Soundness Conditions for Message Encoding Abstractions in Formal Security Protocol Models

Alfredo Pironti and Riccardo Sisto
Politecnico di Torino
Dip. di Automatica e Informatica
c.so Duca degli Abruzzi 24, I-10129 Torino (Italy)
e-mail: {alfredo.pironti, riccardo.sisto}@polito.it

Abstract

In formal methods, security protocols are usually modeled with a high level of abstraction. In particular, marshalling/unmarshalling operations on transmitted messages are generally abstracted away. However, in real applications, errors in this protocol component could be exploited to break protocol security.

In order to solve this issue, this paper formally shows that, under some constraints checkable on sequential code, if an abstract protocol model is secure, then a refined model, which takes into account a wide class of possible implementations of the marshalling/unmarshalling operations, is implied to be secure too. The paper also indicates possible exploitations of this result.

1. Introduction

In the last years, several techniques have been developed to formally analyze abstract models of security protocols where messages are represented as instances of high level abstract data types. One question that arises is how to get assurance about the fact that the logical correctness of an abstract protocol is indeed preserved when concrete versions of the protocol are defined and when their implementations are developed using programming languages. In general, security faults not present in an abstract protocol specification might be introduced when adding implementation details.

Recently, some work has been started in the direction of bridging the gap that exists between abstract formal models and their concrete counterparts. For example, some researchers have been working towards refining the models of cryptography (e.g. [9, 2, 1]). Another research line addresses the problem of ensuring that the formal model used for the analysis of a protocol is a safe abstraction of the protocol implementation, under the assumption that the cryptographic and communication libraries used by the implementation behave as specified by their ideal abstract Dolev-Yao [5] models. In particular, two different strategies have been explored, namely automatic code generation from abstract models ([11, 13]) and automatic model extraction from implementation code ([4, 8, 6, 3]).

Methods based on automatic code generation start from a high-level, formally verified, specification of the protocol, which abstracts away from details about cryptographic and communication operations and binary data representations, and fill the semantic gap between formal specification and implementation, guided by implementation choices provided by the user.

Methods based on automatic model extraction start from an already existing, full blown implementation code, from which an abstract model is extracted and formally verified. One of the things that can be observed looking at the results reported in [4, 8, 3], is that the part of the extracted formal model that describes message encoding and decoding operations can be quite complex, much more complex than the abstract protocol model itself.

This paper presents sufficient conditions under which the detailed models of marshalling and unmarshalling operations on transmitted messages can be soundly abstracted by simpler models or assumptions.

The first step is the definition of simple formal models of these data encoding and decoding transformations. Such models are general, in the sense that they do not describe specific implementations, but rather they capture only some general assumptions that are made on implementations. Verifying a real protocol implementation can thus be reduced to verifying that the protocol implementation fulfils the assumptions made and verifying that the model satisfies the required security properties. Of course, this approach is advantageous with respect to other approaches, provided that the general models are simple enough and the assumptions made are easy to be checked on real protocol implementations.

The second step that is made is to show that these models can be further simplified, using fault-preserving transformations like the ones introduced in [7]. The result that is finally obtained is that, under some further simple assumptions, the protocol models including implementation details can be transformed back into the original abstract protocol models without implementation details, and the classical security faults (secrecy and authentication) are preserved in this transformation. This means that, provided all the assumptions we made are verified on a given implementation, the formal model of the implementation details can be safely abstracted away in verifying the desired security properties.

The remainder of the paper is organized as follows. Section 2 introduces the notation and the modelling approach, based on CSP, that is used to reason on security protocols throughout the paper. Section 3 introduces detailed security protocol models including marshalling and unmarshalling operations, and the conditions under which they can be abstracted. Section 4 discusses about the application of the obtained results through an example, and section 5 concludes.

2. Abstract Protocol Models and Notation

The formalism used in this paper is based on CSP [12], and the datatype definitions and protocol models are an extension of the ones used in [7]. Essentially, they follow the Dolev-Yao approach [5]. The main extension w.r.t. [7] is an added support for non-atomic keys. This extension enables to model protocols where the key is constructed from non-atomic data. The new datatype is defined as

Message ::=
ATOM *Atom* | PAIR *Message Message* |
SHKEY *Message* | PUBKEY *Message* |
PRIKEY *Message* | HASH *Message* |
SHKEYENCRYPT *Message* SHKEY *Message* |
PUBKEYENCRYPT *Message* PUBKEY *Message* |
PRIKEYENCRYPT *Message* PRIKEY *Message*.

