Safe abstractions of data encodings in formal security protocol models

Safe Abstractions of Data Encodings in Formal Security Protocol Models

Alfredo Pironti[1] and Riccardo Sisto[2]

[1]Prosecco, INRIA Paris-Rocquencourt, 23, Avenue dItalie, 75013 Paris, France, e-mail: *alfredo.pironti@inria.fr*

[2]Politecnico di Torino, Dipartimento di Automatica e Informatica, Corso Duca degli Abruzzi, 24, 10129 Torino, Italy, e-mail: *riccardo.sisto@polito.it*

Keywords Model abstraction; Refinement; Security protocols

Abstract *When using formal methods, security protocols are usually modeled at a high level of abstraction. In particular, data encoding and decoding transformations are often abstracted away. However, if no assumptions at all are made on the behavior of such transformations, they could trivially lead to security faults, for example leaking secrets or breaking freshness by collapsing nonces into constants.*

In order to address this issue, this paper formally states sufficient conditions, checkable on sequential code, such that if an abstract protocol model is secure under a Dolev-Yao adversary, then a refined model, which takes into account a wide class of possible implementations of the encoding/decoding operations, is implied to be secure too under the same adversary model. The paper also indicates possible exploitations of this result in the context of methods based on formal model extraction from implementation code and of methods based on automated code generation from formally verified models.

1 Introduction

In the last years, several techniques based on formal methods have been developed to analyze abstract models of security protocols. These models, initially introduced by Dolev and Yao [DY83], represent messages as instances of high level abstract data types. This high abstraction level makes automated verification viable, so that Dolev-Yao verification is now a well-established technique for security protocol verification, even within reach of non-experts. However, one question arises about how to ensure that the logical correctness of an abstract protocol is preserved when more concrete versions of the protocol are defined and when their implementations are developed using programming languages.

This paper focuses on one research line that consists of extending the application of the Dolev-Yao approach from very abstract models, where only the main message components and working scenarios are considered, to more detailed models capturing more closely all the real data structuring and the real program flow of protocol implementations.

Two different strategies have been proposed in order to cope with this extension: automatic code generation from abstract models, e.g. [PSD04, TH04, GHRS05], and automatic model extraction from implementation code, e.g. [BFGT06, Jür05, GLP05, BFG06].

Methods based on automatic code generation start from a high-level, formally verified, specification of the protocol, which abstracts away from many details about cryptographic and communication operations and data representations, and fill the semantic gap between formal specification and implementation, guided by implementation choices provided by the user. In [PS10], a formally sound algorithm is provided to automatically translate abstract models to source Java code. Notably, the code responsible for data transformations is not automatically generated, potentially allowing security flaws to be introduced by incorrect manual implementation of such code. Indeed, in the case study reported in [PPS12], about 30% of the code is dealing with data transformations and is manually implemented.

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. In this case, a formal soundness proof has been given for the method presented in [BFGT06]. One of the things that can be observed by looking at the results reported in [BFGT06, Jür05, BFG06], is that the part of the extracted formal model that describes data encoding and decoding operations can be quite complex, as big in size as the rest of the protocol model. This occurs even though in [BFGT06, BFG06] 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 wrong implementation of data transformations may be responsible for security faults. For this reason, it is not possible to simply neglect them when analyzing security protocols. For example, consider this very simple RPC-like protocol in the Dolev-Yao model (where perfect encryption with a private shared key also subsumes authentication), expressed abstractly in Alice and Bob notation:

$$1 : A \to B : \{n, M, REQ\}_{Kab}\text{; where } n \text{ is a nonce and } REQ \text{ a constant tag}$$
$$2 : B \to A : \{n, f(M), RES\}_{Kab}\text{; where } RES \text{ is a constant tag}$$

Assume that, before sending message 2, B emits a $begin(A, B, n)$ event, meaning that a session of the protocol was started between A and B with nonce n, and that, when receiving message 2, A first checks that the received tag is RES and the received n matches the local one, and only then emits an $end(A, B, n)$ event, meaning that a session between A and B with nonce n was correctly terminated. On this abstract model, assuming Kab is initially not known by the adversary, one can prove the injective correspondence $end(A, B, n) \Rightarrow begin(A, B, n)$, meaning that, even in the presence of a Dolev-Yao adversary, in each execution of the protocol each $end(A, B, n)$ has its own corresponding $begin(A, B, n)$.

Now consider a refined model, where each field of the encrypted content of a message is encoded before applying encryption. Suppose the correct encoding for REQ is 0 and the correct encoding for RES is 1, but the implementations of the encoding and decoding transformations used by A and B have some bugs. More precisely, suppose that, erroneously, $e_A(REQ) = 1$, where $e_A(\cdot)$ is the implementation of the encoding transformation used by A. Suppose also that B uses a different implementation having the reverse bug, i.e. $d_B(1) = REQ$, where $d_B(\cdot)$ is the implementation of the decoding transformation used by B. Because of these errors, both message 1 and 2 have the same value for the tag and the refined protocol model has a security flaw, because the adversary can play message 1 back to A, and A will accept her own message as a valid message 2, breaking the injective agreement. In conclusion, formal analysis can catch this flaw if using a detailed model, close to the real implementation, while the flaw is missed if using a more abstract model.

This kind of errors does not necessarily affect interoperability (in the previous example, A and B can run the protocol successfully despite their errors). This implies more difficulty in discovering such errors by

classical program testing.

The aim of this paper is to formally state and prove sufficient conditions under which the detailed models of data transformations, such as the ones extracted from protocol code in [BFG06], can be avoided and replaced by much simpler models or assumptions that can be checked on sequential code and in isolation (i.e. without considering the behavior of the adversary), while obtaining the same kind of security assurance on the protocol implementation.

In this work, two different kinds of data transformation functions are identified (and named in this paper as follows):

Marshaling functions Data transformation functions that operate on public data sent to or received from the network, such as the ones transforming between the internal representation of a protocol message and its external wire representation;

Encoding functions Data transformation functions that operate on possibly private data, such as a padding function applied before block-encryption, or functions creating keys from raw key material.

Encoding functions are more general than marshaling functions, because the former can operate on both private and public data, while the latter only operate on public data. So, finding sufficient conditions to safely abstract encoding functions from security protocol models would be enough to cover both classes of data transformation functions. However, exploiting the assumption that marshaling functions only operate on public data allows for weaker sufficient conditions for their abstraction. Hence, first specialized results for safe abstraction of marshaling functions are presented, which lead to weaker sufficient conditions; then generalized results for encoding functions are presented, where stronger sufficient conditions are required if private data can be accessed by the data transformation functions.

Specifically, marshaling functions operate on data that can be made available to the adversary without compromising any security property. So, in order to be able to soundly abstract these functions away, it is sufficient to assume that they can do no worse than the adversary itself can do. In practice, the results in this paper formally justify the intuition that, provided marshaling functions do not access private data, any implementation cannot harm the protocol security.

This clearly does not hold for more general encoding functions that can access private data, as showed by the example above, where a wrong encoding of the encrypted REQ tag could lead to a security flaw. For instance, injectivity is one of the sufficient conditions identified for safe abstraction of encoding functions, while this is not required for marshaling functions.

The approach presented in this paper uses Communicating Sequential Processes (CSP) [Hoa85, Ros97, Ros10] as the formal language for representing protocol models. Classically, a CSP model of a security protocol is fairly abstract: message exchanges are represented, and checks on received data are modeled by pattern matching. These are the models on which formal verification is usually performed. Normally, such models have no notion of marshaling or encoding functions, and pattern matching is not the way typical implementations of security protocols would discriminate on received data: normally a stream of bytes is received and interpreted by the implementation, before checks on the received values are performed. Hence, the formal verification results have limited scope, because they do not say anything about those neglected details.

One contribution of this paper is to find sufficient conditions such that verification of a typical abstract CSP model of a security protocol also implies correctness of a more refined CSP protocol model that takes those details into account. This saves verification of the refined – and more complex – model, which requires more verification resources or more advanced verification techniques.

So, the first problem that is addressed in this paper is to find a general way to refine a typical abstract CSP model, into a more refined CSP model that faithfully represents how encoding functions work and how messages are handled within a typical implementation.

For each class of transformations, the refinement is expressed by defining a particular structure of concurrent processes in CSP, where data transformation operations are represented by separate processes, interacting with the core processes that operate on the unencoded data. This modeling approach is quite general and close to real implementations. In addition to the process structure, only some general assumptions are introduced about the data transformation processes. Apart from these assumptions, such processes can be any process. Verifying a concrete protocol implementation using this modeling strategy can thus be reduced to verifying that the protocol implementation fulfills the assumptions made about data transformations, and verifying that the CSP refined protocol model built according to such assumptions satisfies the required security properties. This approach makes verification modular, according to an assume-guarantee style.

Even under the assumptions introduced, refined protocol models can get much more complex than their original fully abstract versions, thus making verification more challenging. To address this second problem,

we formulate sufficient conditions under which these refined models can be soundly simplified. Soundness in this case means that, provided the sufficient conditions hold, all the security faults related to a certain class of security properties are preserved when the model is simplified. This implies that it is enough to prove the absence of these faults in the simplified model to conclude that they are not present in the refined model too, under the same adversary model (Dolev-Yao).

Simplifications are divided into two steps, and the sufficient conditions for applying each step are expressed separately. The first simplification step is the main one, and already leads to a model close to the fully abstract one, while the second step completes the simplification leading to the fully abstract model (without any reference to data transformations).

The concept of fault-preserving simplifying transformations applied to security protocols is not new. It was introduced by Hui and Lowe [HL01], who identified some general classes of such transformations, and sufficient conditions to apply them. In this paper, some of the work by Hui and Lowe is adapted and exploited for our purposes. However, only the second, minor simplification step is reduced to Hui-Lowe simplifications in order to show its soundness. For the main simplification step, instead, a different proof technique is used, which also leads to more general results.

For marshaling functions, the idea is to prove the soundness of the main simplification step by considering that the more abstract model is a re-parenthesization of the CSP expression describing the refined model, where the data transformation layer is moved out of the honest agent and made part of the adversary. The sufficient conditions (data transformations should do nothing more than what the adversary itself could do) imply that this simplification step does not add any new functionality that the adversary could not previously perform. Hence the same attacks that the adversary could carry out against the protocol in the refined model can be carried out against the abstract version.

This proof strategy cannot be used with encoding functions, which access secret data, because the adversary would get to handle some protocol secret values, thus easily breaking for instance confidentiality. So, encoding functions are safely abstracted by showing that a refined CSP model, where encoding functions are separate processes, is a trace refinement of a more abstract CSP protocol that uses pattern matching to implement such encoding functions. To complete this proof, a weak simulation relation and trace refinement properties of some CSP operators are used, and some standard assumptions on pattern matching (like the aforementioned injectivity) are made explicit sufficient conditions.

The results of this work can be exploited both when using the model extraction approach and when using code generation. In the former case, the assumptions and sufficient conditions on data transformations must be checked on the (sequential) code that implements them. If the conditions hold, this code can be safely abstracted during model extraction. With code generation, if the starting point is an already verified abstract protocol model, the results given in this paper formally prove that the same security properties still hold in a refined model where code is generated so as to satisfy our assumptions and sufficient conditions. Then, such assumptions and sufficient conditions can be regarded as requirements on how the code must be generated, and the formal proofs given in this paper can be used as a basis for proving the soundness of refinement in methods based on code generation.

The remainder of the paper is organized as follows. Section 2 introduces the notation and the modeling approach, based on CSP, that is used to reason about security protocols throughout the paper. Then, following the distinction identified between marshaling and encoding functions, section 3 focuses on marshaling functions, while section 4 generalizes the results considering encoding functions. Section 5 discusses how the results can be applied when equational theories are introduced. Then, section 6 discusses experimental application of the results, using as examples the protocols for secure web services and the SSH transport protocol. Finally, section 7 concludes.

2 Abstract Protocol Models and Notation

2.1 The CSP Language

Table 1 illustrates the syntax of the CSP subset that is used in this work. Let e be an event and P, Q processes. The prefixing operator combines an event e and a process P into the process $e \rightarrow P$, which can only emit the event e and then behave like P. It is said that e is the prefix of $e \rightarrow P$. The external choice $P \Box Q$ is the process that behaves like either P or Q, with the choice between the two made by the environment. Similarly, if I is a set, the $\Box_{i \in I} P_i$ process behaves like P_i, the choice of i being made by the environment among the elements of I. In this case, i may occur in P_i and is bound in P_i. The internal choice operator $\sqcap$ is similar, but the choice is made internally, not influenced by the environment. The parallel composition $P \parallel Q$ lets P and Q execute in parallel (subject to event synchronization, described below).

Table 1: Syntax of the CSP subset used in this work.

Process	Description
$e \rightarrow P$	Prefixing
$P \Box Q$	External choice
$\Box_{i \in I} P_i$	External choice with binding
$P \sqcap Q$	Internal choice
$\sqcap_{i \in I} P_i$	Internal choice with binding
$P \parallel Q$	Parallel composition
$\parallel_{i \in I} P_i$	Parallel composition with binding
$P \setminus E$	Event hiding
$P \trianglelefteq b \trianglerighteq Q$	If/else branching
$P[[^{e'}/_{e}]]$	Renaming
$P[x/y]$	Substitution
`let` $x = expr$ `within` P	Let binding
$STOP$	Stuck process

Similarly, the $\parallel_{i \in I} P_i$ process behaves like the parallel execution of several P_i processes, one for each element of I, where in each process i is bound to a different element of I. The $P \setminus E$ process behaves like P where all the events in the set E are removed from P. The $P \trianglelefteq b \trianglerighteq Q$ process behaves like P if the boolean expression b is true, otherwise it behaves like Q. If e and e' are events, $P[[^{e'}/_{e}]]$ is the process that can emit e' whenever P can emit e. The process $P[x/y]$ is P with all occurrences of free variable x substituted by y. The process `let` $x = expr$ `within` P evaluates expression $expr$, binds variable x to its value v, and then behaves like $P[x/v]$. The $STOP$ process is the stuck process.

The alphabet of a process P, denoted by αP, is the set of events that can occur in P.

In the CSP semantics, when a process is ready to emit an event it blocks until the environment or a corresponding process can emit a matching event. When the environment or a corresponding process become ready to emit the matching event, the processes (or the process and the environment) can synchronize, and both simultaneously emit the event and evolve atomically into the next state. By default, the $P \parallel Q$ process requires that P and Q synchronize on all events in $\alpha P \cap \alpha Q$ (informally, on all events that they share), while the environment can match the remaining events. When synchronization between P and Q is desired on a different set of events E, the notation $P \parallel_E Q$ is used. Thus,

$$P \parallel Q \triangleq P \underset{\alpha P \cap \alpha Q}{\parallel} Q$$

An interesting case is $P \parallel_{\{\}} Q$, which means that P and Q run interleaved, that is in parallel without any internal synchronization. For convenience, the interleaving process is defined as

$$P \mid\mid\mid Q \triangleq P \underset{\{\}}{\parallel} Q$$

together with the interleaving with binding process $\mid\mid\mid_{i \in I} P_i$.

Events can be extended to have data after their name. For example, the events $e_1.A$ and $e_2.B.C$ are the events e_1 and e_2 with associated data values A and B, C respectively. A data type can be defined to specify the possible data values that can occur in events. The notation $e!A \rightarrow P$ means that event e can occur with value A, i.e. event $e.A$ can occur (and a corresponding matching event must occur in the environment for the process to evolve to P). The notation $e?x \rightarrow P$ means that event e can occur, and it will match any other event $e.A$, evolving to $P[A/x]$, where A can be any data value belonging to the data type. Pattern matching can be used even in more general ways. For example, $e.A?x!A$ matches events $e.A.B.A$ where B can be any value in the data type. Sometimes, when using data associated with events, an event is also called a channel.

The notation $\{|e_1, \ldots, e_n|\}$ denotes the set containing all events that match one of the e_i event forms. For example, $e.A.B \in \{|e|\}$, and $e.A.B.C \in \{|e?x.B|\}$.

A sequence of events starting with event e_1 and terminating with event e_n is denoted by $\langle e_1, \ldots, e_n \rangle$. A trace tr of a process P is a sequence of events that can occur in P. $tr \downarrow e$ denotes the number of events e occurring in trace tr.

The set of all the traces of a process P is denoted by $traces(P)$. If $traces(P) \subseteq traces(Q)$ then P is a *trace refinement* of Q, written $Q \sqsubseteq P$.

Table 2: Syntactic sugar for the proposed datatype.

Message	Representation
PAIR M N	(M, N)
SHKEY M	$M^{\sim}$
PUBKEY M	M^{+}
PRIKEY M	M^{-}
SHKEYENCRYPT M K	$\{M\}_K$
PUBKEYENCRYPT M K	$\{[M]\}_K$
PRIKEYENCRYPT M K	$[\{M\}]_K$
HASH M	$H(M)$

A trace specification $SPEC(tr)$ is a predicate whose free variable tr represents a trace. A process P satisfies a specification if the corresponding $SPEC(tr)$ predicate is true for all the traces of P:

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

2.2 Modeling Security Protocols with CSP

The datatype definitions and protocol models defined in this work are an extension of the ones used in [HL01]. Essentially, they follow the Dolev-Yao approach. It is believable that the extended datatype proposed in this paper can be enough to abstractly model the most common security protocols. Nevertheless, further extensions or modifications can be made to the datatype. The results presented here will still be valid, provided the new datatype satisfies some properties explicitly stated in this paper.

The main extension that we introduce w.r.t. [HL01] is an added support for non-atomic keys. This extension enables modeling protocols where the key is constructed from non-atomic data. The new datatype *Message* is defined recursively as

$$\begin{array}{lcl}
\mathit{ShKey} & ::= & \text{SHKEY } \mathit{Message}; \\
\mathit{PubKey} & ::= & \text{PUBKEY } \mathit{Message}; \\
\mathit{PriKey} & ::= & \text{PRIKEY } \mathit{Message}; \\
\mathit{Message} & ::= & \text{ATOM } \mathit{Atom} \mid \text{PAIR } \mathit{Message}\ \mathit{Message} \mid \\
 & & \text{HASH } \mathit{Message} \mid \mathit{ShKey} \mid \\
 & & \mathit{PubKey} \mid \mathit{PriKey} \mid \\
 & & \text{SHKEYENCRYPT } \mathit{Message}\ \mathit{ShKey} \mid \\
 & & \text{PUBKEYENCRYPT } \mathit{Message}\ \mathit{PubKey} \mid \\
 & & \text{PRIKEYENCRYPT } \mathit{Message}\ \mathit{PriKey}.
\end{array}$$

where *Atom* represents the atomic messages used as building blocks for any other message. 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*). 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 (as in [HL01]) 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 encoding parameters or encoded data, because the idea is to have a single datatype that can be used to model protocol data at different detail levels.

From now on, M, N and O range over $Message$, K over $ShKey, PubKey$ and $PriKey$, U and S over $2^{Message}$, A and B over honest protocol agents, P, Q and R over processes.

In order to get better reading for processes, the syntactic sugar reported in table 2 is also provided.

Once the datatype is defined, it is also necessary to define the adversary knowledge derivation relation $\vdash$ which models the adversary data derivation capabilities: $U \vdash M$ means that M can be derived from U. The relation $\vdash$ is defined as the smallest relation that satisfies the rules reported in table 3.

