Towards ensuring the correctness of security protocol implementations

Alfredo Pironti[1] and Riccardo Sisto[1]

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
phone: +390115647073 fax: +390115647099

Abstract. In formal methods, security protocols are usually modeled with a high level of abstraction. In particular, message marshalling and unmarshalling operations and cryptographic parameters are often abstracted away. However, in real applications, errors in these protocol components could be exploited to break protocol security.
In order to solve this issue, this paper formally proves that, under some easy to check constraints, if an abstract protocol model is secure, then a refined model, which takes into account all the possible implementations of the marshalling/unmarshalling operations and explicitly models the cryptographic parameters, 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 analyse 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 details such as binary representation of messages, message marshalling/unmarshalling transformations and cryptographic algorithms and parameters, which are all abstracted away in the formal abstract model. Moreover, errors are always possible in the implementation of such operations in a programming language, and techniques such as testing or code reviews can only limit the occurrence of errors rather than rule it out.

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 abstract formal models used for data types, so as to represent the additional properties of certain cryptographic operations, such as for example those based on the use of exponentiation or exclusive-or functions (e.g. [1]). Other researchers aim at reconciling the abstract formal models of cryptography with computational models, which are closer to implementations (e.g. [2]). Another research line, instead, focuses on the concrete implementation of the protocol, rather than on the concrete implementation of cryptography. The addressed problem is 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 models. In particular, two different strategies have been explored, namely automatic code generation from abstract models ([3, 4]) and automatic model extraction from implementation code ([5, 6]).

Methods based on automatic code generation start from a high-level, formally verified, specification of the protocol, which abstracts away from details about how cryptographic and communication operations actually take place, and fill the semantic gap between formal specification and implementation, guided by implementation choices provided by the user. The methods published so far are not yet sufficiently refined, because, for example, they do not allow the use of arbitrary marshalling/unmarshalling methods, thus preventing in practice the development of interoperable implementations of standard protocols. Another current limitation of such methods is the lack of a formal proof that guarantees that the generated implementation code correctly refines the abstract model.

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 [5, 6].

The method described in [5], even with some limitations, has been shown to work on real, interoperable protocol implementations, and the correctness of its model extraction scheme has been formally established.

One of the things that can be observed looking at the results reported in [5] and [6], is that the part of the extracted formal model that describes implementation details such as message encoding and decoding operations can be quite complex, much more complex than the abstract protocol model itself. This occurs even though in [5] the implementations of some low-level library operations, such as those for basic XML manipulation, are not included in the model but rather assumed to correctly refine their symbolic counterpart.

The idea that is developed in this paper is that the complex models of some implementation details can be avoided and substituted by simpler models or assumptions, which can be checked more easily, while obtaining the same kind of security assurance on the protocol implementation. The focus of this paper is on the message encoding and decoding operations, generally referred to as marshalling and unmarshalling, and on the parameters that specify how cryptographic operations must be implemented (for example, the specification of the cryptographic algorithm or the initialization vector to be used for encryption and decryption).

The first step is the definition of some formal general models of the implementation details that are being considered. Such models are general, in the sense that they do not describe specific implementations but rather they capture 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, provided that the general models are simple enough and the assumptions made are easy to be checked on real protocol implementations.

A second step that can be made in order to favor this advantage is to further simplify the models, using fault-preserving transformations like for example the ones introduced in [7]. The interesting result that is finally obtained in this paper is that, under some simple assumptions, the protocol models including implementation details can be transformed into the original abstract protocol models without implementation details, and the classical security faults (strong secrecy and strong forward authenticity) are preserved in this transformation. This means that, provided the assumptions we made are verified on a given implementation, the formal model of the implementation details can be left out in verifying certain security properties.

It is worth noting that this kind of result can be exploited both when using the model extraction approach and when using code generation. In the latter case, the starting point is an already verified abstract protocol model. Consequently, it is enough to ensure that the generated code satisfies the needed assumptions.

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. This is essentially a generalization of the approach adopted in [7], which mainly adds the possibility of non-atomic keys. Section 3 formalizes how message encoding and decoding operations and parameters of cryptographic operations can be modelled, and discusses about the assumptions that such models imply on real implementations. Section 4 shows how the models can be simplified while preserving security properties. While showing these results, a generalization of the fault-preserving transformations introduced in [7] is also introduced. Section 5 shows an example of how the modelling approach presented here can be applied to a real security protocol, i.e. the SSH Transport Layer Protocol, and how its model can be simplified. Finally, section 6 concludes with an overview of the achievements that have been reached and their practical impact.

2 Abstract protocol models and notation

The formalism used in this paper is based on CSP [8,9], and the datatype definitions and protocol models are an extension of the ones used in [7]. Essentially, they follow the Dolev-Yao approach. The main extension w.r.t. [7] is an added support for non atomic keys. The new datatype is defined as

$$
\begin{aligned}
\mathit{Message} ::= {} & \textsc{Atom}\ \mathit{Atom} \mid \\
& \textsc{Pair}\ \mathit{Message}\ \mathit{Message} \mid \\
& \textsc{ShKey}\ \mathrm{Message} \mid \\
& \textsc{PubKey}\ \mathit{Message} \mid \\
& \textsc{PriKey}\ \mathit{Message} \mid \\
& \textsc{ShKeyEncrypt}\ \mathit{Message}\ \textsc{ShKey}\ \mathit{Message} \mid \\
& \textsc{PubKeyEncrypt}\ \mathit{Message}\ \textsc{PubKey}\ \mathit{Message} \mid \\
& \textsc{PriKeyEncrypt}\ \mathit{Message}\ \textsc{PriKey}\ \mathit{Message} \mid \\
& \textsc{Hash}\ \mathit{Message}.
\end{aligned}
$$

This definition has been developed using the following guidelines:

- Each key is typed. It is possible to obtain a key from generic material (that is, any generic *Message*). However, it is not possible to use raw material directly as a key; instead, the material must first be fed to a key construction operator.
- 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.
- No new types are added in order to represent cryptographic 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	Representation
PAIR M M'	(M, M')
SHKEY M	$M^{\sim}$
PUBKEY M	M^{+}
PRIKEY M	M^{-}
SHKEYENCRYPT M SHKEY K	$\{M\}_{K^{\sim}}$
PUBKEYENCRYPT M PUBKEY K	$\{[M]\}_{K^{+}}$
PRIKEYENCRYPT M PRIKEY K	$[\{M\}]_{K^{-}}$
HASH M	$H(M)$

Once the datatype is modified, 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 $B \vdash M \wedge B \vdash M' \wedge \neg pair(M) \Rightarrow B \vdash (M, M')$
splitting $B \vdash (M, M') \Rightarrow B \vdash M \wedge B \vdash M'$
key derivation $B \vdash K \Rightarrow B \vdash K^{\sim} \wedge B \vdash K^{+} \wedge B \vdash K^{-}$
shared key encryption $B \vdash M \wedge B \vdash K^{\sim} \Rightarrow B \vdash \{M\}_{K^{\sim}}$
public key encryption $B \vdash M \wedge B \vdash K^{+} \Rightarrow B \vdash \{[M]\}_{K^{+}}$
private key encryption $B \vdash M \wedge B \vdash K^{-} \Rightarrow B \vdash [\{M\}]_{K^{-}}$
shared key decryption $B \vdash \{M\}_{K^{\sim}} \wedge B \vdash K^{\sim} \Rightarrow B \vdash M$
public key decryption $B \vdash \{[M]\}_{K^{+}} \wedge B \vdash K^{-} \Rightarrow B \vdash M$
private key decryption $B \vdash [\{M\}]_{K^{-}} \wedge B \vdash K^{+} \Rightarrow B \vdash M$

hashing $B \vdash M \Rightarrow B \vdash H(M)$

In [7], the following lemma about the relation $\vdash$ is proven to hold:

Lemma 1.

$$B \vdash M \wedge B \subseteq B' \Rightarrow B' \vdash M, \tag{1}$$

$$B \vdash M \wedge B \cup \{M\} \vdash M' \Rightarrow B \vdash M'. \tag{2}$$

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 following events:

$send.A.B.M$ agent A sends message M, with intended recipient B;
$receive.A.B.M$ agent B receives message M, apparently from agent A;
$claimSecret.A.B.M$ A thinks that M is a secret shared only with B; if B is not the intruder, then the intruder should not learn M;
$running.A.B.Ms$ A thinks it is running the protocol with B; Ms is a message sequence, recording some details about the run in question.
$finished.A.B.Ms$ A thinks it has finished a run of the protocol with B; Ms 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; while 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

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

Finally, the formal definition of the intruder is

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

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 $Agent$ set 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

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

Two predicates, namely strong secrecy and forward authentication, are defined.