This definition has been developed using the following guidelines: First, each key is typed. It is possible to obtain a key from generic material (that is, any generic *Message*). Then, there is no longer need for the inverse K^{-1} of a key K. Indeed, the key construction operators PUBKEY and PRIKEY fulfil this role. Finally, no new types are added in order to represent encoding parameters, because the idea is to have a single datatype that can be used to model a protocol at different detail levels.

In order to get better reading for processes, the following syntactic sugar is also provided:

Message	Symbol
PAIR M M'	(M, M')
SHKEY M	$M^{\sim}$
PUBKEY M	M^{+}
PRIKEY M	M^{-}
HASH M	$H(M)$
SHKEYENCRYPT M SHKEY K	$\{M\}_{K^{\sim}}$
PUBKEYENCRYPT M PUBKEY K	$\{M\}_{K^{+}}$
PRIKEYENCRYPT M PRIKEY K	$\{M\}_{K^{-}}$

Once the datatype is defined, it is also necessary to update the intruder knowledge derivation relation $\vdash$. Eleven rules are defined for this new datatype:

member: $M \in B \Rightarrow B \vdash M$
pairing: $B \vdash M \wedge B \vdash M' \Rightarrow B \vdash (M, M')$
splitting: $B \vdash (M, M') \Rightarrow B \vdash M \wedge B \vdash M'$
key deriv.: $B \vdash K \Rightarrow B \vdash K^{\sim} \wedge B \vdash K^{+} \wedge B \vdash K^{-}$
hashing: $B \vdash M \Rightarrow B \vdash H(M)$
shared key encr.: $B \vdash M \wedge B \vdash K^{\sim} \Rightarrow B \vdash \{M\}_{K^{\sim}}$
public key encr.: $B \vdash M \wedge B \vdash K^{+} \Rightarrow B \vdash \{M\}_{K^{+}}$
private key encr.: $B \vdash M \wedge B \vdash K^{-} \Rightarrow B \vdash \{M\}_{K^{-}}$
shared key decr.: $B \vdash \{M\}_{K^{\sim}} \wedge B \vdash K^{\sim} \Rightarrow B \vdash M$
public key decr.: $B \vdash \{M\}_{K^{+}} \wedge B \vdash K^{-} \Rightarrow B \vdash M$
private key decr.: $B \vdash \{M\}_{K^{-}} \wedge B \vdash K^{+} \Rightarrow B \vdash M$

Honest agents and the intruder remain unchanged from [7]. For completeness, they are briefly recalled here.

A honest agent can take part in a protocol by using the events:

$send.A.B.M$ if agent A sends message M, with intended recipient B;

$receive.A.B.M$ if agent B receives message M, apparently from agent A;

$claimSecret.A.B.M$ if A thinks that M is a secret shared only with B; if B is not the intruder, then the intruder should not learn M;

$running.A.B.M_s$ if A thinks it is running the protocol with B; M_s is a message sequence, recording some details about the run in question.

$finished.A.B.M_s$ if A thinks it has finished a run of the protocol with B; M_s is a message sequence, recording some details about the run in question.

The $send$ and $receive$ events can also be treated as channels, used by agents to exchange data; the remaining events are used to formally define the desired security properties of the protocol. $Honest$ is the set of all honest agents.

The intruder acts as the medium, thus being allowed to see, modify, forge or drop any message. It uses its knowledge derivation relation $\vdash$ to forge new messages from the previously learnt messages. The set of messages it can derive from an initial knowledge S is defined as

$$deds(S) \triangleq \{M \in Message \,|S \vdash M\}.$$

Finally, the formal definition of the intruder is

$$
\begin{aligned}
&INTRUDER(S) \triangleq \\
&\quad \Box_{M \in Message} send?A?B!M \rightarrow INTRUDER(S \cup \{M\}) \\
&\quad \Box \Box_{M \in deds(S)} receive?A?B!M \rightarrow INTRUDER(S) \\
&\quad \Box \Box_{M \in deds(S)} leak.M \rightarrow INTRUDER(S)
\end{aligned}
$$

where $send$ and $receive$ are the communication channels, and $leak.M$ is the event that signals that the intruder can derive M from its current knowledge. The set of all agents is defined as $Agent = Honest \cup \{INTRUDER\}$.

Like in [7], attacks are specified as trace properties. A trace specification $SPEC(tr)$ is a predicate whose free variable tr represents a trace. A process satisfies a specification if the $SPEC(tr)$ predicate is true for all the traces of the process:

$$P \ \mathbf{sat} \ SPEC \Leftrightarrow \forall \, tr \in traces(P) \cdot SPEC(tr).$$

Two predicates, namely secrecy and authentication, are defined.