The following lemma about the relation $\vdash$ is introduced here because it will be needed in the rest of the paper:

Table 3: Rules for the knowledge derivation relation $\vdash$

Name	Definition
member:	$M \in U \Rightarrow U \vdash M$
pairing:	$U \vdash M \wedge U \vdash N \Rightarrow U \vdash (M, N)$
splitting:	$U \vdash (M, N) \Rightarrow U \vdash M \wedge U \vdash N$
key derivation:	$U \vdash M \Rightarrow U \vdash M^{\sim} \wedge U \vdash M^{+} \wedge U \vdash M^{-}$
shared key encryption:	$U \vdash M \wedge U \vdash N^{\sim} \Rightarrow U \vdash \{M\}_{N^{\sim}}$
public key encryption:	$U \vdash M \wedge U \vdash N^{+} \Rightarrow U \vdash \{[M]\}_{N^{+}}$
private key encryption:	$U \vdash M \wedge U \vdash N^{-} \Rightarrow U \vdash [\{M\}]_{N^{-}}$
shared key decryption:	$U \vdash \{M\}_{N^{\sim}} \wedge U \vdash N^{\sim} \Rightarrow U \vdash M$
public key decryption:	$U \vdash \{[M]\}_{N^{+}} \wedge U \vdash N^{-} \Rightarrow U \vdash M$
private key decryption:	$U \vdash [\{M\}]_{N^{-}} \wedge U \vdash N^{+} \Rightarrow U \vdash M$
hashing:	$U \vdash M \Rightarrow U \vdash H(M)$

Lemma 2.1

$$U \vdash M \wedge U \subseteq U' \Rightarrow U' \vdash M, \tag{1}$$
$$U \vdash M \wedge U \cup \{M\} \vdash M' \Rightarrow U \vdash M'. \tag{2}$$

This lemma has been proved in [HL01] for a similarly defined relation based on the datatype defined there, and can be proved to hold for the definition of $\vdash$ in table 3, by structural induction. In order to apply the results given in this paper to an extension of this datatype, it is necessary to ensure that lemma 2.1 holds for the extended datatype.

Honest agents and the adversary are defined as in [HL01]. 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 adversary, then the adversary should not learn M;

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

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

The *send* and *receive* events can also be interpreted as channels, used by the 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 adversary acts as the medium, thus being allowed to see, modify, forge or drop any message. It uses the knowledge derivation relation $\vdash$ to forge new messages from the previously learnt messages. The set of messages the adversary can derive from a knowledge S is defined as

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

Finally, the formal definition of the adversary is

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

where S is the current adversary knowledge, and *leak.M* is the event that signals that the adversary can derive M from its current knowledge. The set of all agents is defined as $Agent = Honest \cup \{ADV\}$.

The model representing all the honest agents and the adversary is called SYS, and is formally defined as

$$SYS \triangleq (|||_{A \in Honest} P_A) \parallel ADV(AK_0)$$

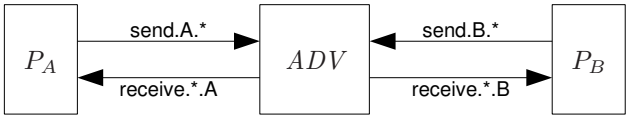

Figure 1: Actors A and B with ADV in SYS.

where, for each $A \in Honest$, P_A is the CSP process that describes A's behavior, and AK_0 is the initial adversary knowledge. By definition, in SYS the adversary and the honest agents synchronize on the *send* and *receive* events. For actors A and B, the SYS process can be depicted as in figure 1.

In this model decryption operations are represented by the CSP pattern matching feature on *receive* channels. For example, $receive?\{x\}_{N^\sim} \rightarrow P$ can receive an encrypted message, decrypt it by using key $N^\sim$ and bind x to the obtained plaintext.

In this work, it is assumed that any trace property $SPEC(tr)$ is such that its truth does not depend on the *send* and *receive* events that may appear in trace tr. Equivalently, process P satisfies a specification $SPEC$, iff the process $P \setminus \{|send, receive|\}$, where *send* and *receive* events are hidden, satisfies the same specification $SPEC$. Formally

$$P \textbf{ sat } SPEC \Leftrightarrow P \setminus \{|send, receive|\} \textbf{ sat } SPEC \tag{3}$$

Assumption (3) is reasonable, because security properties are normally 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.

Two predicates, namely secrecy and injective authentication (or simply authentication), define the two most common properties.

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

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

In order to define authentication, a formal definition of $AgreementSet$ is first needed.

$$\begin{aligned} M \in AgreementSet \Leftrightarrow \exists\ & tr \in traces(P); A \in Agents; B \in Honest \cdot \\ & tr \downarrow finished.A.B.M > 0 \vee tr \downarrow running.B.A.M > 0 \end{aligned} \tag{4}$$

Informally, $AgreementSet$ is the set of all the possible messages upon which the agents should agree (e.g. if the agents should agree on a key and an atom, the $AgreementSet$ includes pairs with the first item that is a key and the second one that is an atom).

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 message $M \in AgreementSet$:

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

Weaker types of authentication have also been defined, for instance non injective authentication, where there is no one-to-one correspondence between the runs of actors A and B, or weak authentication, where there is no agreement on session data; they are described, for example, in [Low97]. It is believable that the results proved in this paper for injective authentication hold for weaker forms of authentication too.

3 Handling the Marshaling 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 representations. For example, one such function could transform a variable-length sequence of items, stored internally in some way, into an external standard representation made up of an integer (the length of the sequence) followed by the encoding of each item of the sequence, with separators between items. The inverse function would do the inverse transformation.

This section focuses on data transformation functions that operate on messages sent to or received from the network, which are called marshaling functions in this paper, after the "marshaling" term that is normally used to denote the operation of encoding data objects for transmission over a communication channel.

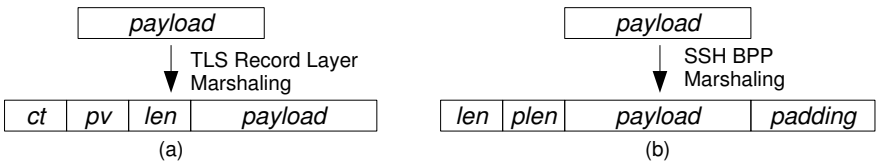

Figure 2: Example of marshaling functions in the (a) TLS and (b) SSH protocols.

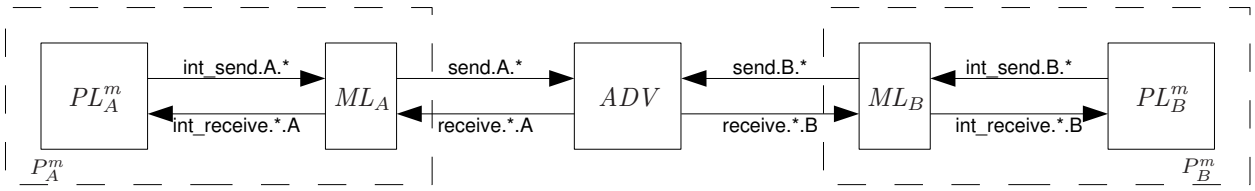

Figure 3: Actors A and B with ADV in SYS^m.

Simple examples of such marshaling functions can be found in the popular TLS and SSH protocols. In TLS, the Record Layer Protocol is responsible for marshaling the encrypted *payload* of a "record" (a packet in TLS terminology) by prefixing it as depicted in figure 2(a): the content type *ct* indicates the kind of data contained in the payload, e.g. if the data are TLS handshake messages, or user application data; the protocol version *pv* indicates which TLS protocol version is in use; and the length *len* indicates the length of the upcoming *payload*. The marshaled packet is then the concatenation of $(ct, pv, len, payload)$.

In SSH, during the handshake phase, when encryption is not in place, the Binary Packet Protocol (BPP) is responsible for marshaling the SSH packets containing handshake messages according to figure 2(b). Given a handshake message *payload*, the BPP adds a header containing the payload length *len*, followed by the padding length *plen*. Then the plaintext *payload* is concatenated and finally the *padding* is appended.

In this work it is assumed, and thus modeled accordingly, that, as usual, the marshaling layer is implemented separately from the protocol logic layer.

In SYS the actors exchange the abstract representation of data with the adversary. In order to model the marshaling layer, a refined model SYS^m (where m stands for marshaling) is defined as depicted in figure 3 for actors A and B.

Basically, SYS^m acts like SYS, but it is explicitly modeled that the *external* representation of data is being exchanged over *send* and *receive*. Hence, even if the same names (*send* and *receive*) are used for channels in SYS and in SYS^m, the data associated with these channels are different in the two models. More precisely, the model of each honest agent A is refined into P_A^m, which is composed of two coupled processes PL_A^m and ML_A, representing the protocol logic and the marshaling layer of a program respectively. Each PL_A^m in SYS^m acts *like* its corresponding P_A in SYS, but it is explicitly modeled that it sends its *internal* representation to its coupled marshaling layer ML_A, which in turn sends the *external* representation to the adversary, and vice versa.

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

$$SYS^m \triangleq ((((|||_{A \in Honest} PL_A^m) \parallel (|||_{A \in Honest} ML_A)) \setminus \{|int\_send, int\_receive|\}) \parallel ADV(AK_0^m)$$

It could be argued that, potentially, this model allows each honest agent to send messages to any marshaling layer, and vice versa. However, the implementations of protocol logic and its coupled marshaling layer are very often part of the same application, so errors that would lead honest agents or marshaling layers to communicate with the wrong process are not realistic. For this reason, it is assumed that PL_A^m and ML_A emit events in the set $\{|int\_send.A, int\_receive?B.A|\}$. This means that PL_A^m only exchanges messages with its coupled marshaling layer model ML_A, and vice versa. Indeed, this assumption implies that such errors cannot happen in the model too. For the same reason, it is reasonable to hide the program internal communication channels *int_send* and *int_receive* from the adversary's view.

The initial adversary knowledge AK_0^m in SYS^m has some relation with AK_0 in SYS, however this relation now is irrelevant, and can be explained later.

Finally, the relation between each P_A and the corresponding PL_A^m and the formal definition of each ML_A are given. For each P_A, PL_A^m can be built by refining P_A so as to model the information that the protocol agent must provide to the marshaling layer for its proper working. More precisely, PL_A^m is obtained from P_A by replacing in P_A each event taking the form $send.A.B.M$ with $int\_send.A.B.(\textsc{Atom}\ \mathcal{L}, (a, M))$, and each event taking the form $receive.B.A.M$ with $int\_receive.B.A.(\textsc{Atom}\ \mathcal{L}, (a, M))$. Here, $\textsc{Atom}\ \mathcal{L}$ is a special

$$
\begin{array}{l}
ML_A \triangleq \\
\Box^{\Box_{a \in Marshaling, M \in Message}\, int\_send!A?B!(\text{ATOM } \mathcal{L}, (a, M)) \rightarrow send!A!B!e_A(a, M) \rightarrow ML_A} \\
\quad \Box_{y \in Message}\, receive?B!A!y \rightarrow \Box_{a \in Marshaling}\, int\_receive!B!A!(\text{ATOM } \mathcal{L}, (a, d_A(a, y))) \rightarrow ML_A
\end{array}
$$

Figure 4: Formal definition of the ML_A process.

atom not present in the definition of P_A, whose only purpose is to tag the messages added by the refinement to PL_A^m, and a is such that $a \in Marshaling \subseteq Message$ where $Marshaling$ is the set of messages that can be used as marshaling parameters, i.e. additional information needed by the marshaling layer for its proper operation (e.g., an element of $Marshaling$ may include the name of the marshaling algorithm to be applied and any related parameters, such as length of paddings etc.). Throughout the rest of the paper, a, b, c and d range over $Marshaling$.

The elements of $Marshaling$ that are used in refining the protocol actor model may already appear in the abstract version P_A as well. For example, if the marshaling parameters a are being negotiated within the protocol logic, then a will be present in P_A. So, the message (a, M) may be in P_A as well. When refining P_A into PL_A^m, new messages of the form (a, M) are introduced, wherever message M is output in P_A. Now, to simplify back PL_A^m to P_A (done formally in section 3.1.2), only the terms of the form (a, M) that have been added during refinement should be abstracted to M, while those (a, M) terms originally present in P_A should be left untouched. The ATOM $\mathcal{L}$ marker (not present in P_A by definition) is introduced to syntactically distinguish the (a, M) terms already present in P_A, from the (ATOM $\mathcal{L}$, (a, M)) terms added by refinement, so that only the latter will be the target of abstraction. Since ATOM $\mathcal{L}$ is just a syntactic marker, it is assumed that neither P_A nor PL_A^m 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. Similarly, it is assumed that ATOM $\mathcal{L} \notin AK_0^m$: this assumption makes some proofs simpler, while not reducing the power of the adversary, since the protocol logic behavior is assumed to never depend on ATOM $\mathcal{L}$ anyway.

Each process ML_A models the behavior of the marshaling layer. Because of this, it can perform two kinds of actions:

- receive from its coupled process PL_A^m internal representations of data, along with marshaling parameters, and send marshaled data to the ADV process;
- receive marshaled data from the ADV process, and send to its coupled process PL_A^m the internal representation, obtained using the unmarshaling parameters specified by PL_A^m.

Apart from these assumptions about the possible interactions of ML_A, it is assumed that internally ML_A can behave in any way, thus even including erroneous implementations of data transformations. The only restriction is that ML_A can access only the data explicitly provided from outside. This lets us see ML_A as part of the adversary, which is the intuition that will be used later on to abstract ML_A from the model.

The behavior of ML_A can be represented by the CSP process in figure 4, where $e_A(a, M)$ and $d_A(a, y)$ represent the result of the encoding and decoding operations implemented in actor A respectively. Each one of them can be any message that can be derived from a and M or a and y respectively; formally:

$$
e_A(a, M) \in deds(\{a, M\}) \ \wedge \ d_A(a, y) \in deds(\{a, y\}) \tag{5}
$$

By this definition, the properties of the marshaling layer model ML_A can be stated. The result $e_A(a, M)$ of encoding M with parameters a can be anything that can be derived from M and a, thus accounting for arbitrary complex encoding schemes. 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 lose some information on M, but can only use information that comes from the internal representation and from the marshaling parameters. In order to model some information that is hard-coded into the marshaling function implementation, that information needs to be added to a explicitly. For example, the marshaling parameters used for a message of the TLS protocol would look like $a = (rlp, ct, pv)$, where rlp is the name of the record layer protocol encoding algorithm, and ct and pv are the content type and version. In this case, $e_A(a, M)$ would be such that $e_A((rlp, ct, v3.0), M) = (ct, v3.0, lenght(M), M)$.

The same reasoning applies to the result $d_A(a, y)$ of decoding y with parameters a, but the case when y is not recognized as a valid encoding for parameters a must be taken into account as well. In the latter case, it is assumed that $d_A(a, y) = \text{ATOM } \mathcal{E}$, where ATOM $\mathcal{E}$ is a special atom that represents a decoding error code. Since this error code is part of the decoding function, it is assumed that ATOM $\mathcal{E} \in deds(a)$ for any $a \in Marshaling$. Moreover, to ensure the adversary capabilities are not restricted, it is assumed that for all adversary knowledges AK, ATOM $\mathcal{E} \in AK$. For example, a decoding function for the TLS record layer protocol that just accepts version $v3.0$ could be defined as:

$$\begin{array}{lcl} d_A((rlp, ct, v3.0), (ct, v3.0, lenght(M), M)) & = & M \\ d_A(a, M) & = & \text{ATOM } \mathcal{E} \end{array}$$

where as usual the second case is taken if the arguments do not match the first case.

In the marshaling layer model analyzed here, it is modeled that decoding error conditions are reported to the protocol logic, through the use of the special ATOM $\mathcal{E}$ error code. A marshaling layer model that gets stuck when a decoding operation fails, so that the protocol logic is never delivered the special ATOM $\mathcal{E}$, is possible, and is actually a refinement of the model analyzed here. So the results obtained in this paper for the marshaling layer model that reports the errors to the protocol logic, are also valid for the refined model of the marshaling layer that immediately stops in case of error and does not require the protocol logic to handle error conditions.

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 for simplicity, but this model could be extended to include them.

All the properties of the modeled marshaling layer, namely that the only data accessed by the marshaling/unmarshaling 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 analysis techniques for sequential code.

3.1 Model Simplifications

The aim of this subsection is to prove that under some light conditions the simplification of the refined model into the abstract one does not lose security faults. This result justifies the possibility to formally claim properties on the refined model by performing verification on the (simpler) abstract one. The simplification can be divided into two subsequent steps as depicted in figure 5. The first one removes the marshaling layer while the second one completes the transformation into the abstract system by removing the encoding parameters from the protocol logic.

3.1.1 Removing the marshaling layer

SYS^m can be simplified by removing processes ML_A and turning processes PL_A^m into their renamed versions $PL_A^m[[^{send}/_{int\_send}]][[^{receive}/_{int\_receive}]]$, so as to connect them directly to the adversary through the right channels *send*, and *receive*.

With abuse of notation, the PL_A^m symbol will be used from now on to refer to both the process communicating on *int_send*, *int_receive*, and the one communicating on *send* and *receive*. In fact, these two processes are the same, up to a renaming of communication channels. Keeping them under the same symbol actually helps in the explanation of the results, still allowing to produce rigorous proofs.

Along with the above transformation, the initial adversary knowledge is also changed from AK_0^m into

$$AK_0^{m\text{-}noML} \triangleq AK_0^m \cup Marshaling \cup \{\text{ATOM } \mathcal{L}\} \tag{6}$$

This is done in order to make sure that in the simplified model the adversary knows the encoding parameters and the special marker ATOM $\mathcal{L}$. Actually this is the sufficient condition introduced to guarantee the soundness of the simplification.

The final result of the transformation is a new process $SYS^{m\text{-}noML}$ defined as

$$SYS^{m\text{-}noML} \triangleq (|||_{A \in Honest} PL_A^m) \parallel ADV(AK_0^{m\text{-}noML})$$

whose graphical representation is given in figure 5.

This simplified model is very close to the abstract model SYS, the only difference being that when in SYS a $send.A.B.M$ or a $receive.B.A.M$ occurs, in $SYS^{m\text{-}noML}$ a $send.A.B.(\text{ATOM } \mathcal{L}, (a, M))$ or a $receive.B.A.(\text{ATOM } \mathcal{L}, (a, M))$ occurs. In other words, $SYS^{m\text{-}noML}$ is like SYS, but with each sent or received message tagged by its intended marshaling parameters and by ATOM $\mathcal{L}$.