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

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

In order to define forward authentication, a formal definition of $AgreementSet$ is given.

$$\begin{aligned} & \forall\, Ms \in Message^*; tr \in traces(P); A \in Agents; B \in Honest \cdot \\ & \quad Ms \in AgreementSet \Leftrightarrow \\ & \quad tr \downarrow finished.A.B.Ms > 0 \lor tr \downarrow running.B.A.Ms > 0 \end{aligned} \tag{3}$$

where $tr \downarrow e$ is the number of events e occurring in the trace tr. Informally, $AgreementSet$ is the set of all message sequences upon which the agents should agree.

Then, forward authentication is defined as

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

That is, 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 Ms of data.

3 Modelling implementation details

This section shows how some implementation details that are usually abstracted away in formal models, can be properly taken into account into CSP protocol specifications. The main goal is first to specify such details, and then to show that, under some conditions, even if these details are removed again, the verification of the obtained abstract model still takes into account all possible implementations of the abstracted details.

Two kinds of implementation details will be considered: message encoding/decoding, and cryptographic algorithms and parameters. A common strategy that is followed in both cases is to model such features without introducing new datatypes or operations.

3.1 Modelling message encoding/decoding functions

Discussion In the field of real applications, each protocol specification that aims to be interoperable, must explicitly specify the encodings applied to the data that are exchanged by protocol actors. That is, all the actors must exchange data with the same specified *external* representation. However, internally, each actor can use any representation that is suitable for its implementation, provided it can translate data from the *internal* representation to the *external* one, and vice versa. In this paper it is assumed that, as usual, such translations are implemented separately from the protocol logic. The functions that offer an interface in order to translate data representations are called here the "encoding/decoding layer".

However, in the field of formal methods, formally verified specifications of cryptographic protocols usually deal with a high level of abstraction, and the encoding/decoding layer is indeed abstracted away. On one hand, this abstraction is necessary to get clean, small formal specifications, that can be verified rapidly. One the other hand, however, it seems still possible that some implementation errors in the encoding/decoding layer, can affect the protocol security properties, thus decreasing the usefulness of formal verification. One possible solution to this issue may consist in explicitly representing the encoding/decoding layer within the formal model of a protocol that is going to be verified. However this approach has many disadvantages. It adds complexity to the model being verified, thus increasing the required verification time; moreover this approach is also error-prone, because by manually adding informations to the model, model errors can also be introduced. Finally, since protocol logic and implementation details are mixed together in the same specification, the interface offered by the encoding/decoding layer is lost.

In this section it is showed how the encoding/decoding layer of a generic protocol can be modeled externally from the protocol logic itself, so that the encoding/decoding layer interface is preserved. The main innovative result, showed in next section, is that, under some constraints that can be easily checked on real applications, any incorrect implementation of the encoding/decoding layer cannot be more harmful than an intruder is. An immediate useful result is that any formally verified model that abstracts the encoding/decoding layer away, actually is verified for all possible implementations of the encoding/decoding layer, provided a few, easy-to-check constraints hold in the implementation.

The following sub-section formalizes the model of the encoding/decoding layer.

Formal model 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. Moreover, the intruder acts as the medium, thus being allowed to see, modify, forge or drop any message. Then, for actors A and B, the abstract formal model of a protocol can be represented as in figure 1.

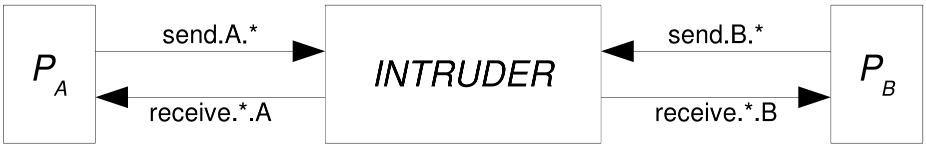

Fig. 1. Actors A and B with $INTRUDER$ in $SYSTEM$.

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

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

where, for each A in $Honest$, P_A is the CSP process that describes A's behaviour, and IIK is the initial intruder knowledge. 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.

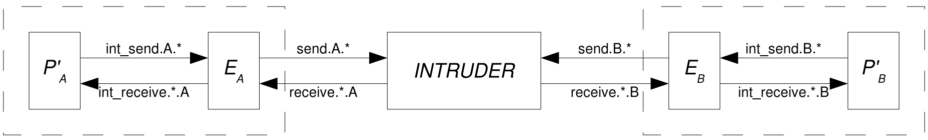

Fig. 2. Actors A and B with $INTRUDER$ in $SYSTEM'$.

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

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

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

It can be argued that, potentially, this model allows each honest agent to send messages to any encoding layer, and vice versa. However, honest agents will not try to fake their identities in order to communicate with other encoding layers. Moreover, the definition of E_A, given below, allows each encoding layer to communicate only with its coupled process P'_A and the intruder, but with no other process. Since each protocol logic P'_A and its coupled encoding layer E_A are considered part of the same application, implementation errors that would lead honest agents or encoding layers to send messages to the wrong process are not realistic. Furthermore, for the same reason, it is reasonable to hide the program internal communication channels *int_send* and *int_receive* from the intruder's view.

The initial intruder knowledge IIK'in $SYSTEM'$ has some relation with IIK in $SYSTEM$, however this relation now is irrelevant, and can be explained later.

Finally, the relation between each P_A and the corresponding P'_A and the formal definition of each E_A are given. The definitions of P_A and P'_A can be parameterized with respect to the channels used for communication. When not needed, process parameters will not be written. Thus, $P'_A(s,r)$ can be written as P'_A, when parameters are irrelevant in the context.

For each P_A, $P'_A(s,r)$ 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, $P'_A(s,r)$ is obtained from $P_A(s,r)$ by replacing each s event $s.A.B.M$ in $P_A(s,r)$ with $s.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))$, and each r event $r.B.A.M$ with $r.B.A.(\textsc{Atom}\ \mathcal{L},(a,M))$. Here, $\textsc{Atom}\ \mathcal{L}$ 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 a message suitable to represent the set of encoding/decoding parameters that must be provided to the encoding layer in order to enable the encode/decode operation, or, more formally

$$a \in Encodings \subseteq Message$$

where *Encodings* is the set of messages that can be used as encoding/decoding parameters. Normally, only atom and pair messages will be used as encoding parameters, but this restriction is not needed for our purposes.

It is worth to note that an accurate model must set, for each message M that is sent or received, its correct encoding parameters a. Which parameters must be used for each communication is a kind of information that is included in the protocol description. 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 $\textsc{Atom}\ \mathcal{L}$, which has the property of never appearing in P_A. Since $\textsc{Atom}\ \mathcal{L}$ is just a syntactic marker, it is assumed that neither P_A nor P'_A ever accept $\textsc{Atom}\ \mathcal{L}$ on inputs or send it on outputs, with the only exception when $\textsc{Atom}\ \mathcal{L}$ is explicitly needed as syntactic marker.

Each process E_A models the behaviour 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 this 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 behaviour can be represented by the following CSP process:

$$
\begin{aligned}
&E_A(i\_s, i\_r, s, r) \triangleq \\
&\quad \mathop{\Box}_{\substack{a \in Encodings \\ M \in Message}} i\_s!A?B!(\textsc{Atom}\ \mathcal{L}, (a, M)) \rightarrow s!A!B!e(a, M) \rightarrow E(i\_s, i\_r, s, r) \\
&\Box \\
&\quad \mathop{\Box}_{y \in Message} r?B!A!y \rightarrow \\
&\qquad\qquad \mathop{\Box}_{a \in Encodings} i\_r!B!A!(\textsc{Atom}\ \mathcal{L}, (a, d(a, y))) \rightarrow E(i\_s, i\_r, s, r)
\end{aligned}
\tag{4}
$$

where $e(a, M)$ and $d(a, y)$ represent the result of the encoding and decoding operations and are messages such that

$$
\begin{aligned}
e(a, M) &\in deds(\{a, M\}) \\
d(a, y) &\in deds(\{a, y\})
\end{aligned}
\tag{5}
$$

By this definition, it is possible to state some properties of the modeled encoding/decoding layer E_A.

The result $e(a, M)$ 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:

– $e(a, M)$ can contain the same or less information than M.
– All information in $e(a, M)$ 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 enough to add that information to a.

The same reasoning applies to the result $d(a, y)$ of decoding y with parameters a.

3.2 Modelling cryptographic algorithms and parameters

The abstract datatype defined in this work, intentionally represents the encryption function as a cyphertext obtained from a plaintext message and a key. The cyphertext has the property of being restored to plaintext only if the appropriate key is known. In the same way, the hash function is simply a non invertible representation of a message.

However, different real cryptographic algorithms obtain these properties in different, incompatible ways. For example, in a real application, if a message M is ciphered with the key K^+, under the RSA algorithm, obtaining $\{[M]\}_{K^+}$, then, even if the intruder knows K^- and $\{[M]\}_{K^+}$, it will not be able to recover M, unless it is using exactly the RSA algorithm. An interesting example is the SSH Transport Layer Protocol, that requires agreement on the value of an hashed message. If both actors obtain the same message M, and compute $H(M)$, it is a prerequisite to agreement that they have previously agreed on the same hashing algorithm, otherwise they are likely to obtain the same logical value (a non invertible representation of M), but different concrete values, so the agreement will fail.

