specification of an SSL server.

and the fresh session ID are generated at line 5S, then the ServerHello message is declared at line 6S. Again, in the chosen scenario, the server chooses protocol version 3.0, and always selects the `SSL_DHE_DSS_3DES_EDE_CBC_SHA` cipher suite, with no compression enabled. Then the server sends the Certificate message to the client: in the chosen scenario, this message contains a DSA certificate chain for the server's public key, that authenticates the server.

In the fourth message, named ServerKeyExchange, the server sends the DH key exchange parameters to the client, and digitally signs them with its public key. In the server specification, the DH server secret value `DH_s_pri` and the corresponding public value `DH_s_pub` are computed at line 7S. Then, at line 8S, the ServerKeyExchange message is declared: it consists of the server DH parameters, along with a digital signature of the DH parameters and the client and server random values, in order to ensure signature freshness.

The fifth 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. In the server specification, these four messages are sent all at once at line 9S.

In the sixth message, the client replies with the ClientKeyExchange message, that contains the client's DH public value. Note that there is no digital signature in this message, since the client is not authenticated. In the server specification the ClientKeyExchange is received at line 10S, where the payload is split from the message header too. Both client and server derive a shared secret from the DH key exchange. This shared secret is called Premaster Secret (PMS), and it is used by both parties to derive some shared secrets used for symmetric encryption of application data. The PMS is computed by the server at line 10S. 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) (line 11S). Finally, different subsets of bytes of the KM are used as shared secrets and as initialization vectors (IVs). Note that IVs, that are extracted at line 12S, are never referenced in the specification. They will be used as cryptographic parameters for subsequent encryptions, during the code generation step, explained in section 5.2.

The seventh message is the ChangeCipherSpec, received and checked at line 13S. This message contains no data, but signals the server that the client will start using the negotiated cipher suite from the next message on.

The client then sends its Finished message. 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 client Finished message is received and decrypted at line 14S. 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. At line 15S the MAC is extracted from the plaintext, and verified. The unencrypted content of the Finished message 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. Both versions of the final hash are extracted and checked at lines 16S–18S. As Spi Calculus does not support different algorithms for the same hash, they are distinguished by a marker (`MD5` and `SHA` respectively) as the last hash argument, making them syntactically different.

Then the server sends its ChangeCipherSpec message to client (line 19S), and its Finished message, that comes with MAC and encryption too (lines 20S–22S). Encryption requires random padding to align the plaintext length to the cipher block size. This random padding must be explicitly represented in the server specification, so that the monitor can recognise and discard it, and only check the real plaintext. Otherwise the monitor would try to locally reconstruct the encryption, but it would always fail, because it could not guess the padding. The protocol handshake is now complete, and next messages will contain secured application data.

In order to verify any security property on this specification, the full SSL specification, including the client and protocol sessions instantiations is required. However, this is outside the scope of this paper; SSL security properties have already been verified, for example, by the AVISPA project [18]. Here it is assumed instead that the specification of the server is correct, and thus secure, so that the monitoring approach can be shown.

The $a2m$ function described in section 4 is applied on the server specification, in order to obtain the online monitor specification for the server role. For brevity,

the resulting specification is not shown here. It can be found, along with more implementation details, in [12]. It is assumed that the monitor has access to the server private DH value, 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. Often, the server will generate a fresh DH private value for each session, and it will usually only store it in memory. In general, with some effort the monitor will be able to directly read this secret from the legacy application memory, without the need of the source code. Nevertheless, in a testing environment, if the source code of the monitored application happens to be available, it is possible to patch the monitored application, so that it explicitly communicates the DH private value to the monitor. Indeed, this is reasonable because the monitor is part of the trusted system, and is actually more trusted than the monitored application.

5.2 Monitor Implementation

The source code of the monitor implementation can be found in [12]. In order to generate the monitor implementation, the spi2java MDD framework is used [13]. Briefly, spi2java is a toolchain that, starting from the formal Spi Calculus specification of a security protocol, semi-automatically derives an interoperable Java implementation of the protocol actors. In the first place, spi2java was designed to generate security protocol actors, rather than monitors. In this paper, we originally reuse spi2java to generate a monitor.