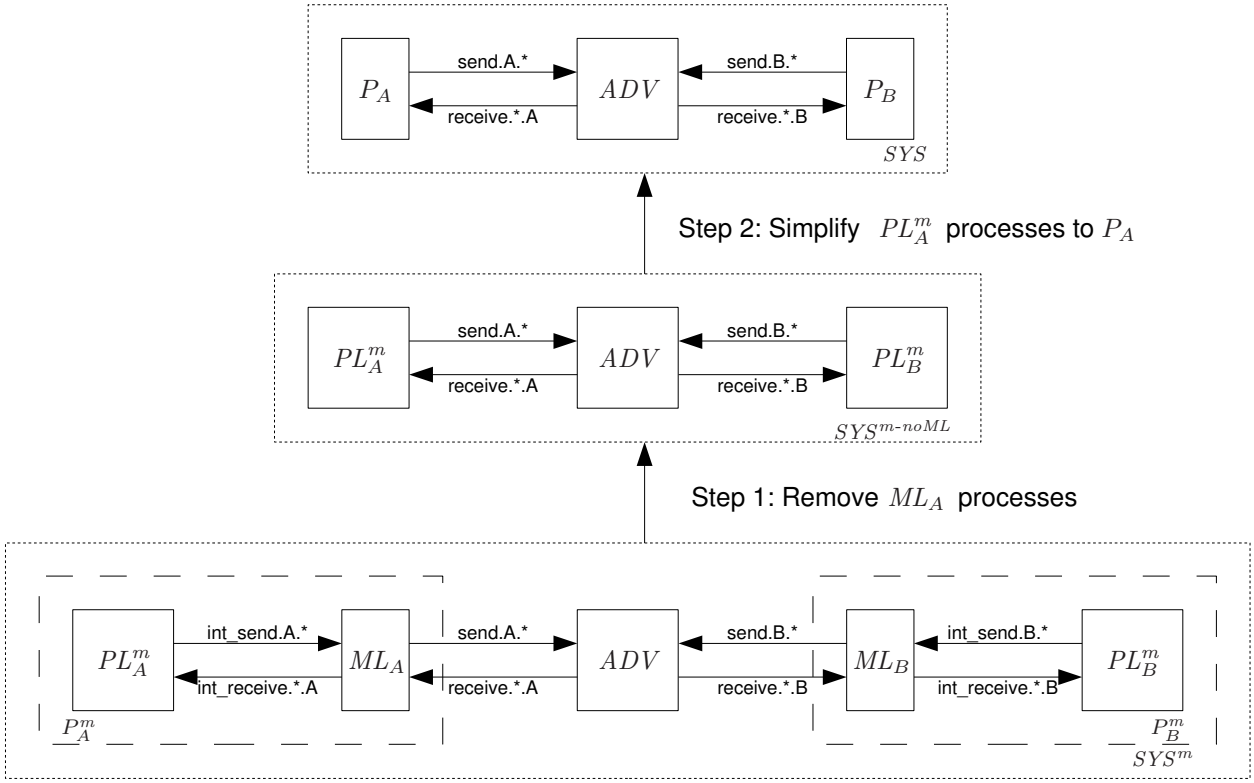

Figure 5: The simplification steps from SYS^m to SYS, using the intermediate $SYS^{m\text{-}noML}$.

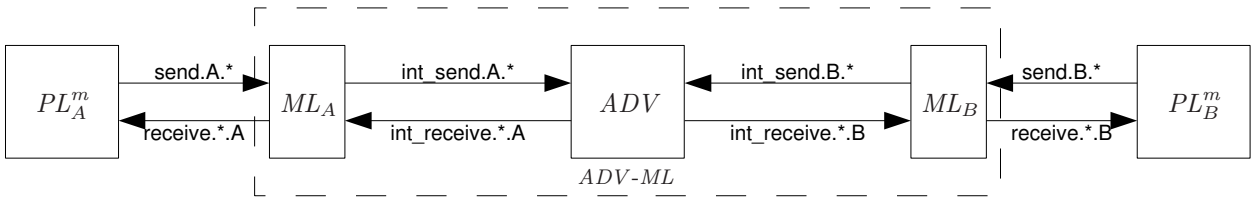

Figure 6: Actors A and B with $ADV\text{-}ML$ in $SYS^{m\text{-}advML}$.

Another possible intuition to interpret $SYS^{m\text{-}noML}$ is that it can be seen as an abstraction of SYS^m, where the ML_A processes have been embedded into the adversary (and the communication channels renamed). This intuition is the basis for the proof justifying the first refinement step.

Preservation of security properties in this transformation is expressed by:

Theorem 3.1

$$\forall SPEC \cdot SYS^{m\text{-}noML} \textbf{ sat } SPEC \Rightarrow SYS^m \textbf{ sat } SPEC$$

That is, all security properties defined on traces that are satisfied by $SYS^{m\text{-}noML}$, are satisfied by SYS^m too. This means that if one has a protocol model like SYS^m, including a marshaling layer complex at will, one can verify any security property defined on traces on the simpler $SYS^{m\text{-}noML}$, where there is no marshaling layer. By theorem 3.1, if the property is verified on $SYS^{m\text{-}noML}$ then it can be concluded that the property holds on SYS^m too.

The proof of theorem 3.1 is given in appendix A.1; here only a sketch is provided.

[Proof sketch of theorem 3.1.] Let us change SYS^m by swapping channels $int\_send, int\_receive$ with channels $send, receive$, and by re-parenthesizing processes as showed in Figure 6, so as to make the marshaling layer become part of the adversary.

In the resulting $SYS^{m\text{-}advML}$, the adversary (in this sketch denoted by $ADV\text{-}ML$) is the parallel composition of ADV (the adversary of SYS) with ML_A.

To keep the proof sketch simple, we are using the same process name for ML_A and ADV, even when they communicate on swapped channels. (However, the proof given in appendix A.1 makes this distinction explicit to avoid ambiguities.)

Any fault trace of SYS^m is also a fault trace of $SYS^{m\text{-}advML}$, because only an injective renaming of public and private channels relates the two models. The re-parenthesization is irrelevant because of the associative property of parallel composition. So, any trace property that holds on the intermediate $SYS^{m\text{-}advML}$ is also implied to hold on SYS^m.

Now, the goal is to prove that the intermediate $SYS^{m\text{-}advML}$ is a trace refinement of $SYS^{m\text{-}noML}$. This implies that any trace property that holds on $SYS^{m\text{-}noML}$ also holds on $SYS^{m\text{-}advML}$, thus closing the chain between $SYS^{m\text{-}noML}$ and SYS^m and the proof of the theorem.

$SYS^{m\text{-}advML}$ and $SYS^{m\text{-}noML}$ only differ by the definition of the adversary, $ADV\text{-}ML$ and ADV respectively. So, the proof reduces to prove a trace refinement between the latter pair of processes. This part of the proof is divided into two steps. Step (i): define the knowledge MAK of $ADV\text{-}ML$ as the adversary knowledge, plus the messages possibly being stored by ML_A, when the latter received a message over a public (private) channel, but not yet delivered its encoded (decoded) form onto the private (public) channel. Then, show that MAK does not change when internal events happen inside $ADV\text{-}ML$ (namely, private communication between ML_A and the adversary).

Step (ii): show, by induction over the length of a trace tr, that for any trace tr of $ADV\text{-}ML$ that leads to some knowledge MAK', the same trace tr can lead ADV to a state with some AK', such that $deds(MAK') \subseteq deds(AK')$.

This proves that $ADV\text{-}ML$ is no more powerful than ADV is, and as a corollary leads to the required proof item showing that $SYS^{m\text{-}advML}$ is a trace refinement of $SYS^{m\text{-}noML}$.

□

Theorem 3.1 states that in a protocol specification where the marshaling layer is modeled as previously described, only the protocol logic represented by PL_A^m is responsible for the security properties of the whole protocol, while any possible implementation of the marshaling layer ML_A of arbitrary complexity can be considered as part of the adversary, provided that the latter knows all required marshaling schemes and parameters (because $Marshaling \subset AK_0^{m\text{-}noML}$). One further condition required by theorem 3.1 is that the adversary 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.

No assumption on the invertibility of encoding functions has been made, thus even erroneous specifications of encoding schemes are safe (though probably not functioning). For instance, an erroneous specification that requires to collapse all nonces into a constant cannot be responsible for replay attacks, since it is protocol logic duty to check that the internal representation of the locally generated nonce is equal to the internal representation of the received unmarshaled nonce. 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.

3.1.2 Removing the marshaling parameters

This second step simplifies each PL_A^m back to P_A, thus simplifying $SYS^{m\text{-}noML}$ to the fully abstract SYS.

For the main simplification step from SYS^m into $SYS^{m\text{-}noML}$, presented in the previous section, it was possible to prove a preservation theorem that applies to any security property that can be defined on traces (theorem 3.1). For this second minor simplification step, instead, the proof technique exploits the Hui and Lowe's theory of fault-preserving simplifying transformations [HL01], which lets us prove preservation on specific security properties rather than on any security property that can be defined on traces. Then, in this paper the preservation theorem for the second step applies only to the main security properties, i.e. secrecy and authentication, leaving extensions to other security properties as future work.

The preservation theorem for the second step can be formulated as follows.

Theorem 3.2 *If* $AK_0 = AK_0^{m\text{-}noML}$

$$SYS \textbf{ sat } Secrecy \quad \Rightarrow \quad SYS^{m\text{-}noML} \textbf{ sat } Secrecy \tag{7}$$

$$SYS \textbf{ sat } Agreement_{AgreementSet} \quad \Rightarrow \quad SYS^{m\text{-}noML} \textbf{ sat } Agreement_{AgreementSet} \tag{8}$$

In practice, it means that if secrecy and authentication have been verified on SYS then the same properties hold on $SYS^{m\text{-}noML}$ too, independently of encoding parameters. Putting together this theorem and theorem 3.1 we get finally that if secrecy and authentication have been verified on SYS then the same properties hold on SYS^m too, provided the adversary knowledge in SYS includes encoding parameters and Atom $\mathcal{L}$. However, the verification of security properties other than secrecy and authentication does not benefit of theorem 3.2, but only of theorem 3.1, i.e. these properties can be safely verified on $SYS^{m\text{-}noML}$.

Before giving the proof of theorem 3.2, let us recall the part of the theory of fault-preserving simplifying transformations [HL01] that will be used.

In its simplest form, a fault-preserving simplifying transformation is a renaming transformation, i.e. 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, in such a way that $f(e)$ is e with any message M occurring in e replaced by $f(M)$, $f(tr)$ is tr with any event e occurring in tr replaced by $f(e)$ and $f(P)$ is a CSP process having as traces the ones that result from the application of f to the traces of P, i.e. such that $traces(f(P)) = f(traces(P))$.

Let

$$SYS^{Ref} = (|||_{A \in Honest} P_A^{Ref}) \parallel ADV(AK_0^{Ref})$$

be a refined protocol model with associated initial adversary knowledge AK_0^{Ref}, and

$$SYS^{Abs} = (|||_{A \in Honest} f(P_A^{Abs})) \parallel ADV(AK_0^{Abs})$$

one of its abstractions with associated initial adversary knowledge AK_0^{Abs}. As proved in [HL01], if $f(\cdot)$ is a renaming transformation that satisfies conditions

$$U \cup AK_0^{Ref} \vdash M \Rightarrow f(U) \cup AK_0^{Abs} \vdash f(M) \tag{9}$$

$$f(AK_0^{Ref}) \subseteq AK_0^{Abs} \tag{10}$$

then

$$SYS^{Abs} \textbf{ sat } Secrecy \Rightarrow SYS^{Ref} \textbf{ sat } Secrecy.$$

For the preservation of authentication Hui and Lowe add another condition to the previous ones. In this work, the following slightly different additional condition is used that is weaker than the one given in [HL01]:

$$\forall\, M, M' \in AgreementSet \cdot\ M \neq M' \Rightarrow f(M) \neq f(M') \tag{11}$$

That is, $f(\cdot)$ must be *locally* injective on $AgreementSet$, and not on the whole $Message$ set, as in [HL01]. The result about authentication is that if equations (9), (10), and (11) are satisfied, then

$$SYS^{Abs} \textbf{ sat } Agreement_{f(AgreementSet)} \Rightarrow SYS^{Ref} \textbf{ sat } Agreement_{AgreementSet}$$

The proofs of this and other extensions can be found in [Pir10].

The reason why the preservation results obtained by the theory of Hui and Lowe are property-dependent is that the renaming transformations introduced in [HL01] can in principle alter any event in the trace, even the special ones used to define security properties. So, for each renaming transformation and each security property, it is necessary to prove that, although the special events are modified, faults in the refined system are preserved as faults in the abstract one (hence the fault-preserving name) by appealing to additional conditions that are property-dependent. In the transformation from SYS^m to $SYS^{m\text{-}noML}$, instead, for any trace of the refined SYS^m, it is possible to find a corresponding trace in $SYS^{m\text{-}noML}$ where any event that is neither *send* nor *receive* is the same. This is why in that case a property-independent result could be obtained.

The proof of theorem 3.2 uses one of the fault-preserving renaming transformations that were studied in [HL01]. This transformation collapses each pair (M, M') belonging to a given set $Pairs$ into its first item M. The formal definition of this function, here named f, is:

$$\begin{array}{rcll}
f(\textsc{Atom}\ A) & = & \textsc{Atom}\ A, & \\
f((M, M')) & = & \begin{cases} f(M), & \text{if } (M, M') \in Pairs \wedge \neg isPair(M'), \\ (f(M), f(M')) & \text{if } (M, M') \notin Pairs \wedge \neg isPair(M'), \end{cases} & \\
f((M, (M', M''))) & = & \begin{cases} f((M, M'')) & \text{if } (M, M') \in Pairs, \\ (f(M), f((M', M''))) & \text{otherwise}, \end{cases} & \\
f(\{M\}_K) & = & \{f(M)\}_{f(K)} & \\
f(\{[M]\}_K) & = & \{[f(M)]\}_{f(K)} & \\
f([\{M\}]_K) & = & [\{f(M)\}]_{f(K)} & \\
f(H(M)) & = & H(f(M)) & \\
f(K^*) & = & f(K)^* &
\end{array}$$

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

As showed in [HL01], one way of having conditions (9) and (10) satisfied with this transformation is to ensure that the knowledge of the adversary in SYS^{Abs} includes the transformed versions of the data known by the adversary in SYS^{Ref} plus the transformed versions of all the removed messages, i.e.

$$AK_0^{Abs} \supseteq f(AK_0^{Ref}) \cup \{f(M') | (M, M') \in Pairs\} \tag{12}$$

The proof that if this condition is satisfied then (9) and (10) are satisfied too for our modified datatype is very similar to the one given in [HL01], by induction on the relation $\vdash$. Condition (12) is needed in order to prove condition (9), because otherwise some source of information for the adversary could be removed by the transformation. In particular, suppose the $\{f(M')|(M,M') \in Pairs\}$ terms were not included in the right hand side of (12). In this case we would have that, for any $U = \{(M,M')\}$ with $(M,M') \in Pairs$, the condition $U \vdash M'$ would be true while $f(U) \vdash f(M')$ would not necessarily be true, because $f(U) = f(\{(M,M')\}) = \{M\}$, thus possibly invalidating condition (9). Hence the definition of condition (12).

In order to preserve agreement, $f(\cdot)$ must also satisfy condition (11). In order to achieve this, only one additional constraint is required:

$$\forall\ M \in AgreementSet; subM \in subterms(M)\ \cdot\ isPair(subM) \Rightarrow subM \notin Pairs \tag{13}$$

where $subterms(M)$ is the set containing M and all its subterms. Constraint (13) means that no subterm of any $M \in AgreementSet$ 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 collapsed.

Using these results about function $f(\cdot)$, it is possible now to prove theorem 3.2.

[Proof of theorem 3.2.] The proof is based on the fact that the function $f(\cdot)$ collapsing pairs can be safely used to transform PL_A^m into P_A, and thus $SYS^{m\text{-}noML}$ into SYS. Indeed, PL_A^m 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 take PL_A^m back to P_A:

1. $P_A^{tmp} = f(PL_A^m)$, with $Pairs = \{(\text{ATOM}\ \mathcal{L}, a) \mid a \in Marshaling\}$
2. $P_A = f^{sym}(P_A^{tmp})$, with $Pairs = \{(\text{ATOM}\ \mathcal{L}, M) \mid M \in Message\}$

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 marshaling parameters that have been added to represent the marshaling layer. Then, step 2 removes the syntactic marker, finally obtaining P_A.

Exploiting the results proved in [HL01], each one of these transformations preserves secrecy and authentication if the required sufficient conditions (12) and (13) hold.

In step 1, the preservation results apply by assigning

$$P_A^{Abs} = P_A^{tmp}; \quad P_A^{Ref} = PL_A^m; \quad AK_0^{Ref} = AK_0^{m\text{-}noML}$$

By setting $AK_0^{Abs} = AK_0^{m\text{-}noML}$ condition (12) becomes

$$AK_0^{m\text{-}noML} \supseteq f(AK_0^{m\text{-}noML}) \cup \{f(M')|(M,M') \in Pairs\} \tag{14}$$

It is now showed that (14) is satisfied. By the definition (6) of $AK_0^{m\text{-}noML}$, and since ATOM $\mathcal{L} \notin AK_0^m$ and ATOM $\mathcal{L} \notin Marshaling$, it follows

$$\begin{aligned} f(AK_0^{m\text{-}noML}) &= f(AK_0^m) \cup f(Marshaling) \cup f(\{\text{ATOM}\ \mathcal{L}\}) \\ &= AK_0^m \cup Marshaling \cup \{\text{ATOM}\ \mathcal{L}\} \\ &= AK_0^{m\text{-}noML} \end{aligned}$$

Moreover, $\{f(M')|(M,M') \in Pairs\} = Marshaling$, because the only elements collapsed by $f(\cdot)$ in step 1 are marshaling parameters. So, (14) is finally reduced to $AK_0^{m\text{-}noML} \supseteq AK_0^{m\text{-}noML}$, which is trivially true.

Condition (13) holds too 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.

In step 2, the preservation results apply with

$$P_A^{Abs} = P_A; \quad P_A^{Ref} = P_A^{tmp}; \quad AK_0^{Ref} = AK_0^{m\text{-}noML}; \quad AK_0^{Abs} = AK_0$$

Condition (12) is clearly satisfied because of the theorem hypothesis $AK_0 = AK_0^{m\text{-}noML}$, and with a reasoning similar to the one done in step 1. Also condition (13) holds because of the same reasoning used for step 1.

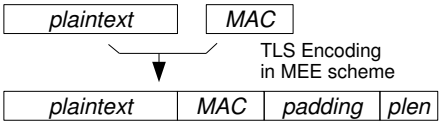

Figure 7: Example of encoding function in the TLS protocol.

Summing up, since each transformation consists of applying the renaming function $f(\cdot)$ with conditions (12) and (13) satisfied, we can conclude that both transformations preserve secrecy and authentication, which proves the theorem.

□

Since (9) depends on the derivation relation $\vdash$, this condition must be checked each time the datatype is updated.

4 Handling the Encoding of Data to be Ciphered and Key Material

Cryptographic protocols must define, for interoperability, not only the encoding of messages that are sent and received on communication channels, but also the encoding of data on which cryptographic operations are applied. For instance, when block ciphers are in use, the plaintext must normally be padded before encryption, so that the length of the padded plaintext is a multiple of the encryption block size.

For example, the TLS protocol uses a MAC-then-Encode-then-Encrypt (MEE) scheme whose encode step is depicted in figure 7. The encoding function concatenates the *plaintext* of a packet with the MAC computed in the first step of the MEE scheme. Then some *padding* is added to ensure the final length is block aligned with the encryption scheme block size, and finally the padding length *plen* is appended. The resulting byte array is finally encrypted, performing the last step of the MEE scheme. The message is sent after applying the marshaling transformation showed in figure 2(a). At the receiver side, the unmarshaling is first performed, which returns the encrypted byte array of the message. After decryption, the resulting byte array is decoded by first computing the plaintext length (using *plen* and the fixed size of MAC) and then extracting the various components from the byte array.

In SSH, when encryption is in place, the BPP operations described in figure 2(b) become the encoding operations performed before encryption, and the marshaling layer becomes transparent. That is, the result of the BPP encoding is first encrypted, and then directly sent over the network (concatenated with a MAC of constant size). Interestingly, when some data are received from the network, the marshaling layer just passes the byte stream as is to the protocol layer, which decrypts the received ciphertext unconditionally. Then, as soon as the first 4 bytes are decrypted, it is the decoding function responsibility to check the value *len* of the length field, to decide on how to process the subsequent data. This design was meant to hide the length of the exchanged data by keeping it secret, in order to avoid attacks based on the knowledge of the length of the exchanged messages. However, checking the length field from decrypted data that is not yet authenticated by a check of the MAC leads to a confidentiality flaw [APW09], if the adversary alters the first block of an encrypted message. Unfortunately, in the standard Dolev-Yao model of perfect encryption, where messages are values of abstract types and the adversary can only encrypt/decrypt messages if it has the correct key, this kind of attacks cannot be caught because encryption is implicitly authentic if the key stays secret.