Secrecy states that if agent A believes that message M is shared only with honest agent B, then the intruder must not be able to derive M from its knowledge.

$$
\begin{aligned}
&Secrecy(tr) \triangleq \forall \, A \in Agent; B \in Honest \cdot \\
&\quad claimSecret.A.B.M \text{ in } tr \Rightarrow \neg leak.M \text{ in } tr
\end{aligned}
$$

Authentication states that, for each protocol run that A thinks it has finished with B, B must have started a protocol run with A, and both A and B must agree on some set M_s of data, belonging to $AgreementSet$.

$$
\begin{aligned}
&Agreement_{AgreementSet}(tr) \triangleq \\
&\quad \forall \, A \in Agent; B \in Honest; M_s \in AgreementSet \cdot \\
&\qquad tr \downarrow finished.A.B.M_s \leq tr \downarrow running.B.A.M_s
\end{aligned}
$$

where $tr \downarrow ev$ is the number of events ev appearing in the trace tr.

3. Modelling and Simplifying the Channel Encoding/Decoding Layer

In interoperable protocol implementations, all actors must exchange data encoded by the specified *external* representation, however, they can store data encoded in any *internal* representation, provided there exist some functions that can translate to and from the two. In this paper, such translation functions are called the "encoding/decoding layer". It is assumed, and thus modeled accordingly, that, as usual, such encoding/decoding layers are implemented separately from the protocol logics.

Following the CSP modelling approach presented in [7], and recalled in section 2, an actor performs all of its inputs on the *receive* channel, and all of its outputs on the *send* channel. Then, for actors A and B, the abstract formal model of a protocol can be represented as in figure 1.

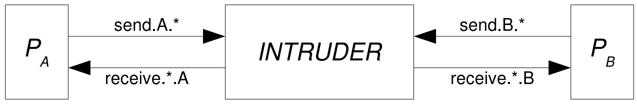

Figure 1. Actors A and B with $INTRUDER$ in $SYSTEM$.

The model representing all the honest agents and the intruder is called *SYSTEM*, and is formally defined as

$$SYSTEM \triangleq INTRUDER(IK_0) \parallel (|||_{A \in Honest} P_A)$$

where, for each $A \in Honest$, P_A is the CSP process that describes A's behavior, and IK_0 is the initial intruder knowledge. That is, the intruder and the protocol agents are synchronized on their common events.

It must be noticed that in $SYSTEM$ the actors directly exchange the abstract representation of data with the intruder.

In order to model the encoding/decoding layer, a refined model $SYSTEM'$ is defined as depicted in figure 2 for actors A and B.

Basically, $SYSTEM'$ acts like $SYSTEM$, but it is explicitly modeled that the *external* representation of data is being sent over *send* and *receive*. More precisely, for each honest agent A, the coupled processes P'_A and E_A represent respectively the protocol logic and the encoding/decoding layer of a program. So each P'_A in $SYSTEM'$ acts *like* its corresponding P_A in $SYSTEM$, but it is explicitly modeled that it sends its *internal* representation to its coupled encoding layer E_A, which in turn sends the encoded data to the intruder, and vice versa.

This model can be described in CSP for all the honest agents as

$$
\begin{aligned}
&SYSTEM' \triangleq INTRUDER(IK'_0) \parallel \\
&\quad (((|||_{A \in Honest} P'_A) \parallel (|||_{A \in Honest} E_A)) \setminus \{int\})
\end{aligned}
$$

where $int = int\_send, int\_receive$.

It could be argued that, potentially, this model allows each honest agent to send messages to any encoding layer, and vice versa. However, implementations of protocol logic and its coupled encoding layer are very often part of the same application, so errors that would lead honest agents or encoding layers to communicate with the wrong process are not realistic. For this reason, a correct model P'_A for the protocol logic must be defined such that it will only exchange messages with its coupled encoding layer model E_A, and vice versa. Indeed, the assumption that the definitions of agent and encoding layer models are correct, implies that such errors cannot happen in the model too. For the same

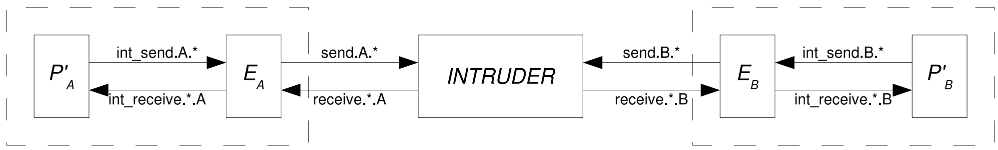

Figure 2. Actors A and B with $INTRUDER$ in $SYSTEM'$.

reason, it is reasonable to hide the program internal communication channels int_send and $int\_receive$ from the intruder's view.