In order to represent cryptographic parameters, we let

$$Parameters \subseteq Message$$

be the set of messages that can be used as such cryptographic parameters. If $a, b \in Parameters$, and P is a process, then any key $K^\sim$ that is sent or received in P, can be rewritten as

$$(\text{ATOM } \mathcal{L}, (b, K))^\sim$$

and any encryption $\{M\}_{K^\sim}$ that is sent or received in P, can be rewritten as

$$\{(\text{ATOM } \mathcal{L}, (a, M))\}_{(\text{ATOM } \mathcal{L},(b,K))^\sim}, \tag{6}$$

where ATOM $\mathcal{L}$ is the special label atom introduced before, useful to distinguish the messages (a, M) and (b, K) already present in P from those added by this transformation. The same reasoning applies to public and private key cryptography. In the same way, any hash $H(M)$ that is sent or received in P can be rewritten as

$$H(\text{ATOM } \mathcal{L}, (a, M))$$

It is worth to note that this model can fully handle the case where both the ciphertext and the key have their own, possibly different parameters. Furthermore, this model implies that it is impossible to achieve “security through obscurity”: in the encryption defined in (6), it is enough for the intruder to know the key $(\text{ATOM } \mathcal{L}, (b, K))^\sim$, or the material K and parameters b, in order to get both the plaintext M and the cryptographic parameters a. This is a strong assumption, however it is realistic. Many cryptographic algorithms put enough redundancy in the key (and even in the ciphertext), that it is possible to guess what algorithm has been used. Moreover algorithms are a few number, so even brute force attack on all known algorithms is feasible. One can argue that some symmetric encryption algorithms use an initialization vector, which is difficult to guess. This is true, however the assumption that the intruder can guess the initialization vector represents the worst case analysis, which is acceptable, and sometimes even desirable, when dealing with security protocols implementations.

However, if there is the need to represent cryptographic parameters secrecy as an obstacle to the intruder, the model can be represented as

$$\{\{M\}_{(\textsc{Atom}\ \mathcal{L},a)^{\sim}}\}_{(\textsc{Atom}\ \mathcal{L},(b,K))^{\sim}}. \tag{7}$$

With this representation, in order to obtain M, the intruder needs to know both the corresponding key $(\textsc{Atom}\ \mathcal{L},(b,K))^{\sim}$ and the parameters a. The key is still modeled as $(\textsc{Atom}\ \mathcal{L},(b,K))^{\sim}$ (and not, for example, as $(\{K\}_{(\textsc{Atom}\ \mathcal{L},b)^{\sim}})^{\sim}$) because keys are like hashes: they cannot be reverted to their material. If this would be possible, then from a public key K^{+} it would be possible to get the material K, and thus the corresponding private key K^{-}, and vice versa. It must be finally noticed that

$$\{\{M\}_{(\textsc{Atom}\ \mathcal{L},(b,K))^{\sim}}\}_{(\textsc{Atom}\ \mathcal{L},a)^{\sim}}$$

is different from (7). The former representation states that if the intruder knows the encryption parameters, then it can deduce some kind of information on the ciphertext; while the latter representation is the most intruder disadvantaging, stating that the key knowledge is priority with respect to parameters knowledge, that is, knowing the parameters is useless until the key is unknown. Which representation is best might depend on what model is willing to be verified.

It is believable that the results obtained in this work for the (6) model, also apply for the other two models.

4 Model simplifications

The models of implementation details, introduced above, allow to start from an abstract $SYSTEM$, and to obtain a more detailed, and complex, $SYSTEM'$, which takes into account encoding/decoding functions, and cryptographic parameters.

This section shows how the detailed model $SYSTEM'$ can be simplified, under some constraints involving the initial intruder knowledge IIK, into a model $SYSTEM^{s}$, as close as possible to $SYSTEM$, still preserving the security faults that were present in $SYSTEM'$.

This step becomes especially useful when $SYSTEM^{s}$ is equal to $SYSTEM$, and $SYSTEM$ has already been verified to be secure. Indeed, a common model-based workflow is to write the protocol specification $SYSTEM$ and to verify it. If one takes care to check that the required constraints on the initial intruder knowledge hold, then the correctness of $SYSTEM$ implies the correctness of the more detailed specification $SYSTEM'$, without even the need to write it.

Two separate simplifications will be showed in this work. The first is about removing the encoding/decoding layer; the second is about removing the cryptographic parameters. They can obviously both be applied to the same system.

4.1 Removing the encoding/decoding layer

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

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

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

$$IIK'' \triangleq IIK' \cup Encodings \cup \{\textsc{Atom}\ \mathcal{L}\} \tag{8}$$

The graphical representation of actors A and B in $SYSTEM''$ is given in figure 3.

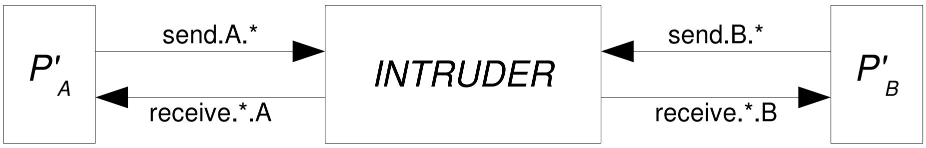

Fig. 3. Actors A and B with $INTRUDER$ in $SYSTEM''$.

Now the main result of this work is given.

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 tr is also a trace of $SYSTEM''$, and an attack exists in tr of $SYSTEM''$;*

$$SYSTEM'' \textbf{ sat } SPEC \Rightarrow SYSTEM' \textbf{ sat } SPEC,$$

provided that $SPEC$ is not defined on the send or receive events.

This is a general result, valid for any security requirement $SPEC$, not only strong secrecy or forward agreement, provided that $SPEC$ does not involve the *send* and *receive* channels, which is a reasonable constraint, since they are always assumed to be insecure.

Proof. In order to prove the theorem, a new $SYSTEM'''$ is defined. Figure 4 shows the diagram for actors A and B in this system.

$$SYSTEM''' \triangleq (|||_{A \in Honest} P'_A) || ENC\_INTRUDER$$

where

$$\begin{aligned} ENC\_INTRUDER \triangleq & (((|||_{A \in Honest} E_A(ext, int)) \\ & \underset{\{int\}}{\parallel} INTRUDER'(IIK')) \setminus \{int\}) \end{aligned} \tag{9}$$

and where $INTRUDER'(IIK') = INTRUDER(IIK')[[^{int}/_{ext}]]$ and $ext = send, receive$.

A lemma needed to complete the proof of this theorem is now introduced. It will be proven afterwards.

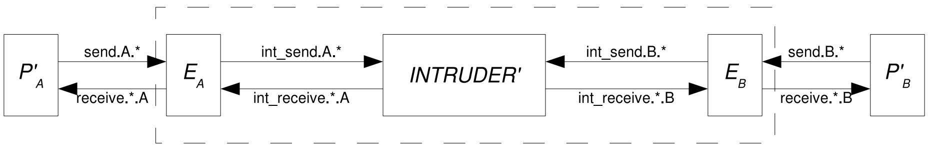

Fig. 4. Actors A and B with $ENC\_INTRUDER$ in $SYSTEM'''$.

Lemma 2. *Process $ENC\_INTRUDER$ as defined in (9) is a refinement of process $INTRUDER(IIK'')$, the intruder in $SYSTEM''$;*

$$INTRUDER(IIK'') \sqsubseteq ENC\_INTRUDER$$

If lemma 2 holds, then, by refinement properties exposed in [9], it follows that

$$SYSTEM'' \sqsubseteq SYSTEM'''$$

From the hypotheses it follows that a fault trace tr' is in $traces(SYSTEM')$. Furthermore, it is worth to notice that $SYSTEM'''$ is obtained from $SYSTEM'$, by swapping actual parameters of processes, and by injectively renaming communication channels of the $INTRUDER$ process. In particular, only the events *send* and *receive* are parameterized or renamed, and, since the renaming on $INTRUDER$ is injective, no non-determinism is introduced in the $INTRUDER$ process. So the trace $tr''' \in traces(SYSTEM''')$, that is obtained by renaming tr', is a fault trace too. Finally, since $SYSTEM'' \sqsubseteq SYSTEM'''$ holds, tr''' is also a fault trace in $SYSTEM''$.

Now the proof about lemma 2 is given.

Proof (of lemma 2). In order to carry out the proof, the states reacheble from $ENC\_INTRUDER$ must be written explicitly. For this reason, let ENC_i be defined as a generic state reachable from the $|||_{A \in Honest} E_A$ process. Moreover, for each state ENC_i, let S_{E_i} be the set of all the encoded messages ready to be delivered to the INTRUDER component of $ENC\_INTRUDER$, but not yet dispatched to it.