The modeling of marshaling functions introduced in the previous section lets one soundly abstract the marshaling layer under very few sufficient conditions (the adversary must know the encoding parameters). Unfortunately, that layered model only applies to the marshaling layer, which operates on already protected data, and not on encoding operations performed on data to be further processed by cryptographic operations. The main difference between the two cases is that the data on which the marshaling layer operates could be accessed by the adversary in their unmarshaled form without compromising the protocol security anyway, while the other encoding functions operate on possibly confidential data.

For this reason, in this section a more general model of encoding functions is introduced. This more general model can handle encoding functions operating on confidential data, as well as marshaling functions. However, when using this general model, the sufficient conditions for the sound abstraction of the encoding functions need to be much stronger. For this reason, it is convenient to apply the results found in this section using the more general model only to encoding functions that operate on possibly confidential data, where such generality is required, and one has to be satisfied with stronger sufficient conditions. For marshaling

functions, instead, it is convenient to use the specialized model of section 3, because it requires weaker sufficient conditions.

Accordingly, and for simplicity, when presenting the general model developed in this section, reference will be made only to the encoding functions applied to data to be fed to cryptographic functions. As showed in section 6, the general and specialized models can be combined together to handle both marshaling, and encoding of possibly confidential data, in the most convenient way.

The refinement of the abstract protocol model into the general model that includes data encoding can be seen as divided into two steps. In the first step, encoding operations are introduced, but using pattern matching for the corresponding decoding operations. In the second step, a more realistic model of a real implementation is finally considered, where decoding operations occur explicitly.

4.1 The First Refinement Step

The starting point is the abstract protocol model SYS, where the encoding of data is completely abstracted away. The result of the first refinement step is denoted $SYS^{e\text{-}pm}$ (where e stands for encoding and pm for pattern matching) and is formally defined as

$$SYS^{e\text{-}pm} \triangleq |||_{A \in Honest} P_A^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm})$$

where $P_A^{e\text{-}pm}$ incorporates the capability of encoding data before applying cryptographic operations on them, and of decoding the outcome of decryption operations.

Similarly to section 3, encoding parameters are formalized as messages $a \in Encoding$, and the encoding of message M performed by actor A using parameters a is formalized as $e_A(a, M)$. As in section 3, condition (5), i.e. the assumption that encodings and decodings are deducible, is assumed to hold, and absence of side effects and of memory between calls is also assumed.

Process $P_A^{e\text{-}pm}$ differs from P_A only by the addition of encoding operations, one before each cryptographic operation. More precisely, in $P_A^{e\text{-}pm}$, each abstract term $\{N\}_K$ occurring in P_A is refined into $\{e_A(a, N)\}_K$, where the choice of a is made so as to comply with the protocol specification documents. Similarly, terms $\{[N]\}_K$, $[\{N\}]_K$, $H(N)$, and K^* (where, as stated above, K^* ranges over $\{K^{\sim}, K^+, K^-\}$) are refined into $\{[e_A(a, N)]\}_K$, $[\{e_A(a, N)\}]_K$, $H(e_A(a, N))$, and $e_A(a, K)^*$ respectively.

The meaning of this refinement is that, when encrypting data, the encoded plaintext is encrypted, instead of its internal representation. Similarly, when building a key, the encoded key material is used instead of its internal representation.

In $P_A^{e\text{-}pm}$ the decoding operations $d_A(a, \cdot)$, corresponding to $e_A(a, \cdot)$, are not represented explicitly. Instead, they are represented implicitly by pattern matching, in the same way as decryption is represented implicitly in the abstract CSP model. For example, $receive.B.A.\{e_A(a, x)\}_K \rightarrow P$ means receiving $\{e_A(a, x)\}_K$, decrypting the received message with key K, decoding the outcome of decryption by using the inverse of $e_A(a, \cdot)$, and finally binding x to the result of decoding, before proceeding with P.

Of course, this model is still rather abstract, because of the pattern matching mechanism, which is not the way decoding is normally implemented. Moreover, using pattern matching introduces a further assumption about $e_A(a, \cdot)$, i.e. $e_A(a, \cdot)$ is assumed to be injective. Encoding functions used in protocols should always be *defined* so as to be injective, because otherwise there could be encoded data that cannot be decoded uniquely. However, one particular *implementation* $e_A(a, \cdot)$ of an injective encoding function could be non-injective, because of implementation errors. For example, let us denote $length(M)$ the length in bytes of a message M, and let us consider the trivial definition of an injective encoding function that encodes a message M as the concatenation of a binary representation of $length(M)$ followed by the $length(M)$ bytes of M. A wrong implementation of this injective function that, for example, trims M to a maximum length K, is non-injective, because it maps all the messages longer than K that have the same prefix to the same encoded message.

After having introduced the first refinement step from SYS to $SYS^{e\text{-}pm}$, sufficient conditions are now introduced under which secrecy and authentication are preserved when transforming back $SYS^{e\text{-}pm}$ into SYS. The conditions for preservation of authentication are stronger than those for secrecy.

One first condition, which is common for both secrecy and authentication, is

$$\forall a, O \;.\; e_A(a, O) = e(a, O) \tag{15}$$

where $e(a, M)$ denotes the *definition* of the encoding function (in contrast with $e_A(a, M)$, which denotes its *implementation* in agent A). This condition means that the encoding implementation in each actor is correct with respect to the specification of the encoding scheme. In practice, if (15) holds, then all actor's implementations of encoding functions are equivalent, so that implementing actors can be ignored. So, the

$e(a, O)$ symbolic form of encoding will be used from now on, regardless of the implementing actor, when condition (15) is assumed to hold.

A second condition, common for both authentication and secrecy, is the injectivity of $e(a, \cdot)$, which normally holds for any well-defined encoding algorithm, as already observed.

A data renaming transformation $f_d(\cdot)$, that transforms $P_A^{e\text{-}pm}$ into P_A, and thus $SYS^{e\text{-}pm}$ into SYS, is defined as the identity function except for the following cases:

$$\begin{array}{rcl} f_d(M, M') & = & (f_d(M), f_d(M')) \\ f_d(\{e(a, M)\}_K) & = & \{f_d(M)\}_{f_d(K)} \\ f_d(\{[e(a, M)]\}_K) & = & \{[f_d(M)]\}_{f_d(K)} \\ f_d([\{e(a, M)\}]_K) & = & [\{f_d(M)\}]_{f_d(K)} \\ f_d(H(e(a, M))) & = & H(f_d(M)) \\ f_d(e(a, K)^*) & = & (f_d(K))^* \end{array}$$

This function is well-defined because $e(a, \cdot)$ is injective. It is the key for proving our preservation results, via Hui and Lowe's theory.

Finally, a third condition, common for both authentication and secrecy, has to be added about the adversary knowledge:

$$AK_0 \supseteq f_d(AK_0^{e\text{-}pm}) \cup f_d(Encoding) \tag{16}$$

Hypothesis (16) is reasonable because it simply requires that the adversary in the abstract system knows at least the simplified form of messages that are known by the adversary in the refined system, along with the encoding parameters.

Preservation of secrecy in the refinement from SYS to $SYS^{e\text{-}pm}$ can now be expressed by the following theorem.

Theorem 4.1

If (15) *holds and* $e(a, \cdot)$ *is injective and* (16) *holds*
SYS ***sat*** *Secrecy* $\Rightarrow$ $SYS^{e\text{-}pm}$ ***sat*** *Secrecy*

In order to prove that secrecy in the abstract system implies secrecy in the refined system, it is enough to show that $f_d(\cdot)$ is actually a fault preserving simplifying transformation that preserves secrecy, which amounts to check that conditions (9) and (10) are satisfied.

Satisfaction of condition (9) can be proved by induction over the knowledge derivation relation $\vdash$; while satisfaction of condition (10) can be proved by hypothesis (16). □

In order to prove that authentication is preserved when refining SYS into $SYS^{e\text{-}pm}$, one further condition that must hold for messages in *AgreementSet* has to be stated. Let us introduce now the *symbolic expression* of a message, that is a term that represents the message but leaving all data encoding operations in their symbolic form $e(a, O)$ (in contrast to resolve them to the resulting term obtained by encoding message O with parameters a).

Using the symbolic expression concept, the *AgreementSet* can be partitioned into equivalence classes. Two messages M and M' belong to the same equivalence class if their symbolic expressions are equal, modulo a renaming of the first argument of each encoding operation occurring in them (namely the a argument of $e(a, O)$). The $M \sim M'$ notation means that M and M' belong to the same equivalence class. For example, if $(H(e(a, O)), N)$ is the symbolic expression of M, and $(H(e(b, O)), N)$ is the symbolic expression of M', then $M \sim M'$ is true; in contrast, if $(H(e(b, O)), N')$ is the symbolic expression of M' and $N \neq N'$, then $M \nsim M'$, because their symbolic expressions also differ by the $N \neq N'$ terms. In other words, each equivalence class contains all the messages that can be obtained by applying encodings with various encoding parameters to the same unencoded message.

The condition to be added for the preservation of authentication is:

$$\forall\, M, M' \in AgreementSet \;\cdot\; M \sim M' \Rightarrow M = M' \tag{17}$$

which states that each equivalence class must have only one element. In other words, it must never happen that *AgreementSet* contains two messages that share the same symbolic expression, except for some encoding parameters. To show why (17) is necessary, consider the following counter-example. Suppose (17) does not hold. In a refined model, agreement could fail because one agent emits his *running* event on $H(e(a, M))$, while another agent emits his *finished* event on $H(e(b, M))$. However, such a fault would not be preserved in the corresponding abstract model, where both events would be done on $H(M)$, letting agreement succeed.

Indeed, the SSH protocol discussed in section 6 performs agreement on a *final hash*, requiring a careful evaluation on whether (17) is satisfied. In practice, (17) is enforced by explicitly including, within each message upon which agreement is required, the parameters that must be used to encode each part of the message itself. For example, agreement on $H(e(a, M))$ becomes agreement on $(a, H(e(a, M)))$. In this way, parameter a is not abstracted away in the abstract model, where agreement happens on $(a, H(M))$.

Preservation of authentication in the refinement from SYS to $SYS^{e\text{-}pm}$ can now be expressed by the following theorem.

Theorem 4.2

$$\textit{If (15) holds and } e(a,\cdot) \textit{ is injective and (16) holds and (17) holds}$$
$$SYS \ \textbf{\textit{sat}} \ Agreement_{f_d(AgreementSet)} \Rightarrow SYS^{e\text{-}pm} \ \textbf{\textit{sat}} \ Agreement_{AgreementSet}$$

In order to prove that $f_d(\cdot)$ preserves agreement, since conditions (9) and (10) have already been proved for theorem 4.1, it is enough to prove (11), that is $f_d(\cdot)$ is locally injective on $AgreementSet$. In other words, if by hypotheses (15), (16) and (17), $f_d(\cdot)$ satisfies condition (11), then this theorem is proved.

Now, function $f_d(\cdot)$ is showed to be locally injective on $AgreementSet$.

Let $M, M' \in AgreementSet$ with $M \neq M'$. If $M \nsim M'$, then M and M' are terms with different structures, or, if they have the same structure, there exist two subterms N and N' with $N \neq N'$, in the same position in M and M' respectively, that cause them to differ. In this case, $f_d(M) \neq f_d(M')$ because it can be easily showed, by structural induction over messages, that $f_d(\cdot)$ preserves message structure and does not alter subterms, except for removing symbolic encoding operations, and their first parameter, which is not what is making M and M' different in this case.

If $M \sim M'$, then, by (17), if follows that $M = M'$, which contradicts the hypothesis, so this case cannot happen. □

By theorem 4.1, one can verify secrecy on the simpler abstract model SYS, and the same property is implied to hold on the more refined model $SYS^{e\text{-}pm}$, under the hypotheses that encoding functions are injective and correctly implemented in all protocol actors (plus the fact that the adversary in the abstract model knows all encoding parameters used by the protocol). By theorem 4.2, proving authentication on SYS implies authentication in $SYS^{e\text{-}pm}$, under the further condition that the encoding parameters used in the agreement terms are themselves agreed upon.

As with theorem 3.2, if extensions of the proposed datatype are used, structural inductions used in the proofs of theorems 4.1 and 4.2 must be checked to hold for the new datatype.

4.2 The Second Refinement Step

As already observed, $SYS^{e\text{-}pm}$ is still very abstract, because of the pattern matching mechanism used for decoding. Sometimes, models like $SYS^{e\text{-}pm}$ are used, with $e(a,\cdot)$ simply defined as a symbolic injective function, and correctness of data encoding and decoding assumed by hypothesis. These models are not much more complex than the corresponding abstract models where data encoding is fully abstracted away, because most of the complexity of data encoding stands in the implementation of encoding and decoding algorithms, which is not represented when encoding functions are symbolic.

The second step of refinement that will be introduced shortly in this section makes the application of data encoding and decoding operations explicit in the model, like it was for the analogous operations in section 3. In this way, the possibility to represent significant parts of the actual implementation of data encoding and decoding algorithms in non-symbolic form is enabled.

In order to make data decoding explicit in the model, the same notation introduced in section 3 is used here: the decoding of message M with parameters a done by actor A is denoted $d_A(a, M)$ and the special atom Atom $\mathcal{E}$ represents a decoding error code returned by the decoding algorithm when M is not recognized as a valid encoding.

For simplicity, in this second refinement step the application of e_A is kept inline, while the application of d_A is delegated to a separate process, which is convenient in order to eliminate the unrealistic pattern matching mechanism for decoding, and make the latter explicit, as typical protocol implementations do.

More precisely, each process $P_A^{e\text{-}pm}$ is refined into a process P_A^e, composed of the protocol logic PL_A^e coupled with a decoding process DEC_A. Each pair of PL_A^e and DEC_A processes internally communicates by the p_send_A and $p\_receive_A$ hidden dedicated channels. The refined system is denoted SYS^e.

The structure of SYS^e with two actors A and B is depicted in figure 8. The formal definition of SYS^e is

$$SYS^e \triangleq (|||_{A \in Honest} P_A^e) \parallel ADV(AK_0^e)$$

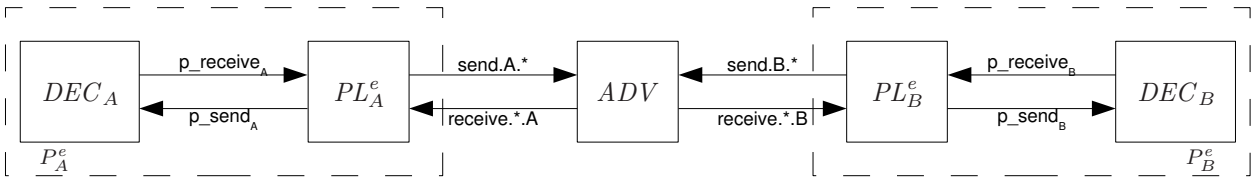

Figure 8: Actors A and B with ADV in SYS^e.

where $P^e_A \triangleq (PL^e_A \parallel DEC_A) \setminus priv_A$ and $priv_A = \{|p\_send_A, p\_receive_A|\}$.

The PL^e_A process incorporates the capability of encoding data before applying cryptographic operations on them, and delegates to DEC_A the task of decoding the plaintext after having decrypted a ciphertext. The call of the decoding function $d_A(a, M)$ is represented by the event $p\_send_A.(M, a)$ while the successful termination of the operation is represented by the event $p\_receive_A.N$, where $N = d_A(a, M)$ is the result of decoding. If decoding is unsuccessful, i.e. $d_A(a, M) = \text{ATOM } \mathcal{E}$, DEC_A gets stuck, and PL^e_A gets stuck too. This behavior, which is realistic for the kind of encoding considered in this section, corresponds to an aborted session.

Each DEC_A process is formally defined as follows:

$$\begin{aligned} DEC_A \quad \triangleq \quad & \Box_{\substack{y \in Message \\ a \in Encoding}} p\_send_A!(y, a) \rightarrow \\ & (p\_receive_A!d_A(a, y) \rightarrow DEC_A) \not\vdash d_A(a, y) \neq \text{ATOM } \mathcal{E} \not\vdash STOP \end{aligned} \tag{18}$$

In order to obtain P^e_A from $P^{e\text{-}pm}_A$, each $receive.B.A.M$ in $P^{e\text{-}pm}_A$ must be turned into a $receive.B.A.M'$, followed by zero or more $p\_send_A.(y, a) \rightarrow p\_receive_A.N'$ prefix pairs, representing the interaction with the DEC_A process. For example, $receive.A.B.\{e_A(a, M)\}_K \rightarrow P$ must be turned into $receive.A.B.\{y\}_K \rightarrow p\_send_A.(y, a) \rightarrow p\_receive_A.M \rightarrow P$.

The refinement transformation from $P^{e\text{-}pm}_A$ to PL^e_A can be formalized by a function r from processes to processes such that $PL^e_A = r(P^{e\text{-}pm}_A)$.

Function r can be defined as

$$\begin{aligned} r(ev \rightarrow P) &= (r_e(ev) \rightarrow r(P)) \\ r(\omega(P_1, \ldots, P_{n_\omega})) &= \omega(r(P1), \ldots, r(P_{n_\omega})) \end{aligned}$$

where ev ranges over event prefixes, ω ranges over CSP operators, n_ω stands for the arity of ω, and r_e is another refinement function that describes how event prefixes are refined. In this definition the usual convention applies according to which the first matching rule is applied. Therefore, according to this definition, only event prefixes are refined, while all other operators are unaffected by the refinement.

According to the idea expressed previously, only *receive* prefixes are refined. Each *receive* event is refined into a similar *receive* event, followed by zero or more pairs of p_send_A $p\_receive_A$ events. The formalization of r_e is

$$\begin{aligned} r_e(receive.B.A?N) &= receive.B.A?F_N\ G_N \\ r_e(ev) &= ev \end{aligned}$$

where $(F_N, G_N) = r_t(N)$ and r_t is another function that describes how terms are refined. r_t takes a term N and returns a pair where the first element F_N is the refined term and the second one G_N is a possibly empty sequence of prefixes that have to be added after the receive event being refined and that represent the interactions with the decoding process.

r_t can be defined by refining encryption terms like $\{e_A(a, N)\}_K$ into $\{y\}_K$, where y is a new fresh variable for storing the received message. The decoding of y is represented by appending the prefixes "$\rightarrow p\_send_A.(y, a) \rightarrow p\_receive_A.N'$", where N' is obtained by recursively applying the same procedure to N.

Formally,

$$\begin{aligned} r_t((N, O)) &= ((F_N, F_O), G_N G_O) \\ r_t(\epsilon_K(e_A(a, N))) &= (\epsilon_K(y), \text{"} \rightarrow p\_send_A.(y, a) \rightarrow p\_receive_A.F_N\text{"}\ G_N) \quad \text{with } y \text{ new fresh variable} \\ r_t(N) &= (N, \epsilon) \end{aligned}$$

where here $(F_N, G_N) = r_t(N)$, $(F_O, G_O) = r_t(O)$, $\epsilon_K(\cdot)$ represents any encryption operator (either $\{\cdot\}_K$ or $\{[\cdot]\}_K$ or $[\{\cdot\}]_K$) for which the inverse of K is known to the process, ϵ represents the empty string and $G_N G_O$ represents the simple concatenation of strings G_N and G_O.