In order to generate an executable Java implementation of a Spi Calculus specification, some details that are not contained in 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.

As shown in figure 1, the spi2java framework assists the developer during the refinement and code generation steps. The spi2java refiner is used to automatically infer some refinement information from the given specification. All inferred information is stored into an eSpi (extended Spi Calculus) document, which is coupled with the Spi Calculus specification. The developer can manually refine the generated eSpi document; 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 one iteration is enough.

The obtained eSpi document and the original Spi Calculus specification are passed to the spi2java code generator that automatically outputs the Java code implementing the given specification. The generated code 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. The SpiWrapper library allows the generated code to be compact

and readable, so that it can be easily mapped back to the Spi Calculus specification. For example, the monitor specification corresponding to line 2S of the server specification in figure 5 is translated as

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

(each Spi Calculus term name is mangled to make sure there is a unique Java identifier for that term). To improve readability, the spi2java code generator outputs the translated Spi Calculus process as a Java comment too. 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_1` 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 [14].

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.

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. 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. Analogous reasoning applies when sending a message. 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. For each of them, a corresponding SpiWrapper class has been created, implementing the custom algorithm. Moreover, the spi2java framework has been extended to support the modular exponentiation, so that DH key exchange can be supported.

Finally, it is worth pointing out some details about the IVs used by cryptographic operations (declared in the server specification at line 12S). For each term of the form $\{M\}_K$, the eSpi document allows its cryptographic algorithm (such as DES, 3DES, AES) and its IV to be specified. However, the IV is only known at run time. The spi2java framework allows cryptographic algorithms and parameters to be resolved either 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)~` 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.

5.3 Experimental Results

The monitor has been coupled in turn with three different SSL server implementations, namely OpenSSL[3] version 0.9.8j, GnuTLS[4] version 2.4.2 and JESSIE[5] version 1.0.1.

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. In the testing scenario, assuming that the server communicates its private DH value to the monitor, it is enough to shut down the running server implementation, and to start the other one; the monitor implementation remains the same, and no action on the monitor is required. Otherwise, it is enough to restart the monitor too, enabling the correct plugin that gathers the private DH value from the legacy application memory. In other words, in a production scenario, the same monitor implementation can handle several different legacy server implementations; in the monitor, the only server-dependent part is the plugin that reads the DH secret value from the server application memory.

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, reporting 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.

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. Thus, performances regarding the JESSIE client are not reported, as no correct session could be completed, due to the

[3] Available at: http://www.openssl.org/
[4] Available at: http://www.gnu.org/software/gnutls/
[5] Available at: http://www.nongnu.org/jessie/

Client	Server	No Monitor [s]	Monitor [s]	Overhead [s]	Overhead [%]
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 2: Average execution times for protocol runs with and without monitoring.

discovered bug. That is, the measured performances all correspond to valid executions of the protocol only. 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 2 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. Columns "No Monitor" and "Monitor" report the average execution times, in seconds, without and with monitoring enabled respectively. When monitoring is not enabled, the clients directly connect to the server on port 4433. The "Overhead" 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. From a practical point of view, a client communicating through a real distributed network could hardly tell whether a monitor is present or not, since network times are orders of magnitude higher. On the other hand, in the worst cases online monitoring can slow down the server machine up to 2.5 times. 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.

The OpenSSL security flaw. Recently, the client side of the OpenSSL library prior to version 0.9.8j has been discovered flawed, such that in principle it could treat a malformed DSA certificate as a good one rather than as an error.[6] By inspecting the flawed code, we were able to forge such malformed certificate that exploited the affected versions. This malformed DSA certificate must have the q parameter one byte longer than expected. Up to our knowledge, this is the first documented and repeatable exploit for this flaw.

Without monitoring enabled, we generated protocol sessions between an SSL server sending the offending certificate, and both OpenSSL clients version 0.9.8i (flawed) and 0.9.8j (fixed). By using the `-state` command line argument, it is possible to conclude that the 0.9.8i version completes the handshake by reaching the "read finished A" state (after message 10 in figure 4); while the 0.9.8j version correctly reports an "handshake failure" error at state "read server certificate A", that is immediately after message 3 in figure 4.