Formally, S_{E_i} is a set of messages that can be defined inductively. In the initial state ENC_0, we have $S_{E_0} = \emptyset$. The evolution of S_{E_i} after an event e can be represented by an extension of the $\xrightarrow{e}$ relation as follows: $S_{E_i} \xrightarrow{e} S_{E_j}$ means that the occurrence of e in a state where the set of ready encoded messages is S_{E_i} leads to a new state where the new set is S_{E_j}. Now, since the events e occurring in $ENCi$ can only take 4 different forms, the relation $\xrightarrow{e}$ on sets of messages can be defined by enumeration as follows: for any A, B, a, M, and function $e()$

following the definition of process E_A in (4),

$$
\begin{array}{lll}
S_{E_i} & \stackrel{send.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))}{\longrightarrow} & S_{E_i} \cup \{e(a,M)\} \\
S_{E_i} & \stackrel{receive.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))}{\longrightarrow} & S_{E_i} \\
S_{E_i} & \stackrel{int\_send.A.B.e(a,M)}{\longrightarrow} & S_{E_i} \setminus \{e(a,M)\} \\
S_{E_i} & \stackrel{int\_receive.A.B.y}{\longrightarrow} & S_{E_i}
\end{array}
$$

A generic state of $ENC\_INTRUDER$ then takes the form

$$ENC_i \parallel INTRUDER'(S'_i)$$

where S'_i is the current intruder knowledge. The initial state is $ENC\_INTRUDER = ENC_i \parallel INTRUDER'(S'_i)$, where $S'_0 = IIK'$.

Finally, the us define the total knowledge S_{T_i} associated with state $ENC_i \parallel INTRUDER'(S'_i)$ as

$$S_{T_i} \triangleq S_{E_i} \cup S'_i$$

It is worth to notice that, in state $ENC_i \parallel INTRUDER'(S'_i)$, $deds(S'_i) \subseteq deds(S_{T_i})$.

Lemma 3.

$$
\begin{array}{c}
\forall ENC_i, ENC_j, S'_i, S'_j \\
ENC_i \parallel INTRUDER'(S'_i) \xrightarrow{\tau} ENC_j \parallel INTRUDER'(S'_j) \\
\Longrightarrow \\
S_{T_i} = S_{T_j}
\end{array}
$$

Proof. There are two possible cases: τ comes from an int_send event, or τ comes from an $int\_receive$ event.

if τ comes from event $int\_send.A.B.e(a,M)$: then, by the definition of each process E_A, a corresponding $send.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))$ must have previously occurred. So, by the definition of S_{E_i},

$$e(a,M) \in S_{E_i}$$

and

$$S_{E_j} = S_{E_i} \setminus \{e(a,M)\}$$

and, by the definition of the $INTRUDER$ process,

$$S'_j = S'_i \cup \{e(a,M)\}$$

so $S_{T_i} = S_{T_j}$.

if τ comes from event $int\_receive.A.B.y$: then, by the definition of S_{E_i},

$$S_{E_j} = S_{E_i}$$

and, by the definition of the $INTRUDER$ process,

$$S'_j = S'_i$$

so $S_{T_i} = S_{T_j}$.

Now that lemma 3 is proven, the following property can be proven, which implies lemma 2:

$$\begin{array}{rl} & (ENC\_INTRUDER \stackrel{tr}{\Longrightarrow} ENC_f \parallel INTRUDER'(S'_f)) \\ & \Longrightarrow \\ \exists S''_f \| & (INTURDER(IIK'') \stackrel{tr}{\Longrightarrow} INTRUDER(S''_f) \quad \wedge \\ & deds(S_{T_f}) \subseteq deds(S''_f)) \end{array}$$

The proof is based on induction on the length of the trace tr.

Base ($length(tr) = 0$) Since tr is the empty trace,

$$ENC_0 \parallel INTRUDER'(S'_0) \stackrel{\tau^*}{\longrightarrow} ENC_f \parallel INTRUDER'(S'_f)$$

Then, by lemma 3, we have that

$$S_{T_f} = S_{T_0} = S_{E_0} \cup S'_0 = IIK'$$

Moreover, $INTRUDER(IIK'')$ cannot execute internal events, so

$$S''_f = IIK''$$

and, by definition (8), it follows that

$$IIK' \subset IIK''$$

so $deds(S_{T_f}) \subseteq deds(S''_f)$.

Induction ($length(tr) = n$) There are two possible cases: next event is a *send* event, or it is a *receive* event.

next event is a *send* event: then, by the definitions of processes, we have:

$$\begin{array}{c} ENC_n \parallel INTRUDER'(S'_n) \\ \stackrel{\tau^*}{\longrightarrow} \quad \stackrel{send.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))}{\longrightarrow} \quad \stackrel{\tau^*}{\longrightarrow} \\ ENC_{n+1} \parallel INTRUDER'(S'_{n+1}) \end{array}$$

By lemma 3, S_{T_i} will remain unchanged for each τ transition before or after the *send* event. Moreover,

$$S_{T_{n+1}} = S_{T_n} \cup \{e(a, M)\}$$

On the other end,

$$INTRUDER(S''_n) \overset{send.A.B.(\textsc{Atom}\ \mathcal{L},(a,M))}{\longrightarrow} INTRUDER(S''_{n+1})$$

where $S''_{n+1} = S''_n \cup \{(\textsc{Atom}\ \mathcal{L},(a,M))\}$
By inductive hypotheses, $deds(S_{T_n}) \subseteq deds(S''_n)$; by definition 5 of $e(a,M)$, it follows that $deds(e(a,M)) \subseteq deds((\textsc{Atom}\ \mathcal{L},(a,M)))$. So

$$deds(S_{T_n} \cup \{e(a,M)\}) \subseteq deds(S''_n \cup \{(\textsc{Atom}\ \mathcal{L},(a,M))\})$$

thus $deds(S_{T_{n+1}}) \subseteq deds(S''_{n+1})$.

case next event is a *receive* event: then, by the process definitions we have that

$$\begin{gathered} ENC_n \parallel INTRUDER'(S'_n) \\ \overset{\tau^*}{\longrightarrow} \overset{receive.A.B.(\textsc{Atom}\ \mathcal{L},(a,d(a,y)))}{\longrightarrow} \overset{\tau^*}{\longrightarrow} \\ ENC_{n+1} \parallel INTRUDER'(S'_{n+1}) \end{gathered}$$

By lemma 3, S_{T_n} will be equal for each τ transition before the *receive* event, and $S_{T_{n+1}}$ will be equal for each τ event after the *receive* event. Moreover

$$S_{T_{n+1}} = S_{T_n}$$

On the other end,

$$INTRUDER(S''_n) \overset{receive.A.B.(\textsc{Atom}\ \mathcal{L},(a,d(a,y)))}{\longrightarrow} INTRUDER(S''_{n+1})$$

where $S''_{n+1} = S''_n$.
So, by inductive hypotheses, $deds(S_{T_{n+1}}) \subseteq deds(S''_{n+1})$.
However, in order to complete the proof of this case, it must still be showed that the *receive* event indeed can happen in $INTRUDER(S''_n)$, that is, in order to send $(\textsc{Atom}\ \mathcal{L},(a,d(a,y)))$ on the *receive* channel, $INTRUDER(S''_n)$ must be able to derive the required message from its knowledge.
If a and y are derivable from S''_n, then, by definition (5), $d(a,y)$ is derivable, and, by applying the **pairing** rule, also $(a,d(a,y))$ is derivable. The message a is derivable from S''_n with the **member** rule, since

$$a \in Encodings \subset S''_0 \subseteq S''_n$$

The message y is derivable from S''_n because, since the *receive* event happens in $ENC_n \parallel INTRUDER'(S'_n)$, then

$$y \in deds(S_{T_n})$$

and, by inductive hypotheses,

$$deds(S_{T_n}) \subseteq deds(S''_n)$$

thus $y \in deds(S''_n)$.
Finally, if $\textsc{Atom}\ \mathcal{L}$ is derivable from S''_n, then, by applying the **pairing** rule, the required $(\textsc{Atom}\ \mathcal{L},(a,d(a,y)))$ message can be obtained. The message $\textsc{Atom}\ \mathcal{L}$ is derivable from S''_n with the **member** rule, because

$$\textsc{Atom}\ \mathcal{L} \in S''_0 \subseteq S''_n$$

Discussion Theorem 1 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 $Encodings \subset IIK''$). 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.

In real protocols, there may be attacks where the intruder can generate fault traces, without even knowing the protocol encoding scheme. Indeed, the model proposed here does take into account this kind of attacks. To get the set of such attacks, it is enough to consider only fault traces where the intruder does not need to use any information from any member of *Encodings*. Furthermore, the model can be checked for other kinds of attacks, by verifying the protocol against a powerful intruder, that is able to recover the content of any encoded message it receives.