As an example, let us consider the refinement of a process $P_A^{e\text{-}pm}$ defined as

$$receive.B.A.\,\{e_A(a,[\{e_A(b,\text{ATOM }A)\}_{K_1}])\}_{e_A(c,K_2)^+} \rightarrow P$$

where K_1 is a shared key known by $P_A^{e\text{-}pm}$ (in this example K_1 is opaque, e.g. $P_A^{e\text{-}pm}$ has received it as an opaque message), and $e_A(c,K_2)^+$ is a public key for which the corresponding private key $e_A(c,K_2)^-$ is known by $P_A^{e\text{-}pm}$. The refined process PL_A^e, to be coupled with its decoding process DEC_A, is in this case

$$receive.B.A.\{[y_1]\}_{e_A(c,K_2)^+} \rightarrow p\_send_A.(y_1,a) \rightarrow p\_receive_A.\{y_2\}_{K_1} \rightarrow \\ p\_send_A.(y_2,b) \rightarrow p\_receive_A.\text{ATOM } A \rightarrow r(P)$$

In order to be able to perform decryption operations, PL_A^e must know the symmetric key K_1 and the private key $e_A(c,K_2)^-$, as $P_A^{e\text{-}pm}$ does. In this example, PL_A^e receives $\{[y_1]\}_{e_A(c,K_2)^+}$, i.e. a message encrypted with public key $e_A(c,K_2)^+$, and decrypts it by using the known refined private key $(e_A(c,K_2)^-$ (because the private key is required to decrypt public-key ciphered data). Then, the encoded message y_1 and the decoding parameters a are sent to the coupled DEC_A process, which returns the decoded representation. The returned message must match $\{y_2\}_{K_1}$, where y_2 is again in the encoded form required by the cryptographic algorithm. This means that PL_A^e decrypts the returned message with K_1 and gets y_2. Finally, PL_A^e sends y_2 and the decoding parameters b to DEC_A, obtaining the decoded plaintext, which is checked to match ATOM A.

As a second example, let us consider the abstract process $P_A^{e\text{-}pm}$ defined as

$$receive.B.A.\{e_A(b,H(e_A(c,\text{ATOM }A)))\}_{e_A(a,K)^\sim} \rightarrow P$$

where $P_A^{e\text{-}pm}$ is assumed to know K, and thus also $e_A(a,K)^\sim$. The refined process PL_A^e, to be coupled with its decoding process DEC_A, is

$$receive.B.A.\{y\}_{e_A(a,K)^\sim} \rightarrow p\_send_A.(y,b) \rightarrow p\_receive_A.H(e_A(c,\text{ATOM }A)) \rightarrow r(P) \tag{19}$$

In this example, $P_A^{e\text{-}pm}$ receives a message and decrypts it using the symmetric key $e_A(a,K)^\sim$, which must be known by PL_A^e too. The obtained plaintext y should be the encoding, with the parameters b required by the cryptographic algorithm, of the original message $H(e_A(c,\text{ATOM }A))$. Here y is treated by PL_A^e as an opaque message and it is sent along with b to the coupled decoding process DEC_A. The latter returns the internal representation of y, which is checked to match $H(e_A(c,\text{ATOM }A))$.

The inverse of r, i.e. the function that takes back from PL_A^e to $P_A^{e\text{-}pm}$, will be denoted f. Note that f can simply be defined as:

$$\begin{aligned} f(receive.B.A?M \rightarrow p\_send_A.(y,a) \rightarrow p\_receive_A.N \rightarrow P) &= f(receive.B.A?M\,[{}^{e_A(a,N)}/_y] \rightarrow P) \\ f(\omega(P_1,\ldots,P_{n_\omega})) &= \omega(f(P1),\ldots,f(P_{n_\omega})) \end{aligned}$$

Although function f is defined for any CSP process, in order to keep the proofs simpler, from now on it will be assumed that PL_A^e is a sequential process. This assumption does not narrow the generality of our results, since multi-threaded implementations of protocol logics can be simulated by corresponding sequential implementations.

Now that the formal relation between SYS^e and $SYS^{e\text{-}pm}$ has been defined, the following sufficient conditions for fault preservation when abstracting from SYS^e to $SYS^{e\text{-}pm}$ can be formulated:

$$\forall\, a,y \quad d_A(a,y) \neq \text{ATOM } \mathcal{E} \Rightarrow e_A(a,d_A(a,y)) = y \tag{20}$$

$$AK_0^{e\text{-}pm} \supseteq AK_0^e \tag{21}$$

Condition (20) is true if, for each actor, the implementation of the encoding function $e_A(a,\cdot)$ is the inverse of the implementation of the decoding function $d_A(a,\cdot)$. This property will be named the "e/d round-trip property". Condition (21) is true if, at the beginning of the protocol run, the adversary in the intermediate system knows at least the same messages known by the adversary in the refined system.

Differently from section 4.1, no assumption on implementation correctness with respect to any encoding scheme specification is made here, and no relationship is being assumed between encoding scheme implementations of different actors. That is, the e/d round-trip property (20) can be checked in isolation on each implementation alone, without referring to any encoding scheme specification. In addition to the above conditions, the injectivity of $e_A(a,\cdot)$ has to be assumed, otherwise the pattern matching mechanism in $SYS^{e\text{-}pm}$ would not work.

Canonicalization schemes are neglected, because they transform data between two different items of the same equivalence class, and all such items are normally represented by a single term in an abstract formal

model. Indeed, as stated in [KR06] too, canonicalization does not impact security properties, as long as all elements of the same canonicalization equivalence class have the same meaning (which actually is the aim of canonicalization). Moreover, representing canonicalization operations in abstract models would introduce non injective functions, whose interaction with some security properties (e.g. authentication) would be rather complex, despite not so significant.

The fault preservation property in this case can be proved with respect to any trace property (and is based on a different reasoning w.r.t. the simplifying transformations introduced by Hui and Lowe). The intuition is that, by defining $P_A^e \triangleq (PL_A^e \parallel DEC_A) \setminus priv_A$, and assuming the e/d round trip property (20), $P_A^{e\text{-}pm}$ and P_A^e behave in the same way except for *receive* events. While $P_A^{e\text{-}pm}$ gets stuck on *receive* events of non matching messages, P_A^e receives those messages but subsequently deadlocks, within DEC_A. Then, if communication events are disregarded, it is possible to prove that any (fault) trace in SYS^e is also a (fault) trace in $SYS^{e\text{-}pm}$, provided the adversary in $SYS^{e\text{-}pm}$ is not less powerful than the adversary in SYS^e (this condition is expressed by (21)).

Formally, the fault preservation property is expressed by the following theorem and corollary.

Theorem 4.3 *Let* $comm = \{|send, receive|\}$*, and let* $e_A(a, \cdot)$ *be injective. Then, the e/d round-trip property* (20) *and condition* (21) *imply*

$$SYS^{e\text{-}pm} \setminus comm \sqsubseteq SYS^e \setminus comm \tag{22}$$

Corollary 4.4 *Under the same hypotheses of theorem 4.3,*

$$\forall SPEC \cdot SYS^{e\text{-}pm} \textbf{ sat } SPEC \Rightarrow SYS^e \textbf{ sat } SPEC \tag{23}$$

Summing up the results of theorem 4.3 and corollary 4.4, if the adversary knowledge in the abstract system is no less than the adversary knowledge in the refined system, and the e/d round-trip property holds (i.e. the implementation of the encoding function of each actor is the inverse of the implementation of the decoding function of the same actor), then the more abstract $SYS^{e\text{-}pm}$ can be verified instead of SYS^e, for any security property.

By this result, when a model extraction approach like the one presented in [BFGT06] is used, the verification of any security property can be safely divided into two distinct verifications. On one hand, the verification of the property on an abstract protocol model, where all the decoding functions that are modeled in this work by the DEC_A processes are left out. On the other hand, the verification that the sequential code of each encoding procedure implements the inverse of the corresponding decoding procedure.

The proof of corollary 4.4 is now given, and the proof of theorem 4.3 is sketched. The full proof of theorem 4.3 is available in appendix A.2.

[Proof of Corollary 4.4] From theorem 4.3, the trace refinement relation (22) implies

$$SYS^{e\text{-}pm} \setminus comm \textbf{ sat } SPEC \Rightarrow SYS^e \setminus comm \textbf{ sat } SPEC$$

Then, by using the $\Leftarrow$ side of assumption (3), it follows that (23) holds. □

The proof of theorem 4.3 uses the following lemma, which states the trace refinement relationship that binds $P_A^{e\text{-}pm}$ and P_A^e.

Lemma 4.5 *Let* $comm_A \triangleq \{|send.A, receive?B.A|\}$*. If* (20) *and* (21) *hold, then, for any* $A \in Honest$*,*

$$(P_A^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm})) \setminus comm_A \sqsubseteq (P_A^e \parallel ADV(AK_0^e)) \setminus comm_A \tag{24}$$

This lemma descends from the fact that, as already observed, the difference between $P_A^{e\text{-}pm}$ and P_A^e stands only in $receive?B.A$ events. If such events are hidden, and $P_A^{e\text{-}pm}$ is combined with an adversary that is not less powerful than the one combined with P_A^e, the fault traces occurring in P_A^e can occur in $P_A^{e\text{-}pm}$ too. Technically, the proof of lemma 4.5, which is in appendix A.3, is based on proving that the two sides of (24) are bound by a weak simulation relationship. This relationship, which formalizes the intuition given above, implies trace refinement.

[Proof sketch of theorem 4.3] Let us start from $SYS^{e\text{-}pm}$ and let us substitute, one by one, each $P_A^{e\text{-}pm}$ into the corresponding P_A^e. At each substitution step a new process is obtained that can be proved, by using lemma 4.5, to be a trace refinement of the previous one. More precisely, this part of the proof is done by induction.

The base of the induction is

$$\forall\, X \in Honest \cdot SYS^{e\text{-}pm} \setminus comm \sqsubseteq \left(\left(|||_{A \in Honest \setminus \{X\}} P_A^{e\text{-}pm}\right) ||| P_X^e\right) \parallel ADV(AK_0^e)) \setminus comm$$

This relation can be proved to hold by using lemma 4.5, the property of $\sqsubseteq$ according to which for any context $C[]$ we have $A \sqsubseteq R \Rightarrow C[A] \sqsubseteq C[R]$, and a distribution-like property of hiding over parallel composition in CSP.

The induction step is formulated as

$$\begin{array}{ll} & \forall\ X \in Honest \setminus Ref \\ & \big(\big(|||_{A \in Honest \setminus Ref} P_A^{e\text{-}pm} \ ||| \ |||_{B \in Ref} P_B^e\big) \parallel ADV(AK_0^e)\big) \setminus comm \\ \sqsubseteq & \big(\big(|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^{e\text{-}pm} \ ||| \ |||_{B \in Ref \cup \{X\}} P_B^e\big) \quad \parallel ADV(AK_0^e)\big) \setminus comm \end{array}$$

The induction step can be proved in a way similar to the base. □

5 Introducing Equational Theories

Equational theories are sometimes useful in security protocol models, when different forms of a message need to be considered equal. For example, one may want to identify nested pairs with different associativity, like for example $((M, N), O)$ and $(M, (N, O))$, if one is just interested in the ordered list of elements contained in a nested pair.

A way to introduce an equational theory in the protocol model is explained in [HL01], and is based on the definition of a normal form for messages. As the equational theory introduces an equivalence relation on the set of messages, for each equivalence class a unique message of that class is selected as the normal form of all the messages belonging to the class.

Formally, a normal form function $nf : Message \rightarrow Message$ is introduced that turns a message into its normal form. The property that nf must satisfy in order to be a valid normal form function is

$$nf(M) = nf(N) \iff M =_E N \tag{25}$$

where $=_E$ means equality modulo the equational theory while $=$ is the original syntactic equality of the term algebra. In other words, if we take the normal form of messages, equality modulo the equivalence relation is reduced to syntactic equality.

In [HL01], it is showed how a normal form function can be defined for pairs, so that, for example, $(M, (N, O))$ and $((M, N), O)$ are considered equal, but, of course, a normal form can be defined for other equational theories as well.

After having defined nf, the models of protocol actors are constrained to use only messages in normal form by applying nf to any message expression occurring in any protocol actor process. In particular, since send and receive action prefixes now take the form $send.A.B.nf(M)$ and $receive.B.A.nf(M)$ respectively, each actor process can only send and receive messages in normal form. Similarly, all $claimSecret$, $running$ and $finished$ events can only occur with messages in normal form. Only $leak$ events, that are executed by the adversary without synchronization with actor processes, may in principle lead messages that are not in normal form.

In order not to limit the adversary power, when a normal form is introduced it is necessary to ensure that the adversary can always compute $nf(M)$, so that all the messages that are in the adversary's knowledge can be leaked in the normal form. To do this, the following new rule is added to the definition of the knowledge derivation relation:

$$\textit{normal form}\text{: } U \vdash M \ \Rightarrow \ U \vdash nf(M)$$

Should this be omitted, our definition of secrecy would not catch the right Dolev-Yao secrecy concept. For example, it would be possible that $claimSecret$ happens on a normal form message $nf(M)$, and the adversary can derive an equivalent message M that is not in normal form, but he cannot derive $nf(M)$ itself. In this case, secrecy as defined in this paper would hold, despite the message not being really secret according to the Dolev-Yao secrecy concept in the presence of an equational theory. The normal form derivation rule solves this issue, making the secrecy definition of this paper match with the Dolev-Yao secrecy concept.

With some equational theories the additional derivation rule is superfluous: for example, with equality modulo associativity of pairs, the only way for the adversary to derive a message M that is not in normal form is by the pairing rule, and $nf(M)$ can also be derived using the pairing rule in a different way. However, with other equational theories it is necessary to keep the normal form derivation rule explicit; an example will be provided in the next section.

As also observed in [HL01], the definition of a normal form is very much orthogonal to how the results about fault-preserving simplifying transformations have been obtained. In particular, it can be proved that

all the theorems about fault preservation of renaming transformations that are used in this paper are still valid when using a normal form for messages in the way explained above. This implies that theorems 3.2, 4.1, and 4.2 still hold.

Let us now consider the other theorems that do not rely on renaming transformations.

The proof of theorem 3.1 is based on a re-parenthesization of processes, which, of course, can be done in the same way when normal forms are used. The core of the proof is then the proof that $ADV\text{-}ML$ is a trace refinement of ADV. This still holds when using a normal form for messages because the ADV process can always derive the normal form of any known message.

In theorem 4.3, the exploited CSP properties are valid under the condition that the alphabets of the protocol agents are disjoint, which is still satisfied in case the normal form of messages is used. Indeed, the protocol agents' alphabets are disjoint (so that they use the adversary as the proxy for their communication) due to the agent names used within the events, rather than the exchanged messages.

In lemma 4.5, the simulation relation is not defined on messages, and so it remains unchanged. Finally, all analyzed traces can happen, and the subset relation between the adversary knowledges in the two systems still holds when a normal form is used, because the adversary can always generate messages in normal form.

6 Applications and Examples

This section shows some practical applications of the results illustrated in this paper. First, it is showed how all sufficient abstraction conditions stated in this paper can be met on a class of concrete encoding schemes, that includes XML encodings.

Then, the modeling of an SSH Transport Layer Protocol client is presented. Both an abstract model and a refined one are provided, along with checks or assumptions needed on implementation code in order to preserve security properties from the abstract model to the refined one. The refined model includes altogether marshaling/unmarshaling functions for network messages, as explained in section 3, and encoding/decoding functions for data on which cryptographic operations are applied, as illustrated in section 4.

6.1 A Class of Data Encodings including XML Encodings

The encoding schemes considered in this example simply add a header to each part of the message being encoded, and do not alter the message content itself. Formally:

$$\begin{array}{rcl} e(a,(M,M')) & = & (e(a,M),e(a,M')) \\ e(a,M) & = & (head(a,M),M) \end{array} \tag{26}$$

where $head(a,M) \in deds(\{a,M\})$ is a header that may depend on parameters a and on message M. The peculiarity of this kind of encoding function is that it distributes headers across pairs. It is possible to define the symmetric encoding function, that adds a trailer, or padding, to data; it is furthermore possible to combine the two.

This class of encoding schemes is general enough to include, for example, XML encodings. Then, it can be used to model the data encodings used in the protocols of the WS-Security [NKHBM06] standard.

In [BFG06], an implementation of the XML encoding scheme where XML fragments are internally represented as F# (an ML dialect) records is described. For example, the XML security header specified in the WS-Security standard is internally stored as:

```
type security = {
  timestamp: ts;
  utoks: utok list;
  xtoks: xtok list;
  ekeys: encrkey list;
  dsigs: dsig list }
```

Then, a set of `gen*` and `parse*` functions, implemented in F#, is used to translate internal records to and from XML, for example when an XML fragment must be encrypted.

In the CSP modeling framework for cryptographic protocols used in this paper, an F# record (i.e. the internal data representation) can be modeled by means of nested pairs. The F# functions translating to and from XML, are actually encoding functions that add some XML header and trailer to each element of the record, that is to the nested pairs. Thus, the `gen*` and `parse*` functions behave like the encoding scheme defined in (26).

If these functions can be proved side-effect free and memoryless, and if they only use the given input parameters or hard-coded values (which amounts to check an information flow property on the implementation code), then all the sufficient conditions required in section 3 are satisfied. So, if these functions are used to generate the XML encoding that is exchanged on communication channels, then they can be safely left out when using the model extraction approach.

Moreover, assuming that the conditions required in section 3 are met, in order to check that the implementation of these functions satisfies theorem 4.3, it is enough to additionally check that, on one hand, the `gen*` functions only add a tagged fragment before and after any record element, leaving the record element unchanged (with the exception of canonicalization operations, which are abstracted in the model). The added tagged fragment may be anything that can be correctly recognized in the decoding phase. This implies that the tagged fragment is something that cannot occur as part of a normal message to be encoded, which also implies injectivity of the encoding function implementation. For theorem 4.3 to hold, correctness of implementation w.r.t. the XML specification is not required. On the other hand, it is enough to check that the `parse*` functions match some expected tagged header and trailer, that must comply with the expected record type and value, and that they store the data between header and trailer into the proper record field without further modification (with the exception of canonicalization operations).

For applying theorems 4.1 and 4.2, the correctness of the encoding functions implementation w.r.t. their specification is additionally needed. This amounts to the extra check that the tagged data added by the encoding `gen*` functions, and recognized by the decoding `parse*` functions, actually comply with the XML and WS-Security standards. However, this simplification step is less important, because most of the complexity is eliminated in the first step.

6.2 Experimental Results

In [BFGT06], a model extraction approach for security protocols was proposed. The FS2PV tool automatically extracts a ProVerif [Bla01, AB05] model from the F# implementation of a protocol, and then security properties are checked with the ProVerif tool on the extracted model. A ProVerif model is specified in applied pi calculus [AF01]. Although a formal mapping between applied pi calculus and CSP is outside the scope of this paper, both modeling languages rely on a Dolev-Yao algebraic datatype and adversary. So, when experimentally evaluating our approach, the ProVerif tool and applied pi calculus will be used, although the results in this paper are formally developed in CSP only.