Finally, the relation between each P_A and the corresponding P'_A and the formal definition of each E_A are given. For each P_A, P'_A can be built by refining P_A so as to model the information that the protocol agent must provide to the encoding/decoding layer for its proper working. More precisely, P'_A is obtained from P_A by replacing each $send.A.B.M$ event in P_A with $int\_send.A.B.(\text{ATOM } \mathcal{L}, (a, M))$, and each $receive.B.A.M$ event with $int\_receive.B.A.(\text{ATOM } \mathcal{L}, (a, M))$. Here, ATOM $\mathcal{L}$ is a special atom not present in the definition of P_A, whose only purpose is to tag the data exchanged on the internal channels, and a is such that $a \in Encoding \subseteq Message$ where $Encoding$ is the set of messages that can be used as encoding/decoding parameters.

It is worth noting that an accurate model must set, for each message M that is sent or received, its correct encoding parameters a according to the protocol specification documents. It can also be noted that the encoding parameters may or may not be already present in P_A. For example, if the encoding parameters a are being negotiated within the protocol logic, then a will be already present in P_A. Because of this, it is possible that the message (a, M) already exists in P_A. This is why we want to distinguish the messages (a, M) already present in P_A, from those added when deriving P'_A, which is achieved by the special label message ATOM $\mathcal{L}$, which has the property of never appearing in P_A. Since ATOM $\mathcal{L}$ is just a syntactic marker, it is assumed that neither P_A nor P'_A ever accept ATOM $\mathcal{L}$ on inputs or send it on outputs, with the only exception when ATOM $\mathcal{L}$ is explicitly needed as syntactic marker.

Each process E_A models the behavior of the encoding/decoding layer. Because of this, it is able to perform two kinds of actions: receive from its coupled process P'_A internal representations of data, along with encoding parameters, and send encoded data to the $INTRUDER$ process; receive encoded data from the $INTRUDER$ process, and send to its coupled process P'_A the internal representation, obtained using the decoding parameters specified by P'_A.

Apart from these assumptions on the possible interactions of E_A, it is assumed that internally E_A can behave in any way. The only restriction is that E_A can access only the data explicitly provided from outside. This behavior can be represented by the CSP process in figure 3, where $e_A(a, M)$ and $d_A(a, y)$ represent the result of the encoding and decoding operations and are messages such that

$$e_A(a, M) \in deds(\{a, M\}) \;\wedge\; d_A(a, y) \in deds(\{a, y\}) \tag{1}$$

By this definition, it is possible to state some properties of the modeled encoding/decoding layer E_A. The result $e_A(a, M)$ of encoding M with parameters a can be anything that can be derived from M and a. Two aspects of this definition are particularly interesting: $e_A(a, M)$ can contain the same or less information than M; all information in $e_A(a, M)$ that is not present in M must be present in a. That is, a possibly incorrect encoding function can loose some information on M, but can only use information that comes from the internal representation and from the encoding parameters. In order to model some information that is hard coded into the encoding function implementation, it is needed to explicitly add that information to a. The same reasoning applies to the result $d_A(a, y)$ of decoding y with parameters a.

Another property implied by this model is that one computation of $e_A(a, M)$ and of $d_A(a, y)$ has no side effects and is memoryless. Encoding mechanisms with memory are not considered here.

It is worth noting that all the properties of the modeled encoding layer, namely that the only data accessed by the encoding/decoding functions, including hard-coded values, are their input parameters and that no side effect occurs, are information flow properties that can be verified on implementation code, by means of static sequential code analysis techniques.

3.1. Model Simplifications

Removing the Encoding/Decoding Layer. The removal of the encoding/decoding layer from $SYSTEM'$ leads to a new process $SYSTEM''$, defined as

$$SYSTEM'' \triangleq INTRUDER(IK''_0) \parallel (|||_{A \in Honest} P'_A)$$

where the initial intruder knowledge is now called IK''_0 and is assumed to be defined as

$$IK''_0 \triangleq IK'_0 \cup Encoding \cup \{\text{ATOM } \mathcal{L}\} \tag{2}$$

$$E_A(i\_s, i\_r, s, r) \triangleq \Box \begin{array}{l} \Box_{a \in Encoding, M \in Message}\, i\_s!A?B!(\text{ATOM}\ \mathcal{L}, (a, M)) \rightarrow s!A!B!e_A(a, M) \rightarrow E_A(i\_s, i\_r, s, r) \\ \Box_{y \in Message}\, r?B!A!y \rightarrow \Box_{a \in Encoding}\, i\_r!B!A!(\text{ATOM}\ \mathcal{L}, (a, d_A(a, y))) \rightarrow E_A(i\_s, i\_r, s, r) \end{array}$$

Figure 3. Formal definition of E_A process.