Finally, in real application implementations, if the protocol logic can be proven to adhere to a formally verified specification, and if there is a well defined interface between the protocol logic and the encoding/decoding layer, then, if the encoding/decoding functions (which *are part* of the intruder) can only access the message that must be encoded and if they do not contain any hard-coded secret, then the whole implementation is proven to be correct, no matter what the implementations of the encoding/decoding functions are. The whole information contained by the code implementing the encoding functions is taken into account in the model within the parameter a, known by the intruder - and this is why, not surprisingly, there must be no hard-coded secrets in the encoding functions.

4.2 Simplifying the encoding/decoding model

The last step that is useful to perform when dealing with the encoding/decoding layer, is to simplify each P'_A, trying to get a simplified process P^s_A as close to P_A as possible. As previously explained, the optimum case is when P^s_A is equal to P_A.

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.

Before explaining how P_A can be obtained from P'_A, and thus $SYSTEM$ from $SYSTEM''$, it is worth to note that, by theorem 1, it follows that $SYSTEM''$ can be checked for all security properties that can be checked on $SYSTEM'$ (provided they are not defined on the *send* or *receive* events, and that the intruder knows the encoding parameters). However, when passing from $SYSTEM''$ to $SYSTEM$, the faults that are currently proven to be preserved are only those regarding strong secrecy and forward agreement specifications. That is, if $SYSTEM$ is verified instead of $SYSTEM'$, then all possible implementation details represented in $SYSTEM'$, are verified against strong secrecy and agreement. On the basis of the results achieved in this paper, it is still required to explicitly verify $SYSTEM'$ (or, equivalently, $SYSTEM''$) in order to verify other security properties.

Simplifying transformations A fault-preserving simplifying transformation 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 a simplifying transformation $f(\cdot)$ satisfies equations

$$\begin{aligned} &\forall\, B \in \mathbf{P}(NFMessage); M \in NFMessage \cdot \\ &\quad B \cup IIK \vdash M \Rightarrow f(B) \cup IIK' \vdash f(M) \end{aligned} \tag{10}$$

and

$$f(IIK) \subseteq IIK' \tag{11}$$

then

$$\forall\, tr \in traces(SYSTEM) \cdot f(tr) \in traces(SYSTEM')$$

and

$$SYSTEM' \textbf{ sat } Secrecy \Rightarrow SYSTEM \textbf{ sat } Secrecy.$$

The work in [7] also gives a sufficient condition about $f(\cdot)$ and *AgreementSet*, for $f(\cdot)$ to be a fault-preserving transformation with respect to authentication agreement specifications. This condition requires $f(\cdot)$ to be injective for all message sequences belonging to *AgreementSet*, that is

$$\begin{aligned} &\forall\, Ms \in AgreementSet; Ms' \in Message^{*} \cdot \\ &\quad Ms \neq Ms' \Rightarrow f(Ms) \neq f(Ms') \end{aligned}$$

However, it is very difficult to find the constraints under which a simplifying transformation satisfies this condition. Indeed, even the examples of simplifying transformations given in [7], although they are said to satisfy this condition under some constraints, actually they do not satisfy it.

In this work, a new, weaker sufficient condition for fault preserving transformations with respect to authentication agreement specifications is provided. This condition requires less constraints on the simplifying transformation, so they are easier to be found. For example, the transformations proposed in [7], along with the given constraints, can be proven to satisfy the new weaker condition, thus proving for the first time that they are safe with respect to agreement.

The new sufficient condition states that

$$\begin{aligned} &\forall\ Ms, Ms' \in AgreementSet \cdot \\ &Ms \neq Ms' \Rightarrow f(Ms) \neq f(Ms') \end{aligned} \tag{12}$$

That is, $f(\cdot)$ must be *locally* injective. In other words, for each message M in the *AgreementSet*, the message $f(M)$ can be obtained only from the message M in the *AgreementSet*, but may also be obtained from zero or more messages that do not belong to the *AgreementSet*.

Lemma 4. *If equation* (12) *is satisfied, then if a particular trace* tr *constitutes a failure of agreement authentication on the original protocol, then* $f(tr)$ *constitutes a failure of agreement authentication on the simplified protocol:*

$$\neg Agreement_{AgreementSet}(tr) \Rightarrow \neg Agreement_{f(AgreementSet)}(f(tr))$$

Proof. Suppose tr constitutes a failure of agreement authentication on the original protocol: $\neg Agreement_{AgreementSet}(tr)$. Then for some $A \in Agent$, $B \in Honest$, $Ms \in AgreementSet$, we have:

$$tr \downarrow finished.A.B.Ms > tr \downarrow running.B.A.Ms$$

By the condition (12) on f:

$$\begin{aligned} f(tr) \downarrow finished.A.B.f(Ms) &= tr \downarrow finished.A.B.Ms + \\ &\sum_{Ms' \notin AgreementSet} tr \downarrow finished.A.B.Ms' \\ f(tr) \downarrow running.B.A.f(Ms) &= tr \downarrow running.B.A.Ms + \\ &\sum_{Ms' \notin AgreementSet} tr \downarrow running.B.A.Ms' \end{aligned}$$

However, by the definition (3) of *AgreementSet*,

$$\begin{aligned} &\forall\ Ms' \in Message^*; tr \in traces(P); A \in Agents; B \in Honest \cdot \\ &\quad Ms' \notin AgreementSet \Rightarrow \\ &\quad tr \downarrow finished.A.B.Ms' = 0 \wedge tr \downarrow running.B.A.Ms' = 0 \end{aligned}$$

so

$$\begin{aligned} &\sum_{Ms' \notin AgreementSet} tr \downarrow finished.A.B.Ms' = \\ = &\sum_{Ms' \notin AgreementSet} tr \downarrow running.B.A.Ms' = \\ &= 0 \end{aligned}$$

From this it follows that

$$f(tr) \downarrow finished.A.B.f(Ms) > f(tr) \downarrow running.B.A.f(Ms)$$

Removing a message or an encryption A fault-preserving simplifying transformation that allows to remove a generic *Message* or an encryption from the specification of a protocol is now introduced. It is worth to note that the *Message* being removed can be used as a key, that is, it is also possible to remove keys from the specification.

The set of messages to be removed is $Messages \subseteq Message$, while the set of encryptions to be removed is $Encs \subseteq Message$. The goal is that for each message M in $Messages$, it is substituted by $\textsc{Atom}\ \mathtt{nil}$; for each message $\{M\}_{K^\sim}$, $\{[M]\}_{K^+}$ or $[\{M\}]_{K^-}$ in $Encs$, it is substituted by M.

In order for this transformation to be fault-preserving with respect to secrecy and authentication, some constraints on what is removed will be necessary. For example, if a message $K^\sim$ is removed, then all encryptions $\{x\}_{K^\sim}$ done with that key must also be removed, otherwise the attacker would lose the ability to discover the plaintext x.

Here follows the definition of $f(\cdot)$:

$$
\begin{aligned}
&\text{if } M \in Ms:\\
&\qquad f(M) \triangleq \textsc{Atom}\ \mathtt{nil},\\
&\text{if } M \notin Ms:\\
&f(\textsc{Atom}\ A) \triangleq \textsc{Atom}\ A,\\
&f(M, M') \triangleq \begin{cases} f(M'), & \text{if } f(M) = \textsc{Atom}\ \mathtt{nil},\\ f(M), & \text{if } f(M') = \textsc{Atom}\ \mathtt{nil},\\ (f(M), f(M')), & \text{otherwise,} \end{cases}\\
&f(\{M\}_{K^\sim}) \triangleq \begin{cases} f(M), & \text{if } \{M\}_{K^\sim} \in Encs,\\ \textsc{Atom}\ \mathtt{nil}, & \text{if } f(M) = \textsc{Atom}\ \mathtt{nil},\\ \{f(M)\}_{f(K)^\sim}, & \text{otherwise,} \end{cases}\\
&f(\{[M]\}_{K^+}) \triangleq \begin{cases} f(M), & \text{if } \{[M]\}_{K^+} \in Encs,\\ \textsc{Atom}\ \mathtt{nil}, & \text{if } f(M) = \textsc{Atom}\ \mathtt{nil},\\ \{[f(M)]\}_{f(K)^+}, & \text{otherwise,} \end{cases}\\
&f([\{M\}]_{K^-}) \triangleq \begin{cases} f(M), & \text{if } [\{M\}]_{K^-} \in Encs,\\ \textsc{Atom}\ \mathtt{nil}, & \text{if } f(M) = \textsc{Atom}\ \mathtt{nil},\\ [\{f(M)\}]_{f(K)^-}, & \text{otherwise,} \end{cases}\\
&f(H(M)) \triangleq \begin{cases} \textsc{Atom}\ \mathtt{nil}, & \text{if } f(M) = \textsc{Atom}\ \mathtt{nil},\\ H(f(M)), & \text{otherwise,} \end{cases}\\
&f(K^\sim) \triangleq \begin{cases} \textsc{Atom}\ \mathtt{nil}, & \text{if } f(K) = \textsc{Atom}\ \mathtt{nil},\\ f(K)^\sim, & \text{otherwise.} \end{cases}\\
&f(K^+) \triangleq \begin{cases} \textsc{Atom}\ \mathtt{nil}, & \text{if } f(K) = \textsc{Atom}\ \mathtt{nil},\\ f(K)^+, & \text{otherwise,} \end{cases}\\
&f(K^-) \triangleq \begin{cases} \textsc{Atom}\ \mathtt{nil}, & \text{if } f(K) = \textsc{Atom}\ \mathtt{nil},\\ f(K)^-, & \text{otherwise.} \end{cases}
\end{aligned}
$$

The initial intruder knowledge for the simplified system is set to

$$IIK' = f(IIK) \cup \{\textsc{Atom}\ \mathtt{nil}\} \tag{13}$$

which clearly satisfies equation (11).