In [BFGT06], when model extraction is performed, a model is extracted from the `gen*` and `parse*` function implementations too, and a global protocol model including the model of these encoding functions is formally analyzed. Moreover, messages are modeled in their encoded, complex form. As reported by some of the same authors in [BBF+11] when reviewing the model extraction approach with FS2PV,

> "Even if ProVerif scales up remarkably well in practice, beyond a few message exchanges, or a few hundred lines of F#, verification becomes long (up to a few days) and unpredictable (with trivial code changes leading to divergence)."

Applying the simplifications proposed here is a way to reduce model size and verification complexity.

In this section, the gain obtained with the proposed approach is evaluated in terms of verification time on a case study based on FS2PV and ProVerif.

In order to provide a fair evaluation, the most recent (and optimized) versions of the tools are used. The current version of FS2PV comes with the Otway-Rees example only, the other examples presented in [BFG06] having been discontinued. Hence, our evaluation is done on the Otway-Rees protocol.

The ProVerif model extracted from the full F# implementation of the protocol is 578 lines long, while the ProVerif model extracted by ignoring the `gen*` and `parse*` functions is almost half in size, counting 324 lines. Verification of the simplified model is on average 1.32 ± 0.01 times faster than verification of the same properties on the refined model. The reductions in size and verification time can in general mitigate the problems exposed in [BBF+11]. In particular, the reduction in model size seems a key factor for reducing unpredictability and divergence, according to the experience reported in [BBF+11].

With the current version of the FS2PV tool, we had to manually alter the F# code so that the encoding/decoding functions could be ignored. However, it would be fairly easy to extend the FS2PV tool so that functions belonging to user-specified F# modules would be ignored. So, at the cost of implementing the extension once, one could get the benefit of smaller models and faster verification every time the tool is used.

1. $SSHClient(IDC, me, you, CAlgs) =$
2. $\quad send!me!you!IDC \rightarrow receive!you!me?IDS \rightarrow$
3. $\quad \sqcap_{cookieC \in Cookies}\, send!me!you!(cookieC, CAlgs) \rightarrow$
4. $\quad receive!you!me?(cookieS, SAlgs) \rightarrow$
5. $\quad$ `let` $g = Negotiate(CAlgs, SAlgs, `g') \in CryptoParameter$ `within`
6. $\quad$ `let` $p = Negotiate(CAlgs, SAlgs, `p') \in CryptoParameter$ `within`
7. $\quad \sqcap_{x \in DHSecrets}\, send!me!you!EXP_p(g, x) \rightarrow$
8. $\quad receive!you!me?(PubKeyS, DHPublicS, [\{H(finalHash)\}]_{PriKeyS}) \rightarrow$
9. $\quad GO(EXP_p(DHPublicS, x), finalHash, PubKeyS, IDS)$

Figure 9: A possible fully abstract model of an SSH-TLP client.

6.3 Modeling an SSH Transport Layer Protocol Client

In this example, lists of messages, like for example (M, M', M''), stand for corresponding nested right-associating pairs, i.e. (M, M', M'') stands for $(M, (M', M''))$.

The SSH Transport Layer Protocol [YL06b] (SSH-TLP) is part of the SSH three protocols suite [YL06a]; 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.

6.3.1 Modeling SSH-TLP abstractly

A possible fully abstract model of an SSH-TLP client is reported in figure 9. According to our modeling approach, this model is part of the abstract SYS, where one or more instances of $SSHClient$ can partake together with one or more instances of the SSH-TLP server model, and with the adversary.

The $SSHClient$ process starts a protocol session with the server at line 2 by sending it the client identification string denoted IDC. The server responds with IDS, the server identification string, which is received by the client at the same line. Then, at line 3, the client generates and sends a nonce $cookieC$, followed by the client lists of supported algorithms $CAlgs$. The generation of a nonce is represented in CSP as an internal choice on the set $Cookies \subseteq Atom$ of all possible nonces. The server responds by sending a nonce $cookieS$, followed by the server lists of supported algorithms $SAlgs$, which are received by the client at line 4. Then, at lines 5–6, the client computes the value of the Diffie Hellman (DH) parameters g and p by computing 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 $SAlgs$. $CryptoParameter \subseteq Message$ is the set of messages that can be used as cryptographic algorithms or parameters. Once g and p have been obtained, at line 7 the client generates a random private key x and sends $EXP_p(g, x)$ (its DH public key, as explained below), which is a message representing the result of the modular exponentiation $e = g^x \bmod p$.

At line 8, the client receives a message containing the opaque server public key $PubKeyS$, the opaque server DH public key $DHPublicS$ and the server signed final hash $[\{H(finalHash)\}]_{PriKeyS}$. As prescribed by the signature algorithm, the server signature of $finalHash$ is performed by encrypting, with the private key, the hashing of the message to be signed, rather than the message itself. The server computes its DH public key as $DHPublicS = EXP_p(g, y)$, 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. Analogous reasoning holds for public and private server keys $PubKeyS$ and $PriKeyS$. The term $finalHash$ is the value upon which agreement is required, and contains all the relevant data of a protocol session. At line 8 of figure 9, this term has not been expanded for simplicity. Its expansion is:

$$finalHash = H(IDC, IDS, (cookieC, CAlgs), (cookieS, SAlgs), PubKeyS, \\ EXP_p(g, x), DHPublicS, EXP_p(DHPublicS, x))$$

The term $EXP_p(DHPublicS, x)$, that is used inside the final hash, is the DH shared key as computed by the client. This is the main session secret, shared between the client and the server. Finally, the $GO(\cdot)$ process invoked at line 9 deals with the security events of the protocol and is defined as

$$GO(DHKey, finalHash, PubKeyS, IDS) = \\ \quad (claimSecret.me.you.DHKey \rightarrow finished.me.you.finalHash \rightarrow STOP) \\ \not\triangleleft\, PubKeyS == TrustedKeyOf(IDS) \,\not\triangleright\, STOP$$

That is, if the received server public key $PubKeyS$ corresponds to the locally stored trusted key for the server identified by IDS, which is retrieved by the function $TrustedKeyOf(IDS)$, 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 $finalHash$. Agreement on $finalHash$ implies agreement on all the data items on which it is computed. However, since the final hash is used in later phases of the SSH protocol with other data in order to establish a set of session keys, it is required that actors agree explicitly on $finalHash$, and not only on its contents.

To model the DH modular exponentiation properties, a new operator Exp is added to the datatype:

$$\textsc{Exp}\ Message\ Message\ Message$$

along with the syntactic sugar $\textsc{Exp}\ M\ N\ O = EXP_O(M,N)$. From the point of view of the adversary knowledge derivation relation $\vdash$, Exp represents a non invertible function like a hash. Accordingly, only a single exponentiation rule is needed, defined as

$$exponentiation\text{:}\quad U \vdash M \wedge U \vdash N \wedge U \vdash O \Rightarrow U \vdash EXP_O(M,N)$$

This rule is like the hashing one. For this reason, all previously proved results still hold.

Now, to model the DH property such that the same DH shared key is obtained by a pair of agents via modular exponentiation of their own private DH key with the other agent's public DH key, the following equation must be modeled

$$EXP_O(EXP_O(M,N),N') = EXP_O(EXP_O(M,N'),N) \tag{27}$$

The two sides of the equation represent the same DH shared key, as computed by each agent. Then, equation (27) is such that this DH shared key is considered the same by all parties, regardless of the agent that computed it.

Equation (27) entails an equational theory in the CSP model, and this can be treated by the approach explained in section 5.

The equational theory induced by (27) is a commutative property, so a normal form can be defined by first defining a total ordering of messages and then taking as the normal form of equivalent messages the one where the arguments that can commute occur in the order defined by the total ordering.

The total ordering on messages can be defined by first introducing a total ordering on the set of atoms and a total ordering on the set of data operators. Formally, let us denote $<_A$ the strict total order relation on $Atom$, Δ the set of datatype operators, $<_\Delta$ the strict total order relation on Δ and $<$ the strict total order relation on $Message$. Also, let δ range over Δ and $n(\delta)$ denote the arity of δ.

Then, $<$ can be defined as the least relation that satisfies the following implications:

$$\begin{array}{lcl}
A <_A A' & \Rightarrow & \textsc{Atom}\ A\ <\ \textsc{Atom}\ A' \\
\delta <_\Delta \delta' & \Rightarrow & \delta\ Arg_1\ \cdots\ Arg_{n(\delta)}\ <\ \delta'\ Arg'_1\ \cdots\ Arg'_{n(\delta')} \\
M_i < M'_i\ \wedge\ \forall j \in [1, i-1]\ M_j = M'_j & \Rightarrow & \delta\ M_1\ \cdots\ M_{n(\delta)}\ <\ \delta\ M'_1\ \cdots\ M'_{n(\delta)}
\end{array}$$

It is straightforward to prove that $<$ is transitive and such that for any $M, N \in Message$ exactly one of $M < N$, $M = N$, and $N < M$ holds, which implies that $<$ actually defines a total ordering on $Message$.

The normal form function can finally be defined as follows:

$$\begin{array}{rcll}
nf(\textsc{Atom}\ A) & = & \textsc{Atom}\ A & \\
nf(EXP_O(EXP_O(M,N),N')) & = & \begin{cases} EXP_{nf(O)}(EXP_{nf(O)}(nf(M),nf(N)),nf(N')) & \text{if } nf(N) < nf(N') \\ EXP_{nf(O)}(EXP_{nf(O)}(nf(M),nf(N')),nf(N)) & \text{else} \end{cases} \\
nf(EXP_O(M,N)) & = & EXP_{nf(O)}(nf(M),nf(N)) & \\
nf(\delta\ M_1\ \cdots\ M_{n(\delta)}) & = & \delta\ nf(M_1)\ \cdots\ nf(M_{n(\delta)}) &
\end{array}$$

The proof that (25) holds with this definition (and hence this is a valid normal form) can be done by induction on the size (i.e. number of operators) of M. The base case (size=1) corresponds to $M = \textsc{Atom}\ A$ and is trivially true, while the induction case can be proved by using the definition of nf. Interestingly, this normal form function is one that could not necessarily be computed by the adversary unless we give explicitly the adversary the capability to do so, because $EXP_O(EXP_O(M,N),N') \notin deds(\{EXP_O(EXP_O(M,N'),N)\})$ without the addition of the normal form derivation rule $U \vdash M\ \Rightarrow\ U \vdash nf(M)$.

Finally, such definition of the normal form leads to the desired DH property. In the DH key exchange algorithm, if K and K' are the DH private keys, then $EXP_p(g,K)$ and $EXP_p(g,K')$ are the DH public keys, where g and p are the DH group parameters, while $EXP_p(EXP_p(g,K),K')$ and $EXP_p(EXP_p(g,K'),K)$ both represent the DH shared key that can be obtained by each actor. Only one of the two forms (the normal form) will actually occur in the protocol, the other one being converted to the normal form by function nf.

6.3.2 Refining the SSH-TLP model and checking sufficient conditions for abstraction

This section shows how the abstract model presented in the previous section can be refined, according to the modeling approach presented in this paper, and how the checks required to safely abstract the refined model can be done.

Two kinds of details that have been studied in isolation, namely the marshaling layer, as modeled in section 3, and the encoding of data to be ciphered or hashed and key material, as modeled in section 4, are applied altogether in this model.

Moreover, a new kind of refinement, related to cryptographic algorithms and parameters, is needed in this example. For example, it is needed to distinguish whether a hash is performed by using the MD5 or the SHA-1 algorithms, or whether encryption is 3DES or AES. The abstract datatype defined in section 2, intentionally represents encryption or hashing independently of how they are concretely implemented.

Different real cryptographic algorithms (e.g. SHA-1 or MD5) implement cryptographic primitives (e.g. a hash function) in different, incompatible ways, and missing information on the used algorithms may lead to miss possible security faults. For example, the SSH-TLP itself requires agreement on the value of the final hash, which is then used as the material to build a set of shared session keys. If both actors are to obtain the same $finalHash$, and compute the $finalHash^{\sim}$ key, it is a prerequisite to key agreement that they have previously agreed on the same hashing algorithm and key construction algorithm. Otherwise, they could obtain the same symbolic value (a shared key obtained from a non invertible representation of the hashed data), but different concrete values. Then, key agreement may fail in the concrete world, even if the actors agree on the abstract $finalHash^{\sim}$ key.

In order to faithfully describe this issue, cryptographic algorithms and their parameters are sometimes introduced in abstract descriptions. For example, in [HL01, BFGT06], encryption functions are distinguished according to the algorithm and parameters they use.

In this example, in order to handle cryptographic algorithms and parameters, one could extend the datatype and the associated derivation relation $\vdash$ showed in section 2. The obtained datatype would be similar to the one presented in [HL01]. However, this approach would not give us the opportunity to easily discuss about the conditions under which abstracting these details is safe.

So, the idea is to fit cryptographic algorithms and parameters inside the current datatype, and then to abstract them away, by reusing some results already obtained in this paper. In practice, the cryptographic algorithms and parameters are taken from the set $CryptoParameter \subseteq Message$ and are added as the first argument of each cryptographic operation. The refined model that is obtained in this way corresponds to the ones used in [HL01, BFGT06] where different encryption, decryption and hashing functions are used for each different choice of algorithms and parameters.

For example, the refined encryption

$$[\{RSA, H(SHA\_1, M)\}]_{PriKey}$$

expressed in the modeling framework being presented here, corresponds, using an approach like the one presented in [HL01, BFGT06], to a term like $RSAEncrypt(PriKey, SHA1Hash(M))$, i.e. the hashing of M performed using the SHA-1 algorithm and associated parameters, encrypted with the RSA algorithm and associated parameters.

The three different kinds of refinements must be applied on the same system in the correct order, so as to avoid improper interactions: first, data to be ciphered or hashed and key material should be refined; then cryptographic algorithms and parameters should be added; finally the marshaling layer should be introduced.

The refined model for the SSH-TLP client includes, in addition to a refined model of the client protocol logic, which can be rewritten as in figure 10, a marshaling layer model ML_A, and a decoding process model DEC_A. Most of the complexity of the protocol indeed stands in ML_A and DEC_A.

In figure 10, *string* denotes the SSH string encoding, *bytes* denotes a raw encoding (a sequence of bytes), *mpint* denotes the SSH encoding of a multiple precision integer and *namelists* denotes the SSH encoding of a list of lists of strings. The *bin_pack* parameter specifies that the SSH-BPP encoding as depicted in figure 2(b) must be used, with a *payload* made up of as many fields as the number of subsequent parameters, each of which in turn specifies the encoding to be applied to each field. So, for example, KEX specifies the encoding of an SSH key exchange packet, which is an SSH-BPP packet with a payload that includes a sequence of bytes followed by a list of strings.

Similarly to the abstract model, at line 2, the client exchanges client and server identification strings by the first two protocol messages, while at lines 3–4 the client and server key exchange messages are sent, containing the client and server nonces and the lists of supported algorithms. Then, at lines 5–10 algorithms negotiation takes place. While in the abstract model only p and g where explicitly negotiated, because they

1. $SSHClientRef(IDC, me, you, CAlgs) =$
2. $\quad send!me!you!(\text{ATOM}\ \mathcal{L}, (string, IDC)) \rightarrow receive!you!me?(\text{ATOM}\ \mathcal{L}, (string, IDS)) \rightarrow$
3. $\quad \sqcap_{cookieC \in Cookies}\ send!me!you!(\text{ATOM}\ \mathcal{L}, (KEX, (cookieC, CAlgs))) \rightarrow$
4. $\quad receive!you!me?(\text{ATOM}\ \mathcal{L}, (KEX, (cookieS, SAlgs))) \rightarrow$
5. $\quad$ `let` $g = Negotiate(CAlgs, SAlgs, `g') \in CryptoParameter$ `within`
6. $\quad$ `let` $p = Negotiate(CAlgs, SAlgs, `p') \in CryptoParameter$ `within`
7. $\quad$ `let` $FinalHashAlg = Negotiate(CAlgs, SAlgs, `FinalHashAlg') \in CryptoParameter$ `within`
8. $\quad$ `let` $SignHashAlg = Negotiate(CAlgs, SAlgs, `SignHashAlg') \in CryptoParameter$ `within`
9. $\quad$ `let` $SignMode = Negotiate(CAlgs, SAlgs, `SignMode') \in CryptoParameter$ `within`
10. $\quad$ `let` $SignPadding = Negotiate(CAlgs, SAlgs, `SignPadding') \in CryptoParameter$ `within`
11. $\quad \sqcap_{x \in DHSecrets}\ send!me!you!(\text{ATOM}\ \mathcal{L}, ((bin\_pack, mpint), EXP_p^e(g, x))) \rightarrow$
12. $\quad receive!you!me?(\text{ATOM}\ \mathcal{L}, ((bin\_pack, string, mpint, string),$
$\qquad (PubKeyS, DHPublicS, [\{((SignMode, SignPadding), y)\}]_{PriKeyS}))) \rightarrow$
13. $\quad int\_send_{me}!(y, (SignMode, SignPadding)) \rightarrow$
14. $\quad int\_receive_{me}!H(SignHashAlg, e_{me}(SignHashAlg, finalHash\_enc)) \rightarrow$
15. $\quad GO(EXP_p^e(DHPublicS, x), finalHash\_enc, PubKeyS, IDS)$

where:

$$KEX = (bin\_pack, bytes, namelists)$$
$$EXP_p^e(a, b) = EXP_{e_{me}(p)}(e_{me}(a), e_{me}(b))$$
$$finalHash\_enc = H(FinalHashAlg, e_{me}(FinalHashAlg, (IDC, IDS, (cookieC, CAlgs),$$
$$(cookieS, SAlgs), PubKeyS, EXP_p^e(g, x), DHPublicS, EXP_p^e(DHPublicS, x))))$$

Figure 10: A possible refined model of an SSH-TLP client.

were needed by the EXP function anyway, in the refined model all negotiated algorithms and parameters occur. In particular, the $FinalHashAlg$, $SignHashAlg$, $SignMode$, and $SignPadding$ variables represent the cryptographic algorithms and parameters (such as SHA-1 for $FinalHashAlg$ or RSA for $SignHashAlg$) that are negotiated for the protocol run. As in the abstract model, at line 11, the client sends his DH public key to the server, while at line 12 he receives the server public key, the server DH public key, and the server signature of the final hash, performed with the server private key. Interestingly, since the client has the corresponding server public key (also received at line 12), at line 12 the result of decrypting the server signature is stored into variable y, which represents the encoded form of the data signed by the server. At lines 13–14, the client deals with his encoding layer DEC_A, sending the encoded signed data to DEC_A, and getting back the internal representation of such signed data. Finally, the protocol run ends at line 15 with the invocation of the GO process, dealing with the security events for the protocol run.

The $e_{me}(\cdot)$ function used in this refined model represents the implementation of encoding transformations. Its detailed description is omitted here for simplicity.

Let us show now how, by the results presented in this paper, this refined model can be simplified to perform formal verification of security properties, and what checks or assumptions are needed on the sequential code that implements encoding/decoding functions in order to safely apply each simplification.

In order to remove the models of the marshaling layer ML_A from the refined system model SYS^m, the two steps described in figure 5 are applied in order. Step 1, which is justified by theorem 3.1, requires by the definition (6) that the adversary knowledge used for verification in the abstract system includes encoding parameters (already showed to be a reasonable constraint). Moreover, the implementation of the marshaling functions must be checked to be memoryless, and to access no external data but their input parameters (which amounts to check an information flow property on sequential code). Since the properties to be verified for this protocol are secrecy and authentication, also step 2 can be performed, further abstracting the ATOM $\mathcal{L}$ term and the messages in $Marshaling$ that have been added by the marshaling layer refinement procedure. This step, justified by theorem 3.2, can be applied without the need of further checks.