The fault preservation of this transformation is expressed by:

Theorem 1 *If the definition of attack does not involve the send and receive events, then if an attack exists in a trace tr' of $SYSTEM'$, then there exists a trace tr'' of $SYSTEM''$, such that an attack exists in tr''. Formally, let $comm = \{send, receive\}$, then*

$$P \textbf{ sat } SPEC \Leftrightarrow P \setminus comm \textbf{ sat } SPEC \quad (3)$$
$$\Longrightarrow$$
$$SYSTEM'' \textbf{ sat } SPEC \Rightarrow SYSTEM' \textbf{ sat } SPEC$$

A formal proof of this theorem is not included here for lack of space. It can be found in [10].

Theorem 1 is a general result, valid for any security requirement $SPEC$, not only secrecy or agreement, provided that $SPEC$ does not involve the $send$ and $receive$ channels, as formally stated by expression (3). This is a reasonable constraint, because usually security properties are obtained by correct use of special events, such as $claimSecret$, $running$ or $finished$, and not directly by observing the sequence of messages exchanged on the communication channels.

This theorem states that in a protocol specification where the encoding/decoding layer is modeled as previously described, only the protocol logic represented by P'_A is responsible for the security properties of the whole protocol, while any possible implementation of the encoding/decoding layer E_A can be considered as part of the intruder, provided that the latter knows all required encoding schemes (because $Encoding \subset IK''_0$). It is also needed that the intruder knows the syntactic marker ATOM $\mathcal{L}$. This is not an issue, since it is assumed that ATOM $\mathcal{L}$ will only be treated as a marker by honest agents.

Note that no assumption on the invertibility of encoding functions has been made, thus even erroneous specifications of encoding schemes are safe (thought not functional, of course). Moreover, since no assumption on implementation correctness has been made, even erroneous implementations of the encoding scheme are safe, provided they satisfy the data flow assumptions made.

Removing the Encoding Parameters. Since each P'_A is being built from P_A (plus other information), it is possible to find a simplifying transformation that can take from P'_A back to P_A.

Simplifying transformations have been introduced in [7]. The work proposed here extends that approach, and applies it in order to obtain new results.

A fault-preserving simplifying transformation in its simplest form is a function $f : Message \rightarrow Message$ that defines how messages in the original protocol are replaced by messages in the simplified protocol. The function f is then overloaded to take events, traces and processes, such that all messages in the events, traces or processes are replaced.

As stated in [7], if $SYSTEM^R$ is a process and $SYSTEM^A = f(SYSTEM^R)$ where $f(\cdot)$ is a simplifying transformation that satisfies conditions

$$\begin{aligned} &\forall\, B \in \mathbf{P}(Message); M \in Message \cdot \\ &B \cup IK_0^R \vdash M \Rightarrow f(B) \cup IK_0^A \vdash f(M) \end{aligned} \quad (4)$$
$$f(IK_0^R) \subseteq IK_0^A \quad (5)$$

then $SYSTEM^A$ **sat** $Secrecy \Rightarrow SYSTEM^R$ **sat** $Secrecy$.

In this work, a sufficient condition for fault preserving transformations with respect to authentication agreement specifications, weaker than the one given in [7], is used:

$$\begin{aligned} &\forall\, M_s, M'_s \in AgreementSet \cdot \\ &M_s \neq M'_s \Rightarrow f(M_s) \neq f(M'_s) \end{aligned} \quad (6)$$

That is, $f(\cdot)$ must be *locally* injective on $AgreementSet$, and not on the whole $Message$ set, like in [7]. The proofs of this and other extensions can be found in [10].

A fault-preserving renaming transformation that transforms P'_A into P_A, and thus $SYSTEM''$ into $SYSTEM$, is now introduced. Like in [7], this transformation collapses each pair (M, M') belonging to the $Pairs$ set into its first item M.

The definition of $f(\cdot)$ is

$$\begin{aligned} f(\text{ATOM}\ A) &= \text{ATOM}\ A, \\ f(M, M') &= \begin{cases} f(M), & \text{if } (M, M') \in Pairs \\ & \quad \wedge \neg isPair(M'), \\ (f(M), f(M')), & \text{if } (M, M') \notin Pairs \\ & \quad \wedge \neg isPair(M'), \end{cases} \\ f(M, (M', M'')) &= \begin{cases} f(M, M''), & \text{if } (M, M') \in Pairs, \\ (f(M), f(M', M'')), & \text{otherwise,} \end{cases} \\ f(\{M\}_{K^*}) &= \{f(M)\}_{f(K^*)} \\ f(H(M)) &= H(f(M)) \\ f(K^*) &= f(K)^* \end{aligned}$$

where K^* ranges over $\{K^{\sim}, K^+, K^-\}$.