[6] http://www.openssl.org/news/secadv_20090107.txt

When monitoring is enabled, the malformed server certificate is passed to the monitor as an input parameter, that is, the server certificate is *known* by the monitor. In this case the monitor actually refuses to start. Indeed, when loading the server certificate, the monitor spots that it is malformed, and does not allow any session to be even started. If we drop the assumption that the monitor knows the server certificate, then the monitor starts, and checks the server certificate when it is received over the network. During these checks, the malformed certificate is found, and the session is dropped, before the server Certificate message is forwarded to the client. This prevents the aforecited flaw to be exploited on OpenSSL version 0.9.8i.

6 Conclusion

The paper shows a formally-based yet 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.

This paper introduces a function that, given the specification of a security protocol actor, automatically generates the specification of a monitor that stops incorrect sessions, without rising false positive alarms. 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 proposed methodology has been validated by implementing a monitor starting from the server role of the widely used SSL protocol. Core insights gained from conducting the SSL case study include that the same generated monitor implementation can in fact monitor several different SSL server implementations against different clients, in a black-box way. The only needed information is the private Diffie-Hellman key used by the server, in order to check message contents. Moreover, by reporting session errors, the monitor effectively helped us in finding a bug in an open source SSL client implementation.

The "online" monitoring paradigm proved useful in avoiding protocol violations, for example by stopping malicious data that would have otherwise exploited a known flaw of the widely deployed OpenSSL client. 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. Together with the soundness proofs of the spi2java framework, this would produce a sound monitor implementation, directly from the monitored agent's specification.

References

1. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. Information and Computation 174(1), 37–83 (2002)
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The Spi Calculus. Digital Research Report 149 (1998)
3. Bauer, A., Jürjens, J.: Security protocols, properties, and their monitoring. In: International Workshop on Software Engineering for Secure Systems. pp. 33–40 (2008)
4. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: Computer Security Foundations Symposium, IEEE. pp. 17–32 (2008)
5. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. In: Computer Security Foundations Workshop. pp. 139–152 (2006)
6. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
7. Fraser, T., Badger, L., Feldman, M.: Hardening COTS software with generic software wrappers. In: IEEE Symposium on Security and Privacy. pp. 2–16 (1999)
8. Hubbers, E., Oostdijk, M., Poll, E.: Implementing a formally verifiable security protocol in Java Card. In: Security in Pervasive Computing. Lecture Notes in Computer Science, vol. 2802, pp. 213–226 (2003)
9. Jeon, C.W., Kim, I.G., Choi, J.Y.: Automatic generation of the C# code for security protocols verified with Casper/FDR. In: International Conference on Advanced Information Networking and Applications. pp. 507–510 (2005)
10. Joglekar, S.P., Tate, S.R.: ProtoMon: Embedded monitors for cryptographic protocol intrusion detection and prevention. Journal of Universal Computer Science 11(1), 83–103 (2005)
11. Jürjens, J., Yampolskiy, M.: Code security analysis with assertions. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 392–395 (2005)
12. Pironti, A., Jürjens, J.: Online resources about black-box monitoring, available at: http://alfredo.pironti.eu/research/projects/monitoring/
13. Pironti, A., Sisto, R.: An experiment in interoperable cryptographic protocol implementation using automatic code generation. In: IEEE Symposium on Computers and Communications. pp. 839–844 (2007)
14. Pironti, A., Sisto, R.: Provably correct Java implementations of Spi Calculus security protocols specifications. Computers & Security (2009), in Press
15. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR (1997)
16. Schneider, S.: Security properties and CSP. In: IEEE Symposium on Security and Privacy. pp. 174–187 (1996)
17. Süßkraut, M., Fetzer, C.: Robustness and security hardening of COTS software libraries. In: IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 61–71 (2007)
18. Viganò, L.: Automated security protocol analysis with the AVISPA tool. Electronic Notes on Theoretical Computer Science 155, 61–86 (2006)
19. Voydock, V.L., Kent, S.T.: Security mechanisms in high-level network protocols. ACM Computing Surveys 15(2), 135–171 (1983)