In order to abstract away the decoding processes DEC_A, added by the refinement about encoding functions applied to data to be ciphered or hashed, according to theorem 4.3 three conditions have to be checked: (i) the injectivity of the encoding function implementation $e_{me}(\cdot)$ must be checked on its sequential code; (ii) the e/d round-trip property (20) must be checked on the sequential code of the implementation of the encoding and decoding functions $e_{me}(\cdot)$ and $d_{me}(\cdot)$; and (iii) the same data flow properties already specified for the marshaling and unmarshaling functions must be checked. Finally, the adversary knowledge must not be smaller in the abstract system w.r.t. the adversary knowledge in the refined system, which is enforced when verifying the abstract system.

When verifying secrecy, by theorem 4.1 even data encodings can be abstracted, by removing the $e_{me}(\cdot)$ functions from the client role model, provided the implementation of the sequential encoding/decoding functions is showed to be correct with respect to their specification; this check can be done in isolation. The same simplification can be applied when verifying authentication, by exploiting theorem 4.2, but in this case

condition (17) must be guaranteed to hold too. As discussed in section 4.1 when introducing condition (17), this can be ensured by explicitly adding the negotiated parameters to the internal actions used to specify agreement. For example, the $finished$ action should become

$$finished.me.you.(finalHash\_enc, FinalHashAlg)$$

This condition is needed even though the negotiated algorithm is already included in the agreed $CAlgs$ and $SAlgs$, because it is necessary to ensure that the specific negotiated algorithm is agreed, rather than the sets of algorithms from which it is selected.

The models of cryptographic algorithms and parameters only affect data terms including cryptographic operations. As previously hinted, the idea is to reuse some results presented in this paper, namely the modeling approach developed in section 4, in a particular way, such that the models of cryptographic algorithms and parameters can be represented as encodings in the current datatype, rather than being explicitly added to it. The main advantage is that the sufficient abstraction conditions already showed in section 4 can be directly reused in order to abstract cryptographic algorithms and parameters too.

So, referring to the modeling approach presented in section 4, let us define the following encoding:

$$\begin{gathered} e(a, M) = (a, M) \\ d(a, (a, M)) = M \\ d(a, N) = \textsc{Atom}\ \mathcal{E},\ \text{if } N \text{ does not take the form } (a, M) \end{gathered} \tag{28}$$

that formally defines the refinement that models cryptographic algorithms and parameters. Clearly, by these definitions $e(a, \cdot)$ is injective and the e/d round-trip property (20) is satisfied. Here, $a \in CryptoParameter$ are the cryptographic algorithms and parameters chosen according to the protocol specification documents. However, this refinement is only a data refinement; in particular it does not add any decoding process, nor the associated int_send and $int\_receive$ actions, because there is no direct mapping of this refinement to protocol implementations. Indeed, the Dolev-Yao model used in this paper assumes perfect cryptography, so the only errors that can be found by this refinement are logic ones, not implementation specific ones.

Then, in order to verify the secrecy property, by the results stated in section 4, cryptographic algorithms and parameters can be abstracted away with no additional check. When verifying authentication, they can be abstracted away if condition (17) holds, which can be ensured as already explained for the abstraction of encodings applied to data on which cryptographic operations are performed.

7 Conclusions

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

The main contributions of the paper are:

- the formulation of a set of sufficient conditions under which the models of data encoding and decoding functions can be safely simplified or even completely abstracted away under the Dolev-Yao assumption
- the formal justification of the above conditions

It has been showed that different conditions apply to different kinds of encoding and decoding operations. Specifically, two kinds of data transformations can be distinguished.

For what concerns marshaling functions, a refined protocol model corresponding to a typical layered implementation of such operations has been defined. It has been proved 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 adversary 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, that excludes explicit models of encoding and decoding functions but preserves the parameters of such operations, 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 and implementations, namely the memoryless and side effect free ones. Moreover, no assumption has been made about invertibility nor about implementation correctness of marshaling functions; instead, it is only required that the implementation of such marshaling functions satisfies some information flow properties, that can be checked by standard static analysis techniques. The consequence of this result is that, if such information flow properties are satisfied on implementation code,

then even erroneous specifications or implementations of encoding schemes cannot be more harmful than a Dolev-Yao adversary is.

Then, in order to deal with the wider class of encoding functions, including the ones applied to data to be ciphered or hashed, a more general model supporting a wide class of data encoding schemes has been introduced. On this model, it has been showed that the models of the decoding operations can be safely omitted when verifying security properties, but under stricter constraints: it is required that the implementations of encoding functions are injective and that they are the inverses of the corresponding decoding functions for each actor. If secrecy is being verified, then the encoded form of messages can also be safely abstracted away, by additionally checking that the implementation of the encoding and decoding functions is correct w.r.t. their specification and by ensuring that encoding parameters are in the adversary knowledge. If instead authentication is being verified, then the encoded form of messages can be abstracted if all actors agree on the same encoding parameters too. This condition can be checked as part of the abstract protocol verification, as shown in section 6.

Therefore, an important distinction in criticality has been showed to exist between marshaling functions, for which neither invertibility nor correctness is needed, and encodings applied to key material or to data to be encrypted, for which both are needed in order to use the fully abstract model.

A previous work related to our approach is [KR06], where it is showed that any Dolev-Yao model of a WS-Security protocol, where the XML encoding is embedded into the datatype, can be simplified into a more abstract protocol, still preserving secrecy and agreement, by abstracting away XML tag encodings and just keeping the contents of XML elements. Another related work is [BCFG07], where WS-Security protocols are verified for security properties. In those works, XML tags are part of the datatype. It turns out that the results being presented here can be reused to generalize in two directions the works of [KR06, BCFG07]. On one hand, the work presented here takes the implementation of encoding functions into account, and is not limited to consider only how they are specified. On the other hand, the results proposed here apply to any encoding scheme, XML and WS-Security being a particular instance of it.

Although some of the results presented in this paper are probably not so surprising, all of them have been formally proved 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.

An important step towards a formally safe refinement process in methods based on code generation has been made. For example, when dealing with marshaling operations, the developer only needs to write and verify the abstract protocol model, and the code generation engine can take care of ensuring that the generated code meets the data flow assumptions made about the refined model. A similar approach can be used to deal with encoding and decoding operations made on data on which cryptographic operations are applied. In this case, more constraints are required on the generated implementation code. By the way, it has been showed that, at least for the important class of XML encodings, the check on the correctness of the encoding and decoding functions is simple enough.

When adopting a model extraction approach, the results presented in this paper can be used to simplify the extracted model. We have showed 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 independent from the adversary, and thus simpler to be checked in isolation than it is checking a full protocol model, which is a concurrent system that includes a detailed model of the encoding and decoding functions and an adversary.

Moreover, sometimes the source code that implements encoding and decoding functions may not be available (e.g. for libraries). In such cases, code extraction is even impossible. The results presented in this paper show what are the requirements on this code, which can be used to derive specifically targeted testing procedures in order to get a good assurance level in these cases too.

Some issues on the topics presented in this paper are still open for future work. In particular, the conditions under which the final transformation back to the original abstract model is safe for other security trace properties or for security properties not defined on traces (e.g. strong secrecy) could be further explored. Furthermore, one could consider verification with computational models or timed models instead of verification with Dolev-Yao models. For example, in [BCF07] it is explicitly stated that being able to safely exclude some parts of protocol implementations would be useful in model extraction, when dealing with a computational model too; it is believable that stricter constraints on the implementation code than the ones presented in this paper will be required, however. After all, a Dolev-Yao model provides a simple but rigorous way to reason about security protocols, but it also provides a limited form of security assurance. It

can uncover and avoid several kinds of logical errors in security protocols, which makes it very useful, but it should be complemented by more low-level risk analysis techniques, in order to get best security assurance. Specifically, protocols formally verified by using Dolev-Yao models can still be affected by lower-level security faults, like for example the ones showed in [BKN02, APW09]. Extending the work presented here to computational models and timed models could then undoubtedly broaden its application scope.

Acknowledgments

We thank the authors of [BFG06] for kindly providing access to the FS2PV tool and support on reproducing their results. We wish also to thank the anonymous reviewers for their helpful comments.

References

[AB05] Martn Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *IEEE Symposium on Security and Privacy*, pages 16–26, 2009.

[BBF+11] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8:1–8:45, 2011.

[BCF07] Karthikeyan Bhargavan, Ricardo Corin, and Cédric Fournet. Crypto-verifying protocol implementations in ML. In *Proceedings of Workshop on Formal and Computational Cryptography*, 2007.

[BCFG07] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Andrew D. Gordon. Secure sessions for web services. *ACM Transactions on Information and System Security*, 10(2):article 8, 2007.

[BFG06] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verified reference implementations of WS-security protocols. In *Proceedings of Web Services and Formal Methods*, pages 88–106, 2006.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *Proceedings of Computer Security Foundations Workshop*, pages 139–152, 2006.

[BKN02] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: provably fixing the SSH binary packet protocol. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, pages 1–11, 2002.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[GHRS05] Joshua Guttman, Jonathan Herzog, John Ramsdell, and Brian Sniffen. Programming cryptographic protocols. In *Trustworthy Global Computing*, pages 116–145, 2005.

[GLP05] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 363–379, 2005.

[HL01] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jür05] Jan Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *Proceedings of Formal Methods and Models for Co-Design*, pages 89–98, 2005.

[KR06] E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. *Electronic Notes in Theoretical Computer Science*, 155:583–603, 2006.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of Computer Security Foundations Workshop*, pages 31–43, 1997.

[NKHBM06] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. OASIS web services security: SOAP message security 1.1 (WS-security 2004), 2006.

[Pir10] Alfredo Pironti. *Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications.* PhD thesis, Politecnico di Torino (Italy), 2010.

[PPS12] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. Formally-based semi-automatic implementation of an open security protocol. *Journal of Systems and Software*, 85:835–849, 2012.

[PS10] Alfredo Pironti and Riccardo Sisto. Provably correct Java implementations of Spi Calculus security protocols specifications. *Computers & Security*, 29:302–314, 2010.

[PSD04] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *Proceedings of International Conference on Advanced Information Networking and Applications*, pages 400–405, 2004.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[Sch05] Gerhard Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theoretical Computer Science*, 336(2-3):403–435, 2005.

[TH04] Benjamin Tobler and Andrew Hutchison. Generating network security protocol implementations from formal specifications. In *Proceedings of Certification and Security in Inter-Organizational E-Services*, Toulouse, France, 2004.

[YL06a] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[YL06b] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.

APPENDIX: Proofs of Theorems

A.1 Proof of Theorem 3.1

The following statement is proved, that implies the theorem: for all traces tr' of SYS^m such that an attack exists in tr', there exists a trace tr'' of $SYS^{m\text{-}noML}$, such that an attack exists in tr'' too. Formally,

$$\forall\ tr' \in traces(SYS^m)\ \cdot\ \neg SPEC(tr') \Rightarrow$$
$$\exists\ tr'' \in traces(SYS^{m\text{-}noML})\ \cdot\ \neg SPEC(tr'')$$

In order carry out the proof, the intermediary $SYS^{m\text{-}advML}$ depicted in figure 6 is used. It is formally defined as

$$SYS^{m\text{-}advML} \triangleq (|||_{A \in Honest} PL_A^m) || ADV\text{-}ML$$

where PL_A^m is communicating on the *send* and *receive* channels. To avoid ambiguities in the remainder of the proof, the ADV-ML process, which is the part inside the dashed box in figure 6, is formally defined as

$$ADV\text{-}ML \triangleq (((|||_{A \in Honest} ML_A^s) \parallel ADV^s(AK_0^m)) \setminus \{|int|\}) \tag{29}$$

where the swapping of the communication channels is made explicit by defining

$$ext = send, receive$$
$$ML_A^s = ML_A[[{}^{int,ext}/_{ext,int}]]$$
$$ADV^s(AK_0^m) = ADV(AK_0^m)[[{}^{int}/_{ext}]]$$

A lemma is now introduced. It is needed to complete the proof of this theorem.

Lemma A.1 *The adversary of $SYS^{m\text{-}advML}$ is a trace refinement of the adversary of $SYS^{m\text{-}noML}$;*

$$ADV(AK_0^{m\text{-}noML}) \sqsubseteq ADV\text{-}ML$$

By lemma A.1 and by refinement properties exposed in [Ros97], it follows that

$$SYS^{m\text{-}noML} \sqsubseteq SYS^{m\text{-}advML}$$

Let us assume that a fault trace tr' exists in $traces(SYS^m)$. Recall that $SYS^{m\text{-}advML}$ is obtained from SYS^m, by injectively renaming communication channels of the processes appearing in SYS^m. In particular, only the *send*, *receive*, *int_send* and *int_receive* channels are renamed by each other, so that they are in fact swapped. Moreover, since every renaming is injective, no non-determinism is introduced in the whole $SYS^{m\text{-}advML}$. So, by assumption (3), and by taking into account that events on private channels do not affect the truth of a security property, the trace $tr''' \in traces(SYS^{m\text{-}advML})$, that is obtained by swapping channels in tr', is a fault trace too. Finally, since $SYS^{m\text{-}noML} \sqsubseteq SYS^{m\text{-}advML}$ holds, tr''' is also a (fault) trace in $SYS^{m\text{-}noML}$. Then, having showed that faults are preserved from SYS^m to $SYS^{m\text{-}noML}$, the theorem is proved.

[Proof of lemma A.1.] In order to carry out the proof, the reachable states of ADV-ML are written explicitly. For this reason, let ML_i be defined as a generic state reachable from $|||_{A \in Honest} ML_A^s$. Moreover, for each state ML_i, let MK_i be the "marshaling layer knowledge", that is the set of all the encoded messages ready to be delivered to the ADV^s component of ADV-ML, but not yet dispatched to it.

Formally, MK_i is a set of messages that can be defined inductively. In the initial state

$$ML_0 = |||_{A \in Honest} ML_A^s$$

we have $MK_0 = \emptyset$. The evolution of MK_i after an event e can be represented by an extension of the $\xrightarrow{}$ state transition relation as follows: $MK_i \xrightarrow{e} MK_j$ means that the occurrence of e in a state where the set of encoded messages ready to be delivered to ADV^s is MK_i leads to a new state where the new set is MK_j. Now, since the events e occurring in ML_i can only take 4 different forms, the relation $\xrightarrow{e}$ on sets of messages

can be defined by enumeration. By the definition of process ML_A in figure 4, and taking into account that ML_A^s has swapped channels, it follows that

$$\begin{array}{lll} MK_i & \xrightarrow{send.A.B.(\text{ATOM } \mathcal{L},(a,M))} & MK_i \cup \{e_A(a,M)\} \\ MK_i & \xrightarrow{receive.A.B.(\text{ATOM } \mathcal{L},(a,M))} & MK_i \\ MK_i & \xrightarrow{int\_send.A.B.e_A(a,M)} & MK_i \setminus \{e_A(a,M)\} \\ MK_i & \xrightarrow{int\_receive.A.B.y} & MK_i \end{array}$$

A generic state of $ADV\text{-}ML$ then takes the form

$$ADV\text{-}ML_i \triangleq (ML_i \parallel ADV^s(AK_i^m)) \setminus \{|int|\}$$

where AK_i^m is the current adversary knowledge. The initial state is

$$ADV\text{-}ML_0 = ADV\text{-}ML$$

i.e.

$$ADV\text{-}ML_0 = (ML_0 \parallel ADV^s(AK_0^m)) \setminus \{|int|\}$$

Finally, the total knowledge MAK_i associated with state $ADV\text{-}ML_i$ is defined as

$$MAK_i \triangleq MK_i \cup AK_i^m$$

By lemma 2.1, in state $ADV\text{-}ML_i$, we have $deds(AK_i^m) \subseteq deds(MAK_i)$.

Let us preliminary prove the following

Lemma A.2

$$ADV\text{-}ML_i \xrightarrow{\tau} ADV\text{-}ML_j \Longrightarrow MAK_i = MAK_j$$

where $\xrightarrow{\tau}$ represents the occurrence of an internal event (i.e. an event not visible from the environment, like for example a hidden event). There are two possible cases: τ comes from an int_send event, or τ comes from an $int\_receive$ event.

case τ comes from event $int\_send.A.B.e_A(a,M)$ By the definition of each process ML_A^s, a corresponding $send.A.B.(\text{ATOM } \mathcal{L},(a,M))$ must have previously occurred. So, by the definition of MK_i,

$$e_A(a,M) \in MK_i$$

and

$$MK_j = MK_i \setminus \{e_A(a,M)\}$$

and, by the definition of ADV^s,

$$AK_j^m = AK_i^m \cup \{e_A(a,M)\}$$

so $MAK_i = MAK_j$.

case τ comes from event $int\_receive.A.B.y$ By the definition of MK_i,

$$MK_j = MK_i$$

and, by the definition of ADV^s,

$$AK_j^m = AK_i^m$$

so $MAK_i = MAK_j$.

□

Now that lemma A.2 is proved, the following property can be proved for each trace tr, which implies lemma A.1:

$$(ADV\text{-}ML_0 \overset{tr}{\Longrightarrow} ADV\text{-}ML_f) \Longrightarrow$$
$$\exists\ AK_f^{m\text{-}noML} \mid (ADV(AK_0^{m\text{-}noML}) \overset{tr}{\Longrightarrow} ADV(AK_f^{m\text{-}noML}) \ \wedge\ deds(MAK_f) \subseteq deds(AK_f^{m\text{-}noML}))$$

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

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

$$ADV\text{-}ML_0 \xrightarrow{\tau}{}^{*} ADV\text{-}ML_f$$

Then, by lemma A.2, we have that

$$MAK_f = MAK_0 = MK_0 \cup AK_0^m = AK_0^m$$

Moreover, $ADV(AK_0^{m\text{-}noML})$ cannot execute internal events, so if we take

$$AK_f^{m\text{-}noML} = AK_0^{m\text{-}noML}$$

then, considering that by definition (6) it follows that

$$AK_0^m \subset AK_0^{m\text{-}noML}$$

we can conclude $deds(MAK_f) \subseteq deds(AK_f^{m\text{-}noML})$.