In order for $f(\cdot)$ to be fault-preserving with respect to secrecy, it must still be verified that it satisfies equation (10). This is not true in general. However, putting a few constraints on the content of $Messages$ and $Encs$, it can be proved that $f(\cdot)$ satisfies (10). The required constraints are:

$$(M, M') \in Messages \Rightarrow M \in Messages \wedge M' \in Messages \tag{14}$$

$$\forall\ K^{\sim}, \forall\ M, f(K^{\sim}) = \textsc{Atom}\ \texttt{nil} \Rightarrow \{M\}_{K^{\sim}} \in Encs \tag{15a}$$
$$\forall\ K^{+}, \forall\ M, f(K^{+}) = \textsc{Atom}\ \texttt{nil} \Rightarrow [\{M\}]_{K^{-}} \in Encs \tag{15b}$$
$$\forall\ K^{-}, \forall\ M, f(K^{-}) = \textsc{Atom}\ \texttt{nil} \Rightarrow \{[M]\}_{K^{+}} \in Encs \tag{15c}$$

$$\{M\}_{K^{\sim}} \in Messages \Rightarrow M \in Messages \tag{16a}$$
$$\{[M]\}_{K^{+}} \in Messages \Rightarrow M \in Messages \tag{16b}$$
$$[\{M\}]_{K^{-}} \in Messages \Rightarrow M \in Messages \tag{16c}$$

$$\forall\ K^{+}, \forall\ M, f(K^{+}) = \textsc{Atom}\ \texttt{nil} \Rightarrow \{[M]\}_{K^{+}} \in Encs \tag{17a}$$
$$\forall\ K^{-}, \forall\ M, f(K^{-}) = \textsc{Atom}\ \texttt{nil} \Rightarrow [\{M\}]_{K^{-}} \in Encs \tag{17b}$$

Constraint (14) tells that if a Pair is removed, then also its constituent messages must be removed. This is needed because an attacker receiving (M, M') is able to deduce M and M'; if (M, M') is removed, the attacker loses this ability, unless also M and M' are removed from the protocol.

Constraints of group (15) tell that if a key $K^{\sim}$, K^{+}or K^{-}, collapse to Atom `nil`, then all encryptions that could be decrypted with that key must be removed. This is necessary, because if a key is removed, then the attacker is not able to recover encryptions that can be decrypted with that key, unless the plaintext of that encryptions is explicitly revealed.

Constraints of group (16) tell that for any encryption that is removed from the protocol, then also the plaintext of that encryption must be removed. Suppose an attacker has $K^{\sim}$, and receives $\{M\}_{K^{\sim}}$, then it can deduce M. However, if $\{M\}_{K^{\sim}}$ is removed, the attacker cannot recover M, unless also M is removed from the protocol.

Finally, constraints of group (17) tell that if a public (private) key collapse to Atom `nil`, then all encryptions made with that key must be removed.

The reason why both constraints (15b), (15c), and (17a), (17b) are required, is that they are actually dual. One the one hand, like explained above, if a key is removed, then every message encrypted with the key's counterpart would be unrecoverable; this issue is solved by constraints (15b) and (15c). On the other hand, if a key is removed, then it makes no sense to encrypt any message with that key. Actually encrypting a message with a removed key must be the same as revealing that message; this is the meaning of constraints (17a) and (17b). Finally, it can be noticed that for symmetric encryption, only constraint (15a) is required, because the same key $K^{\sim}$ is used to encrypt and decrypt a message.

Lemma 5 is now introduced. It is necessary to prove that $f(\cdot)$ satisfies equation (11).

Lemma 5. *If M is not a* PAIR, *then $f(M)$ is not a* PAIR.

$$\neg pair(M) \Rightarrow \neg pair(f(M))$$

Proof. By the definition of $f(\cdot)$, if M is not a pair, then all possible values of $f(M)$ are not pair, so $f(M)$ cannot be a pair.

Proof (of $f(\cdot)$ satisfying equation (11)). The proof is based on induction and case analysis over the knowledge derivation relation $\vdash$.

Only the proofs for some rules of the knowledge derivation relation are reported here. Proofs for the other rules can be carried out by using the same reasoning showed here.

member $M \in B \Rightarrow f(M) \in f(B) \Rightarrow f(B) \vdash f(M)$, thus $f(B) \cup IIK' \vdash f(M)$

pairing $B \vdash M \wedge B \vdash M' \wedge \neg pair(M)$. By inductive hypotheses and lemma 5 $f(B) \cup IIK' \vdash f(M) \wedge f(B) \cup IIK' \vdash f(M') \wedge \neg pair(f(M))$. It must be proved that $f(M, M')$ can be derived from this knowledge.
- Case $f(M) =$ ATOM `nil`: by the definition of $f(\cdot)$, $f(B) \cup IIK' \vdash f(M') = f(M, M')$.
- Case $f(M') =$ ATOM `nil`: by the definition of $f(\cdot)$, $f(B) \cup IIK' \vdash f(M) = f(M, M')$.
- Case $f(M) \neq$ ATOM `nil` $\wedge f(M') \neq$ ATOM `nil`: by applying the **pairing** rule, $f(B) \cup IIK' \vdash (f(M), f(M')) = f(M, M')$.

splitting $b \vdash (M, M')$. By inductive hypotheses $f(B) \cup IIK' \vdash f(M, M')$. It must be proved that $f(M)$ and $f(M')$ can be derived from this knowledge.
- Case $(M, M') \in Ms$: by contraint (14) $M \in Ms$ and $M' \in Ms$, so $f(M) = f(M') =$ ATOM `nil`. By the definition (13) of IIK' it follows that $f(B) \cup IIK' \vdash$ ATOM `nil`.
- Case $(M, M') \notin Ms$:
  - Case $f(M) =$ ATOM `nil`: by hypotheses and definition of IIK it follows that $f(B) \cup IIK' \vdash f(M) =$ ATOM `nil`. Moreover, by the definition of $f(\cdot)$, $f(B) \cup IIK' \vdash f(M') = f(M, M')$.
  - Case $f(M') =$ ATOM `nil`: by hypotheses and definition of IIK it follows that $f(B) \cup IIK' \vdash f(M') =$ ATOM `nil`. Moreover, by the definition of $f(\cdot)$, $f(B) \cup IIK' \vdash f(M) = f(M, M')$.
  - Case $f(M) \neq$ ATOM `nil` $\wedge f(M') \neq$ ATOM `nil`: by the definition of $f(\cdot)$ it follows that $f(M, M') = (f(M), f(M'))$. By applying the **splitting** rule, $f(B) \cup IIK' \vdash f(M) \wedge f(B) \cup IIK' \vdash f(M')$.

The same reasoning as above applies to all other knowledge derivation rules.

Now it is worth to show how $f(\cdot)$ can be proven to be fault-preserving with respect to agreement, under some constraints. The proof can be kept simple because of the condition (12), which only requires local injectivity.

If the following constraints

$$\begin{aligned}
\forall\ M, \forall Ms, M \in Ms, Ms &\in \mathit{AgreementSet} \\
M &\notin Messages \\
M &\notin \mathit{Encs} \\
M = (M', M'') &\Rightarrow f(M') \neq \textsc{Atom}\ \texttt{nil} \wedge f(M'') \neq \textsc{Atom}\ \texttt{nil} \\
M = \{M'\}_{K^\sim} &\Rightarrow f(M') \neq \textsc{Atom}\ \texttt{nil} \\
M = \{[M]'\}_{K^+} &\Rightarrow f(M') \neq \textsc{Atom}\ \texttt{nil} \\
M = [\{M'\}]_{K^-} &\Rightarrow f(M') \neq \textsc{Atom}\ \texttt{nil} \\
M = H(M') &\Rightarrow f(M') \neq \textsc{Atom}\ \texttt{nil} \\
M = K^\sim &\Rightarrow f(K) \neq \textsc{Atom}\ \texttt{nil} \\
M = K^+ &\Rightarrow f(K) \neq \textsc{Atom}\ \texttt{nil} \\
M = K^- &\Rightarrow f(K) \neq \textsc{Atom}\ \texttt{nil}
\end{aligned}$$

hold, then $f(\cdot)$ is locally injective on the messages that belong to the *AgreementSet*. The meaning of the required constraints is, in general, that if agreement is required on a message, then that message, and all of its parts, must not be removed. For example, if agreement is required on a message M, then M must not be removed from the protocol. In the same way, if agreement is required on $\{M\}_{K^\sim}$, then that encryption must not be removed. If agreement is required on a Pair, then none of its constituent parts must be removed. The same reasoning applies for all other constraints.