In order to preserve secrecy, $f(\cdot)$ must satisfy conditions (4) and (5). It can be shown that if we set

$$IK_0^A = f(IK_0^R) \cup \{f(M') | (M, M') \in Pairs\} \quad (7)$$

then condition (4) holds, and condition (5) is clearly satisfied too. Equation (7) states that the intruder must already know the information that is going to be coalesced away.

In order to preserve agreement, $f(\cdot)$ must satisfy condition (6). In order to achieve this, only one additional constraint is required:

$$\forall\, M_s \in AgreementSet;\, subM \in subterms(M_s) \cdot \\ isPair(subM) \Rightarrow subM \notin Pairs \quad (8)$$

where $subterms(M)$ is the set containing M and all its subterms. Constraint (8) means that no subterm of any $M_s \in AgreementSet$ that is a pair must be in the $Pairs$ set, that is if agreement is required on a pair, then that pair must not be coalesced away.

Now it is possible to show how the coalescing pairs function $f(\cdot)$ can be safely used to transform P'_A into P_A, and thus $SYSTEM''$ into $SYSTEM$. It is worth reminding that P'_A has been obtained from P_A by replacing each sent or received message M with $(\text{ATOM } \mathcal{L}, (a, M))$. Then, the following two steps are required in order to obtain back P_A from P'_A:

1. $P_A^{tmp} = f(P'_A)$, with $Pairs = \{(\text{ATOM } \mathcal{L}, a) | a \in Encoding\}$
2. $P_A = f^{sym}(P_A^{tmp})$, with $Pairs = \{(\text{ATOM } \mathcal{L}, M) | M \in Message\}$

where $f^{sym}(\cdot)$ is the symmetric function of $f(\cdot)$, that coalesces pairs of the form (M, M') into their second item M'.

In step 1, the syntactic marker ATOM $\mathcal{L}$ is used to find and remove all encoding parameters that have been added by the representation of the encoding/decoding layer. Then, step 2 removes the syntactic marker, finally obtaining P_A.

Each one of these transformations preserves secrecy and authentication if the required sufficient conditions (7) and (8) hold.

In step 1, by setting $IK_0^{tmp} = f(IK''_0) \cup f(Encoding) = f(IK'_0) \cup f(Encoding) \cup \{\text{ATOM } \mathcal{L}\}$, that is, by requiring that the intruder already knows all encoding schemes, condition (7) is clearly satisfied. As stated above, ATOM $\mathcal{L}$ in the intruder knowledge is not an issue.

Moreover, condition (8) holds because $Pairs \cap subterms(AgreementSet) = \emptyset$. Indeed, in step 1 each element in $Pairs$ has the form $(\text{ATOM } \mathcal{L}, a)$; but ATOM $\mathcal{L}$ can never appear in any $running$ or $finished$ event, and thus in any subterm of the $AgreementSet$, because it is assumed that no honest agent will ever input or internally generate the ATOM $\mathcal{L}$ value, except when the syntactic marker is explicitly needed.

$$\begin{array}{l} SSHClient(IDC, me, you, CAlgs, TrustedKeys) = \\ send!me!you!IDC \rightarrow receive!you!me?IDS \rightarrow \\ \sqcap_{cookieC \in Cookies}\, send!me!you!(cookieC, CAlgs) \rightarrow \\ receive!you!me?(cookieS, SAlgs) \rightarrow \\ g := Negotiate(CAlgs, SAlgs, \text{`}g\text{'}) \in Parameter \\ p := Negotiate(CAlgs, SAlgs, \text{`}p\text{'}) \in Parameter \\ \sqcap_{x \in DHSecrets}\, send!me!you!EXP(g, x, p) \rightarrow \\ receive!you!me?(KeyS^+, DHPublicS, \{sshash\}_{KeyS^-}) \rightarrow \\ GO(EXP(DHPublicS, x, p), sshash, KeyS^+, TrustedKeys) \end{array}$$

Figure 4. A possible abstract model of an SSH-TLP client.

In step 2, condition (7) is clearly satisfied if we set $IK_0 = IK_0^{tmp}$; condition (8) holds because of the same reasoning used for step 1.

4. Modelling an SSH Transport Layer Protocol Client

In this example, another syntactic sugar is added: lists of n messages are reduced to nested pairs. So, for example, (M, M', M'') is equal to $(M, (M', M''))$.

The SSH Transport Layer Protocol [15] (SSH-TLP) is part of the SSH three protocols suite [14]; in particular it is the first protocol that is used in order to establish an SSH connection between client and server. SSH-TLP gives server authentication to the client, and establishes a set of session shared secrets.

A possible abstract model of an SSH-TLP client is reported in figure 4.