It is worth to note that under these conditions, and by restricting the domain to all messages in the *AgreementSet*, $f(\cdot)$ can be rewritten as

$$\begin{aligned}
f(\textsc{Atom}\ A) &= \textsc{Atom}\ A \\
f(M, M') &= (f(M), f(M')) \\
f(\{M\}_{K^\sim}) &= \{f(M)\}_{f(K)^\sim} \\
f(\{[M]\}_{K^+}) &= \{[f(M)]\}_{f(K)^+} \\
f([\{M\}]_{K^-}) &= [\{f(M)\}]_{f(K)^-} \\
f(H(M)) &= H(f(M)) \\
f(K^\sim) &= f(K)^\sim \\
f(K^+) &= f(K)^+ \\
f(K^-) &= f(K)^-
\end{aligned} \tag{18}$$

which can be proven to be injective by structural induction on messages, thus satisfying condition (12).

Coalescing Pairs A fault-preserving renaming transformation that is required in order to transform P'_A into P_A is now introduced. This transformation collapses a pair (M, M') into its first item M. This transformation is similar in purpose to the one described in [7]. However, some differences must be pointed out:

- The definition of the renaming function is updated to the new datatype.
- Different constraints are given for $f(\cdot)$ to be fault-preserving with respect to secrecy.
- For the first time this kind of function is proven to be fault-preserving with respect to authentication on *any* message sequence (and not only with respect to message sequences made of atoms, as in [7]), and the required constraints are formally specified.

Let $Pairs$ be the set of pairs (M, M') that must be coalesced to M. The updated definition of $f(\cdot)$ is

$$\begin{aligned}
f(\textsc{Atom}\ A) &= \textsc{Atom}\ A, \\
f(M, M') &= \begin{cases} f(M), & \text{if } (M, M') \in Pairs \wedge \neg Pairs(M'), \\ (f(M), f(M')) & \text{if } (M, M') \notin Pairs \wedge \neg Pairs(M'), \end{cases} \\
f(M, (M', M'')) &= \begin{cases} f(M, M'') & \text{if } (M, M') \in Pairs, \\ (f(M), f(M', M'')) & \text{otherwise,} \end{cases} \\
f(\{M\}_{K^{\sim}}) &= \{f(M)\}_{f(K)^{\sim}} \\
f(\{[M]\}_{K^{+}}) &= \{[f(M)]\}_{f(K)^{+}} \\
f([\{M\}]_{K^{-}}) &= [\{f(M)\}]_{f(K)^{-}} \\
f(H(M)) &= H(f(M)) \\
f(K^{\sim}) &= f(K)^{\sim} \\
f(K^{+}) &= f(K)^{+} \\
f(K^{-}) &= f(K)^{-}
\end{aligned}$$

The initial intruder knowledge in the simplified system is

$$IIK' = f(IIK) \cup \{f(M') | (M, M') \in Pairs\}$$

which clearly satisfies condition (11). That is, the intruder must already know the information that is going to be coalesced away.

With the given definition of IIK', and with no other constraint, it can also be proven that $f(\cdot)$ satisfies condition (10), thus being fault-preserving with respect to strong secrecy.

In order to prove condition (12), only one additional constraint is required:

$$(M, M') \in AgreementSet \Rightarrow (M, M') \notin Pairs \tag{19}$$

If constraint (19) holds, and if the domain of $f(\cdot)$ is restricted to $AgreementSet$, then $f(\cdot)$ can be rewritten as (18). Then, with structural induction over messages, it can be proven to be locally injective, thus satisfying condition (12).

The meaning of constraint (19) is that if agreement is required on a pair, then that pair must not be coalesced away. It is worth to note, however, that it is still possible to have agreement on M', even if (M, M') is in $Pairs$ (that is, if M' is going to be coalesced). Indeed, M' will be removed only when it appears in (M, M'), and not everywhere in the process, so agreement can still take place.

Applying transformations to P'_A 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 $SYSTEM$.

Two steps are required in order to obtain this result.

1. $P_A^{tmp} = f(P'_A)$, with $Pairs = \{(\text{ATOM } \mathcal{L}, a) | a \in Encodings\}$
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 is safe if the required sufficient conditions hold.

For the transformation applied in step 1, strong secrecy requires that

$$IIK = IIK' \cup Encodings$$

This constraint is always satisfied, since, by theorem 1, in

$$SYSTEM'' : Encodings \subset IIK'$$

For authentication, it is required that

$$Pairs \cap AgreementSet = \emptyset$$

This condition holds, because each element in $Pair$ has the form $(\text{ATOM } \mathcal{L}, a)$. Indeed, ATOM $\mathcal{L}$ can never appear in any *running* or *finished* event, and thus in 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.

The transformation in step 2 requires, for secrecy, that $IIK = IIK' \cup \{\text{ATOM } \mathcal{L}\}$, which is satisfied again by theorem 1. For authentication, it is required that $Pairs \cap AgreementSet = \emptyset$, which is satisfied by using the same reasoning as above.

4.3 Simplifying the cryptographic parameters model

If a process P' is obtained by P by replacing all instances of $send.A.B.\{M\}_{K^\sim}$ with

$$send.A.B.\{\text{ATOM } \mathcal{L}, (a, M)\}_{(\text{ATOM } \mathcal{L}, (b, K))^\sim}$$

(the same for asymmetric cryptography) and all instances of $send.A.B.H(M)$ with $send.A.B.H(\text{ATOM } \mathcal{L}, (a, M))$ (and the same with all instances of the *receive* event), then it is possible to apply a sequence of simplifying transformations on P', such that the simplified process is equal to P.

The required steps are the same as the ones required in order to simplify the model representing the encoding/decoding layer, but the *Encodings* set is replaced by the *Parameters* set. Again, it is required that IIK in the system containing P is equal to $IIK' \cup Parameters \cup \{\textsc{Atom}\ \mathcal{L}\}$, where IIK' is the initial intruder knowledge for the system containing P'. The reasoning justifying *Encodings* and $\textsc{Atom}\ \mathcal{L}$ being part of the intruder knowledge also applies for *Parameters* and $\textsc{Atom}\ \mathcal{L}$ being part of IIK.

If the other two proposed models of encryption parameters are used, then it is possible to apply on P' the previously introduced renaming transformation that removes some specified encryptions, namely the encryptions where the keys are in the set $\{(\textsc{Atom}\ \mathcal{L}, a)^{\sim} | a \in Parameters\}$.

5 An example

In order to show how the methodologies proposed in this paper apply to real protocols, both an abstract CSP model of a client for the SSH Transport Layer Protocol[10–12] and its refined model (including both encoding layer and cryptographic parameters) are presented in this section. The goal is to show an example of how the refined model can be obtained from the abstract model, and how the abstract model can be obtained again by simplifying the refined model.

Indeed, in real model-based development, the steps presented here are unnecessary, since in this paper it is showed that there is no need to generate the refined model, because, under some given conditions, it is enough to verify the abstract model, in order to also verify the refined model.

Instead, the example proposed here aims to show how the theoretical results obtained in this paper take shape when applied to a real case.

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

A possible abstract model of an SSH Transport Layer Protocol client is the following:

$$
\begin{array}{l}
SSHClient(IDC, me, you, CAlgs, TrustedKeys) = \\
\quad send!me!you!IDC \rightarrow \\
\quad receive!you!me?IDS \rightarrow \\
\quad \sqcap_{cookieC \in Cookies}\ send!me!you!(cookieC, CAlgs) \rightarrow \\
\quad receive!you!me?(cookieS, SAlgs) \rightarrow \\
\quad g := Negotiate(CAlgs, SAlgs, \text{`}g\text{'}) \in Parameters \\
\quad p := Negotiate(CAlgs, SAlgs, \text{`}p\text{'}) \in Parameters \\
\quad \sqcap_{x \in DHSecrets}\ send!me!you!EXP(g, x, p) \rightarrow \\
\quad receive!you!me?(KeyS^{+}, DHPublicS, [\{sshash\}]_{KeyS^{-}}) \rightarrow \\
\quad GO(EXP(DHPublicS, x, p), sshash, KeyS^{+}, TrustedKeys)
\end{array}
\tag{20}
$$

The *SSHClient* process begins a protocol session with the server by sending to it IDC, which is the client identification string. 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 with the $Negotiate(CAlgs, SAlgs, Param)$ function, which returns the requested negotiated algorithm parameter $Param$, obtained from the supported client and server algorithms $CAlgs$ and $SAlgs$. Once g and p are obtained, the client sends $EXP(g, x, p)$ (its DH public key, as explained below), which is a message representing the modular exponentiation function, that computes $e = g^x \bmod p$. This message is added to the datatype and is defined as

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

along with the following 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

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

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 property:

$$EXP(EXP(g, y, p), x, p) = EXP(EXP(g, x, p), y, p) \tag{21}$$

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 (21) 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. Then, 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 relevant data of a protocol session.

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

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 computed by the client, that is the session secret shared between the client and the server. Finally, the $GO(\cdot)$ process is defined as

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

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*, $sshash$.

Now that the abstract model (20) of the client has been introduced, it is possible to show the refined model, which takes into account both the encoding/decoding layer, and the cryptographic parameters. This model is obtained by following the steps described in section 3.1:

$$\begin{aligned}
&SSHClientRef(IDC, me, you, CAlgs, TrustedKeys) = \\
&\quad send!me!you!(\text{ATOM } \mathcal{L}, (string, IDC)) \rightarrow \\
&\quad receive!you!me?(\text{ATOM } \mathcal{L}, (string, IDS)) \rightarrow \\
&\quad \sqcap_{cookieC \in Cookies}\, send!me!you!(\text{ATOM } \mathcal{L}, (KEX, (cookieC, CAlgs))) \rightarrow \\
&\quad receive!you!me?(\text{ATOM } \mathcal{L}, (KEX, (cookieS, SAlgs))) \rightarrow \\
&\quad g := Negotiate(CAlgs, SAlgs, \text{`}g\text{'}) \in Parameters \\
&\quad p := Negotiate(CAlgs, SAlgs, \text{`}p\text{'}) \in Parameters \\
&\quad \sqcap_{x \in DHSecrets}\, send!me!you!(\text{ATOM } \mathcal{L}, ((bin_pack, mpint), EXP(g, x, p))) \rightarrow \\
&\quad receive!you!me?(\text{ATOM } \mathcal{L}, ((bin_pack, string, mpint, string), \\
&\qquad (KeyMat^{+}, DHPublicS, \\
&\qquad [\{(\text{ATOM } \mathcal{L}, ((SignMode, SignPadding), sshash_enc))\}]_{KeyMat^{-}}))) \rightarrow \\
&\quad GO(EXP(DHPublicS, x, p), sshash_enc, KeyMat^{+}, TrustedKeys)
\end{aligned} \tag{22}$$

where:

$$\begin{aligned}
KEX &= (bin_pack, bytes, namelists) \\
KeyMat &= (\text{ATOM } \mathcal{L}, (SignKeyType, KeyS)) \\
sshash_enc &= H(\text{ATOM } \mathcal{L}, (SignHash, \\
&\qquad (H(\text{ATOM } \mathcal{L}, (FinalHash, \\
&\qquad (IDC, IDS, (cookieC, CAlgs), (cookieS, SAlgs), \\
&\qquad KeyMat^{+}, EXP(g, x, p), DHPublicS, \\
&\qquad EXP(DHPublicS, x, p))))))) \\
Encodings &= \{string, bin_pack, bytes, namelists, mpint\} \\
Parameters &= \{CAlgs\} \cup \{SAlgs\}
\end{aligned}$$

It must be noticed that $CAlgs$ and $SAlgs$ contain all supported parameters. In particular they contain the negotiated parameters g, p, $FinalHash$, $SignHash$, $SignKeyType$, $SignMode$ and $SignPadding$, which, for this reason, are in the $Parameters$ set too. Moreover, the keys in the $TrustedKeys$ set are also updated to the refined model.

In this refined model, the difference between the internal *final hash* required by the protocol, and the external hash required by the signature algorithm is more clear. The internal hash uses the $FinalHash$ parameter, which represents the hashing algorithm negotiated for the *final hash*; while the external hash uses the $SignHash$ parameter, which represents the hashing algorithm associated with the signature algorithm that has been negotiated.

Finally, it is worth to notice that both g and p are in $Parameters$ (because they have been negotiated, and thus are in $CAlgs$ and $SAlgs$), but, differently from other cryptographic parameters, they are not added to the refined model. The reason is that the exponentiation cryptographic parameters are already represented completely in the abstract model (and not only if they are being negotiated or agreed upon): without them, the exponentiation property (21) could not be expressed. Since these cryptographic parameters need to be fully represented in the abstract model, there is no need to add them again in the refined model.

Finally, by applying the simplifying transformations as described in sections 4.2 and 4.3, the $SSHClient$ process can be obtained again from the $SSHClientRef$ process. When simplifying the cryptographic parameters, the messages ($\text{ATOM}\ \mathcal{L}, g$) and ($\text{ATOM}\ \mathcal{L}, p$) will be in $Pairs$, even if these parameters have not been added explicitly during refinement. However this is not an issue, because the messages ($\text{ATOM}\ \mathcal{L}, g$) and ($\text{ATOM}\ \mathcal{L}, p$) cannot be coalesced away, because they never appear in $SSHClientRef$. This is because the syntactic marker $\text{ATOM}\ \mathcal{L}$ is never used in $SSHClient$, and those messages have not been added during refinement.

5.1 Discussion

The refined model is more complex and less readable than the abstract model. This means that more time is required to verify its security properties, and more errors can be introduced in the model itself. If strong security or forward authentication are going to be verified, then this paper shows that it is possible to verify the abstract model, provided that the intruder knows all the encoding and cryptographic parameters.

In practice, if the protocol logic implementation code is automatically generated from the abstract model, and is proven to be correct, then if the encoding functions can only access the data that they must encode, then the whole implementation is implied to be correct (but no functional, indeed), with no other check on the encoding layer implementation.

Instead, when following the model extraction approach, if a model is extracted from a full blown implementation, then modelling the protocol logic is enough, and the encoding layer can be ignored, provided it only access the given data.

6 Conclusions

The work presented in this paper represents a step towards the verification of refined security protocol models, which take encoding layers and cryptographic parameters into account, thus allowing the verification of important implementation details that can affect the security properties of the model.

The main innovative result, is that, in order to verify strong secrecy or forward authentication properties on a refined model, it is enough to verify those properties on an abstract model and to verify that some easy to check conditions on the intruder knowledge hold. Alternatively, in order to check a generic security property defined on protocol traces, a simplified refined system can be used, instead of the original one.

This theoretical result has many immediate practical applications.

At first, for any abstract model where the intruder knows the encoding or cryptographic parameters, if this abstract model is already verified to be secure with respect to strong secrecy or forward authentication, then the corresponding refined model taking into account the encoding layer or cryptographic parameters, is now implied to be secure, with no other effort required.

Specifically, with model-based development, it is now enough to write an abstract model and to give the intruder enough knowledge, in order to take into account all protocol issues concerning the encoding functions and the cryptographic parameters and algorithms. On the other hand, with model extraction approaches, instead of extracting from code a complex model that tries to handle all implementation details, now it is possible to extract a simpler model, and then to verify some easy to check constraints on the implementation code.

Finally, in order to obtain the results achieved in this paper, some general extensions to the results presented in [7] have been necessary. While these extensions are immediately useful in this work, it is worth to remember that they are also stand alone achievements that can be used as a starting point for future works.

Some issues on the topics presented in this paper are still opened for future work. In particular, it will be interesting to explore under which conditions the abstract model can be verified to satisfy a generic security property, such that it implies that the refined model also satisfies the given property.

Nevertheless, the results achieved in this work are immediately useful and applicable to real model-based development environments, model extraction approaches, or automatic code generation from abstract formal models.

References

1. Millen, J.K., Shmatikov, V.: Symbolic protocol analysis with an abelian group operator or diffie-hellman exponentiation. Journal of Computer Security **13**(3) (2005) 515–564
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography (the computational soundness of formal encryption). Journal of Cryptology **15**(2) (2002) 103–127

3. Pozza, D., Sisto, R., Durante, L.: Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In: International Conference on Advanced Information Networking and Applications. (2004) 400–405
4. Tobler, B., Hutchison, A.: Generating network security protocol implementations from formal specifications. In: Certification and Security in Inter-Organizational E-Services, Toulouse, France (2004)
5. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. In: Computer Security Foundations Workshop. (2006) 139–152
6. Jürjens, J.: Verification of low-level crypto-protocol implementations using automated theorem proving. In: Formal Methods and Models for Co-Design. (2005) 89–98
7. Hui, M.L., Lowe, G.: Fault-preserving simplifying transformations for security protocols. Journal of Computer Security **9**(1/2) (2001) 3–46
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
9. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)
10. Lehtinen, S., Lonvick, C.: The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250 (Proposed Standard) (January 2006)
11. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard) (January 2006)
12. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard) (January 2006)