The $SSHClient$ process begins a protocol session with the server by sending it the client identification string denoted IDC. The server responds with IDS, the server identification string. Then the client sends a nonce $cookieC$, followed by the client lists of supported algorithms $CAlgs$. The server responds sending a nonce $cookieS$, followed by the server lists of supported algorithms $SAlgs$. The client then computes the value of the Diffie Hellman (DH) parameters g and p by the $Negotiate(CAlgs, SAlgs, Param)$ function, which returns the requested negotiated algorithm parameter named $Param$, obtained from the supported client and server algorithms $CAlgs$ and $SAlgs$. Here, $Parameter \subseteq Message$ is the set of all messages that can be used as DH parameters. Once g and p have been obtained, the client sends its DH public key $EXP(g, x, p)$, which is a message representing $g^x \bmod p$. This message is added to the datatype and is defined as

$$\text{EXP } Message, Message, Message$$

along with the syntactic sugar $\text{EXP } g, x, p = EXP(g, x, p)$.

From the point of view of the intruder knowledge derivation relation $\vdash$, the $EXP(\cdot)$ message is like an hash. Only a single 'exponentiation' rule, defined as

$$B \vdash g \wedge B \vdash x \wedge B \vdash p \Rightarrow B \vdash EXP(g, x, p)$$

is needed. This rule is similar to the 'hashing' one. For this reason, all previously proven results still hold.

The $EXP(\cdot)$ function has the following additional property

$$EXP(EXP(g, y, p), x, p) = EXP(EXP(g, x, p), y, p) \tag{9}$$

so that two messages satisfying equation (9) are considered equal by all parties, both protocol actors and the intruder.

When the EXP function is used for the DH key exchange algorithm, x and y can be considered as DH private keys, $EXP(g, x, p)$ and $EXP(g, y, p)$ as DH public keys, and the expression in (9) as the DH shared key that can be obtained by each actor. Finally, g and p are the DH group parameters, that must be explicitly represented in the datatype, in order to express the exponentiation property. In the next step of the protocol, the client receives a message containing the server public key $KeyS^+$, the server DH public key $DHPublicS$ and the server signed final hash $\{sshash\}_{KeyS^-}$. The server will compute its DH public key as $DHPublicS = EXP(g, y, p)$, where y is the server's DH private key. However the client is modeled to receive an opaque $DHPublicS$ message, because the server DH public key is an opaque value from the client's point of view. The server final hash $sshash$ is the value upon which agreement is required, and contains all the relevant data of a protocol session, i.e.

$$\begin{aligned} &sshash = H(H(IDC, IDS, (cookieC, CAlgs),\\ &\quad (cookieS, SAlgs), KeyS^+, EXP(g, x, p),\\ &\quad DHPublicS, EXP(DHPublicS, x, p))) \end{aligned}$$

It is worth to notice the double hashing $H(H(\cdot))$. The internal hash is the final hash, computed on all relevant session data, prescribed by the protocol specification, while the external hash is the one required by the signature algorithm, which prescribes to hash, and then cipher, the value that must be signed. The $EXP(DHPublicS, x, p)$, that is used inside the final hash, is the DH shared key as computed by the client, that is the session secret shared between the client and the server. Finally, the $GO(\cdot)$ process is defined as

$$\begin{aligned} &GO(DHKey, sshash, KeyS^+, TrustedKeys) =\\ &\quad (claimSecret.me.you.DHKey \rightarrow\\ &\qquad finished.me.you.sshash)\\ &\not< KeyS^+ \in TrustedKeys \not> STOP \end{aligned}$$

where $P \not< b \not> Q$ means *if b then P else Q*. That is, if the server public key $KeyS^+$ is in the $TrustedKeys$ set, the set of server public keys trusted by the client, then the protocol run ends well, and all security properties can be claimed, namely the secrecy of the DH shared key $DHKey$, and the agreement on the server signed final hash, $sshash$.

$$\begin{aligned} &SSHClientRef(IDC, me, you, CAlgs, TrustedKeys) =\\ &send!me!you!(\textsc{Atom}\ \mathcal{L}, (string, IDC)) \rightarrow\\ &receive!you!me?(\textsc{Atom}\ \mathcal{L}, (string, IDS)) \rightarrow\\ &\sqcap_{cookieC \in Cookies}\ send!me!you!(\textsc{Atom}\ \mathcal{L},\\ &\quad (KEX, (cookieC, CAlgs))) \rightarrow\\ &receive!you!me?(\textsc{Atom}\ \mathcal{L}, (KEX, (cookieS, SAlgs))) \rightarrow\\ &g := Negotiate(CAlgs, SAlgs, `g\text{'}) \in Parameters\\ &p := Negotiate(CAlgs, SAlgs, `p\text{'}) \in Parameters\\ &\sqcap_{x \in DHSecrets}\ send!me!you!(\textsc{Atom}\ \mathcal{L},\\ &\quad ((bin\_pack, mpint), EXP(g, x, p))) \rightarrow\\ &receive!you!me?(\textsc{Atom}\ \mathcal{L}, ((bin\_pack, string, mpint, string),\\ &\quad (KeyS^+, DHPublicS, \{sshash\}_{KeyS^-}))) \rightarrow\\ &GO(EXP(DHPublicS, x, p), sshash, KeyS^+, TrustedKeys) \end{aligned}$$

Figure 5. A possible refined model of an SSH-TLP client.

Now that the abstract model of the client has been introduced, figure 5 shows the refined client model, which takes interaction with the encoding/decoding layer into account.

In this refined model,

$$\begin{aligned} KEX &= (bin\_pack, bytes, namelists)\\ Encoding &= \{string, bin\_pack, bytes, namelists, mpint\} \end{aligned}$$

and each $send$ or $receive$ event has its own associated marshalling parameters, as specified by the SSH-TLP description [14, 15].

The simplifying transformation as described in section 3.1 maps the $SSHClientRef$ process back onto the $SSHClient$ process.

If secrecy or authentication are going to be verified, then it is possible to verify the abstract model, provided that the intruder knows all the encoding schemes. If instead other properties are going to be verified, the refined client model should be used. Note that, even if this model is a bit more complex than the abstract one, it is still much simpler than a fully refined model including marshalling and unmarshalling operations.

5. Conclusions

The work presented in this paper is a useful step towards the verification of refined security protocol models that take encoding and decoding data transformations into account, thus allowing formal verification to get closer to protocol code written in a programming language.

The main contribution of the paper is the formulation of a set of sufficient conditions under which the models of marshalling and unmarshalling operations applied when sending and receiving messages on channels can be safely simplified or even completely abstracted away, and its formal justification.

A refined protocol model corresponding to a typical layered implementation of such operations has been defined. It has been shown that, in order to verify secrecy or authentication properties on the refined model, it is enough to verify those properties on the corresponding abstract model, provided that the intruder knows the encoding parameters, which is a reasonable assumption. Alternatively, in order to check a generic security property defined on protocol traces, still a simplified refined model can be used, in place of the full one.

The model of encoding schemes that has been developed in this work is general enough to take into account a widely used class of encoding schemes, namely the memoryless and side effect free ones. Moreover, no assumption has been made on encoding scheme invertibility, nor on implementation correctness; it is required instead that the implementation of encoding and decoding functions satisfies some data flow properties, that can be checked by standard static analysis techniques. The exploitation of this result is that, if some data flow properties are satisfied on implementation code, then even erroneous specifications or implementations of encoding schemes cannot be more harmful than an intruder is.

Although some of these results are probably not so surprising, all of them have been formally stated for the first time in this paper, and they find application in improving the development of formally verified implementation code of security protocols, both using the code generation approach or the model extraction approach.

With code generation, the developer only needs to write the abstract protocol model, and the code generation engine can take care of ensuring that the generated code meets the needed requirements.

When adopting a model extraction approach, it is possible to avoid extracting from code a complex model that represents all the implementation details of encoding and decoding functions, but a simpler model can be extracted, provided that some static checks are first made on the implementation code.

We can expect that, by abstracting away implementation details from the model, a reduction in the time needed for formal verification and the possibility to analyze more complex protocols are finally achieved. It may be argued that, in order to abstract details away, some properties have to be verified on the implementation code. This is true, however, the implementation code to be checked is sequential, and thus easier and faster to be checked in isolation than it is checking a protocol model that includes a detailed model of the encoding and decoding functions.

Some issues on the topics presented in this paper are still open for future work. In particular, it can be interesting to explore under which conditions the final transformation back to the original abstract model is safe for other security trace properties. Another interesting further work is to consider verification with computational models instead of verification with Dolev-Yao models.

References

[1] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Theoretical Aspects of Computer Software*, pages 82–94, 2001.

[2] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[3] K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-security protocols. In *Web Services and Formal Methods*, pages 88–106, 2006.

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

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

[6] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation*, pages 363–379, 2005.

[7] M. L. Hui and G. Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

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

[9] J. K. Millen and V. Shmatikov. Symbolic protocol analysis with an abelian group operator or diffie-hellman exponentiation. *Journal of Computer Security*, 13(3):515–564, 2005.

[10] A. Pironti and R. Sisto. Reasoning about some security protocol implementation details, revision 4. Technical Report, Politecnico di Torino, Oct. 2007. Available at: http://staff.polito.it/riccardo.sisto/reports/encoding_4.ps.

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

[12] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

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

[14] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[15] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), Jan. 2006.