Induction ($length(tr) = n + 1$) The trace tr is then composed of a subtrace tr' of length n, followed by the $n + 1^{th}$ event. There are three possible cases: the $n + 1^{th}$ event is a $send, receive$ or $leak$ event.

case $n + 1^{th}$ **event is a** *send* **event** By the definitions of processes and by inductive hypotheses

$$ADV\text{-}ML_0 \xRightarrow{tr'} ADV\text{-}ML_i$$

Moreover:

$$ADV\text{-}ML_i \xrightarrow{\tau}{}^{*} \xrightarrow{send.A.B.(\text{ATOM } \mathcal{L},(a,M))} \xrightarrow{\tau}{}^{*} ADV\text{-}ML_f$$

By lemma A.2, MAK_i will remain unchanged for each τ transition before the *send* event, and MAK_f will remain unchanged for each τ transition after the *send* event. Moreover, by definition,

$$MAK_f = MAK_i \cup \{e_A(a, M)\}$$

At the other side, by inductive hypotheses, there exists $AK_i^{m\text{-}noML}$ such that

$$ADV(AK_0^{m\text{-}noML}) \xRightarrow{tr'} ADV(AK_i^{m\text{-}noML})$$

and

$$ADV(AK_i^{m\text{-}noML}) \xrightarrow{send.A.B.(\text{ATOM } \mathcal{L},(a,M))} ADV(AK_f^{m\text{-}noML})$$

where $AK_f^{m\text{-}noML} = AK_i^{m\text{-}noML} \cup \{(\text{ATOM } \mathcal{L}, (a, M))\}$

By inductive hypotheses, $deds(MAK_i) \subseteq deds(AK_i^{m\text{-}noML})$; by definition (5) of $e_A(a, M)$ and by lemma 2.1, it follows that $deds(e_A(a, M)) \subseteq deds((\text{ATOM } \mathcal{L}, (a, M)))$. So

$$deds(MAK_i \cup \{e_A(a, M)\}) \subseteq deds(AK_i^{m\text{-}noML} \cup \{(\text{ATOM } \mathcal{L}, (a, M))\})$$

thus $deds(MAK_f) \subseteq deds(AK_f^{m\text{-}noML})$.

case $n + 1^{th}$ **event is a** *receive* **event** By process definitions and by inductive hypotheses

$$ADV\text{-}ML_0 \xRightarrow{tr'} ADV\text{-}ML_i$$

Moreover:

$$ADV\text{-}ML_i \xrightarrow{\tau}{}^{*} \xrightarrow{receive.A.B.(\text{ATOM } \mathcal{L},(a,d_A(a,y)))} \xrightarrow{\tau}{}^{*} ADV\text{-}ML_f$$

By lemma A.2, MAK_i will remain unchanged for each τ transition before the *receive* event, and MAK_f will remain unchanged for each τ transition after the *receive* event. Moreover

$$MAK_f = MAK_i$$

At the other side, by inductive hypotheses, there exists $AK_i^{m\text{-}noML}$ such that

$$ADV(AK_0^{m\text{-}noML}) \xRightarrow{tr'} ADV(AK_i^{m\text{-}noML})$$

Then, we need to show that

$$ADV(AK_i^{m\text{-}noML}) \xrightarrow{receive.A.B.(\text{Atom } \mathcal{L},(a,d_A(a,y)))} ADV(AK_f^{m\text{-}noML})$$

where $AK_f^{m\text{-}noML} = AK_i^{m\text{-}noML}$, because the adversary knowledge does not change on *receive* events.
Since, by inductive hypotheses, $deds(MAK_i) \subseteq deds(AK_i^{m\text{-}noML})$, and it has been showed that both $MAK_f = MAK_i$ and $AK_f^{m\text{-}noML} = AK_i^{m\text{-}noML}$ hold, it is possible to conclude that

$$deds(MAK_f) \subseteq deds(AK_f^{m\text{-}noML})$$

holds too. In order to complete the proof of this case, it is enough to show that the *receive* event indeed can happen in $ADV(AK_i'')$, that is, in order to send $(\text{Atom } \mathcal{L}, (a, d_A(a, y)))$ on the *receive* channel, $ADV(AK_i^{m\text{-}noML})$ must be able to derive the required message from its knowledge.
If a and y are derivable from $AK_i^{m\text{-}noML}$, then, by definition (5), $d_A(a, y)$ is derivable too, and, by applying the **pairing** rule, $(a, d_A(a, y))$ is derivable too. Message a is derivable from $AK_i^{m\text{-}noML}$ with the **member** rule, since

$$a \in Marshaling \subset AK_0^{m\text{-}noML} \subseteq AK_i^{m\text{-}noML}$$

Message y is derivable from $AK_i^{m\text{-}noML}$ because, since the *receive* event can happen in $ADV\text{-}ML_i$, then

$$y \in deds(MAK_i)$$

and, by inductive hypotheses,

$$deds(MAK_i) \subseteq deds(AK_i^{m\text{-}noML})$$

thus $y \in deds(AK_i^{m\text{-}noML})$.
Finally, if $\text{Atom } \mathcal{L}$ is derivable from $AK_i^{m\text{-}noML}$, then, by applying the **pairing** rule, the required $(\text{Atom } \mathcal{L}, (a, d_A(a, y)))$ message can be obtained. The message $\text{Atom } \mathcal{L}$ is derivable from $AK_i^{m\text{-}noML}$ with the **member** rule, because

$$\text{Atom } \mathcal{L} \in AK_0^{m\text{-}noML} \subseteq AK_i^{m\text{-}noML}$$

case $n + 1^{th}$ **event is a** *leak* **event** By process definitions and by inductive hypotheses

$$ADV\text{-}ML_0 \xRightarrow{tr'} ADV\text{-}ML_i$$

Moreover:

$$ADV\text{-}ML_i \xrightarrow{\tau}{}^{*} \xrightarrow{leak.M} \xrightarrow{\tau}{}^{*} ADV\text{-}ML_f$$

By lemma A.2, MAK_i will remain unchanged for each τ transition before the *leak* event, and MAK_f will remain unchanged for each τ transition after the *leak* event. Moreover, the *leak* event is not engaged by the ML_i process, so $MK_f = MK_i$, and, by the definition of adversary, $S_f = S_i$, so $MAK_f = MAK_i$.
At the other side, by inductive hypotheses, there exists $AK_i^{m\text{-}noML}$ such that

$$ADV(AK_0^{m\text{-}noML}) \xRightarrow{tr'} ADV(AK_i^{m\text{-}noML})$$

and

$$ADV(AK_i^{m\text{-}noML}) \xrightarrow{leak.M} ADV(AK_f^{m\text{-}noML})$$

where $AK_f^{m\text{-}noML} = AK_i^{m\text{-}noML}$. So, by inductive hypotheses, it follows that $deds(MAK_f) \subseteq deds(AK_f^{m\text{-}noML})$. Finally, the *leak.M* event can indeed happen in $ADV(AK_i^{m\text{-}noML})$ because, by hypotheses, the *leak.M* event can happen in $ADV\text{-}ML_i$, which implies $M \in MAK_i$. Since, by inductive hypotheses, $deds(MAK_i) \subseteq deds(AK_i^{m\text{-}noML})$, it follows that $M \in AK_i^{m\text{-}noML}$ too, so the *leak.M* event can happen in $ADV(AK_i^{m\text{-}noML})$.

□

A.2 Proof of Theorem 4.3

As explained in the proof sketch, the proof idea for theorem 4.3 is to refine all abstract protocol logics $P_A^{e\text{-}pm}$ in $SYS^{e\text{-}pm}$ into their refined counterparts P_A^e, one at a time, thus refining the whole $SYS^{e\text{-}pm}$ to SYS^e. Unfortunately, it is not possible to trivially infer this result directly from lemma 4.5. Indeed, it is true that, for any processes A and R, and for any context $C[\ ]$,

$$A \sqsubseteq R \Rightarrow C[A] \sqsubseteq C[R]$$

So, let us consider that lemma 4.5 holds for honest actor H. Then, setting the context to

$$C[X] = (|||_{A \in Honest \setminus \{H\}} P_A^{e\text{-}pm} \parallel X) \setminus comm$$

leads to a process

$$(|||_{A \in Honest \setminus \{H\}} P_A^{e\text{-}pm} \parallel ((P_H^{e\text{-}pm} || ADV(AK_0^{e\text{-}pm})) \setminus comm_H)) \setminus comm$$

which is not trivially proved to have all and the same traces of $SYS^{e\text{-}pm}$.

The proof steps of trace refinement reported here use two CSP operator properties that are introduced now.

The first property states that, for any processes P, Q, R, if $\alpha P \cap \alpha Q = \emptyset$, then

$$P \parallel (Q \parallel R) = (P \ ||| \ Q) \parallel R$$

This property follows by the definition of the parallel and interleaved operators.

Informally, this property means that if P and Q cannot communicate directly (because, being the intersection of their alphabets empty, they cannot synchronize on any event), then, on one hand, P communicating with $Q \parallel R$, is actually only communicating with R; on the other hand, Q is only communicating with R, and never with P. Thus, R acts as a proxy between P and Q, while the latter two processes can execute in interleaving.

The second property states that, for any processes P, Q, R, if $\alpha P \cap \alpha Q = \emptyset$, then

$$(P \parallel (Q \parallel R)) \setminus (\alpha Q \cap \alpha R) \cup (\alpha P \cap \alpha R) = (P \parallel ((Q \parallel R) \setminus (\alpha Q \cap \alpha R))) \setminus (\alpha P \cap \alpha R)$$

Informally, this property means that, since P and Q never communicate directly, and R is their proxy, from P's view it is irrelevant whether communication between Q and R is observable or not, thus allowing to put P in parallel either with $Q \parallel R$ or with $(Q \parallel R) \setminus (\alpha Q \cap \alpha R)$. However, from an observer point of view, communication between Q and R must always be hidden, so if it is not hidden with the $(Q \parallel R) \setminus (\alpha Q \cap \alpha R)$ process, it must be hidden at the top level process.

Indeed, in the used Dolev-Yao approach, any pair of protocol actors has disjoint alphabets, because the adversary is the only proxy between any pair of actors, and they can never communicate directly. Thus, these properties directly apply when P and Q are two processes representing interleaved actors and R is the adversary.

Trace refinement is proved by induction over the number of protocol logics that are step by step refined in $SYS^{e\text{-}pm}$.

base It will be proved that $SYS^{e\text{-}pm}$, where all protocol logics are abstract, is refined by a process where one protocol logic is refined, that is

$$\forall\ X \in Honest \cdot$$
$$(|||_{A \in Honest} P_A^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm})) \setminus comm \sqsubseteq$$
$$\left((|||_{A \in Honest \setminus \{X\}} P_A^{e\text{-}pm} ||| P_X^e) \parallel ADV(AK_0^e)\right) \setminus comm$$

The proof steps are:

$$
\begin{aligned}
& (|||_{A \in Honest} P_A^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm})) \setminus comm \\
= \;& \big((|||_{A \in Honest \setminus \{X\}} P_A^{e\text{-}pm} \;|||\; P_X^{e\text{-}pm}) \parallel ADV(AK_0^{e\text{-}pm})\big) \setminus comm \\
= \;& \langle \text{by letting } Others = |||_{A \in Honest \setminus \{X\}} P_A^{e\text{-}pm} \rangle \\
& ((Others \;|||\; P_X^{e\text{-}pm}) \parallel ADV(AK_0^{e\text{-}pm})) \setminus comm \\
= \;& \langle \text{by property of } ||| \text{ and } \parallel, \text{ and by } \alpha(Others) \cap \alpha P_X^{e\text{-}pm} = \emptyset \rangle \\
& (Others \parallel (P_X^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm}))) \setminus comm \\
= \;& \langle \text{by hiding property; letting } comm^- = comm \setminus comm_X \rangle \\
& (Others \parallel (P_X^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm}) \setminus comm_X)) \setminus comm^- \\
\sqsubseteq \;& \langle \text{consider the context } Others \text{ surrounding the process refined in lemma 4.5} \rangle \\
& (Others \parallel ((P_X^e \parallel ADV(AK_0^e)) \setminus comm_X)) \setminus comm^- \\
= \;& \langle \text{by hiding property} \rangle \\
& (Others \parallel (P_X^e \parallel ADV(AK_0^e))) \setminus comm \\
= \;& \langle \text{by property of } ||| \text{ and } \parallel, \text{ and by } \alpha(Others) \cap \alpha P_X^e = \emptyset \rangle \\
= \;& ((Others \;|||\; P_X^e) \parallel ADV(AK_0^e)) \setminus comm \\
= \;& \langle \text{by definition of } Others \rangle \\
= \;& \big((|||_{A \in Honest \setminus \{X\}} P_A^{e\text{-}pm} \;|||\; P_X^e) \parallel ADV(AK_0^e)\big) \setminus comm
\end{aligned}
$$

induction It will be showed that if the refinement relation holds when n actors have been refined, then it keeps holding when the $n+1^{th}$ actor is refined too. Let Ref be the set of already refined actors, then $\forall\ X \in Honest \setminus Ref$

$$
\begin{aligned}
& \big((|||_{A \in Honest \setminus Ref} P_A^{e\text{-}pm} \;|||\; |||_{B \in Ref} P_B^e) \parallel ADV(AK_0^e)\big) \setminus comm \\
= \;& \big((|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^{e\text{-}pm} \;|||\; |||_{B \in Ref} P_B^e \;|||\; P_X^{e\text{-}pm}) \\
& \quad \parallel ADV(AK_0^e)) \setminus comm \\
\sqsubseteq \;& \langle \text{by setting } Others = |||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^{e\text{-}pm} \;|||\; |||_{B \in Ref} P_B^e; \\
& \quad \text{by using the same steps as in base case} \rangle \\
& \big((|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^{e\text{-}pm} \;|||\; |||_{B \in Ref} P_B^e \;|||\; P_X^e) \\
& \quad \parallel ADV(AK_0^e)) \setminus comm \\
= \;& \big((|||_{A \in Honest \setminus (Ref \cup \{X\})} P_A^{e\text{-}pm} \;|||\; |||_{B \in Ref \cup \{X\}} P_B^e) \\
& \quad \parallel ADV(AK_0^e)) \setminus comm
\end{aligned}
$$

In the inductive step, the adversary knowledge in the abstract system (the one having n refined actors) is set to AK_0^e, and not $AK_0^{e\text{-}pm}$. Indeed, after the first refinement step made in the base case, the adversary knowledge is "restricted" to AK_0^e, the refined one. This is not an issue during the inductive step, because condition (21) in lemma 4.5 requires, in the abstract system, that the adversary knowledge is a *weak* superset of the adversary knowledge in the refined system.

A.3 Proof of Lemma 4.5

Lemma 4.5 states that, under the same assumptions used by theorem 4.3, if communication channels are hidden, then one abstract protocol logic $P_A^{e\text{-}pm}$ acting with the adversary, is refined by its more detailed protocol logic P_A^e, that explicitly models the decoding process, acting with the adversary. A weak simulation relation between the abstract process

$$P_A^{e\text{-}pm} \parallel ADV(AK_0^{e\text{-}pm}) = f(PL_A^e) \parallel ADV(AK_0^{e\text{-}pm})$$

and the refined process

$$P_A^e \parallel ADV(AK_0^e) = ((PL_A^e \parallel DEC_A) \setminus priv_A) \parallel ADV(AK_0^e)$$

is first proved, which is then showed to imply the desired trace refinement. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a refined (also called *concrete* in other works) process, but each process is still allowed to perform any internal step in between two external states. More details about the weak refinement used here can be found, for example, in [Sch05].

An external state of the refined protocol logic of actor A is defined as a generic state $PL_{A_i}^e$ of the process PL_A^e, such that $PL_{A_i}^e$ does not begin with an event in the $priv_A$ set; that is, $PL_{A_i}^e$ does not take the form $ev \to Q$, with $ev \in priv_A$ (by construction of PL_A^e, if an event $ev \in priv_A$ can occur, it is the only event that can occur). Accordingly, an external state of the refined process takes the form

$$SYS_{A_i}^e = ((PL_{A_i}^e \parallel DEC_A) \setminus priv_A) \parallel ADV(AK_i^e)$$

because, by construction of PL_A^e and DEC_A, the latter process will always be in its initial state, which is DEC_A, when the former is in an external state.

In the same way, an external state $P_{A_i}^{e\text{-}pm}$ of the abstract protocol logic is defined as a generic state of $f(PL_A^e)$ that is not ready to perform an event that is in $priv_A$. With this definition it turns out that *any* state of $f(PL_A^e)$ is an external state, since all events in $priv_A$ have been removed by $f(\cdot)$. Then, any external state of the abstract process takes the form

$$SYS_{A_i}^{e\text{-}pm} = P_{A_i}^{e\text{-}pm} \parallel ADV(AK_i^{e\text{-}pm})$$

The relation R that binds external states of the abstract process to external states of the refined one is formally defined as

$$R(SYS_{A_i}^{e\text{-}pm}, SYS_{A_i}^e) \Leftrightarrow P_{A_i}^{e\text{-}pm} = f(PL_{A_i}^e) \ \wedge \ AK_i^{e\text{-}pm} \supseteq AK_i^e$$

Relation R holds on the initial states of the abstract and refined processes. Indeed, PL_A^e must be an external state, because, by construction, it cannot begin with an event in $priv_A$. Moreover, by hypothesis (21), $AK_0^{e\text{-}pm} \supseteq AK_0^e$ holds.

Let us denote with P/ev ("P after ev") the state of process P after it has engaged the event ev; if P/ev is not applicable, that is, P cannot engage the event ev, then $P/ev = P$. With this definition, the operator / ("after") is distributive with respect to the operator $\parallel$, which synchronizes on the intersection of the alphabets, that is

$$(P \parallel Q)/ev = P/ev \parallel Q/ev$$

The after operator is then overloaded to sequences of events in the obvious way.

In order to show that the weak simulation relation holds, it is enough to show that, for any external states $SYS_{A_i}^e, SYS_{A_i}^{e\text{-}pm}$, and event sequence ev_s:

$$\begin{array}{rcl} R(SYS_{A_i}^{e\text{-}pm}, SYS_{A_i}^e) & \wedge & SYS_{A_i}^e \stackrel{ev_s}{\to} SYS_{A_i}^e/ev_s \\ & \Longrightarrow & \\ SYS_{A_i}^{e\text{-}pm} \stackrel{ev_s^*}{\to} SYS_{A_i}^{e\text{-}pm}/ev_s^* & \wedge & R(SYS_{A_i}^{e\text{-}pm}/ev_s^*, SYS_{A_i}^e/ev_s,) \end{array} \tag{30}$$

where $\stackrel{ev_s}{\to}$ denotes the concatenation of state transitions for all events in ev_s, and ev_s^* is the sequence of events obtained by stripping all τ events from ev_s.

By the definition of adversary,

$$ADV(AK_i^{e\text{-}pm})/ev_s^* = ADV(AK_j^{e\text{-}pm})$$

where $AK_j^{e\text{-}pm}$ is the new adversary knowledge reached after ev_s^*, and, by the distribution property of the after operator

$$\begin{aligned} SYS_{A_i}^{e\text{-}pm}/ev_s^* &= (f(PL_{A_i}^e) \parallel ADV(AK_i^{e\text{-}pm}))/ev_s^* \\ &= f(PL_{A_i}^e)/ev_s^* \parallel ADV(AK_i^{e\text{-}pm})/ev_s^* \\ &= f(PL_{A_i}^e)/ev_s^* \parallel ADV(AK_j^{e\text{-}pm}) \end{aligned}$$

The same reasoning applies in the refined model.

Now all the possible event sequences ev_s that can lead to one of the next external states must be considered. ev_s cannot start with a τ step, because it starts from an external state. Moreover it is enough to prove (30) for event sequences ev_s such that no proper subsequence of ev_s leads to an external state because then the most general case descends by induction. Then, if ev_s starts with a $claimSecret, running, finished, leak$ or $send$ event, it is possible to consider only the case when it is composed of just one event, because after the first event an external state is reached.

Let us first consider these cases, where $ev_s = \langle ev \rangle = ev_s^*$ and let us show that (30) holds.

case $ev = claimSecret.A.B.M$ By hypothesis this event can happen in the refined system. If we let $PL_{A_j}^e = PL_{A_i}^e/ev$, we have

$$\begin{aligned} ((PL_{A_i}^e \parallel DEC_A) \setminus priv_A)/ev &= (PL_{A_i}^e/ev \parallel DEC_A/ev) \setminus priv_A \\ &= (PL_{A_j}^e \parallel DEC_A) \setminus priv_A \end{aligned}$$

because $ev \notin priv_A$ and DEC_A is only engaged in the events in $priv_A$.

Since ev can occur in $PL_{A_i}^e$, and $PL_{A_i}^e \xrightarrow{ev} PL_{A_j}^e$, by the properties of $f(\cdot)$, it can be concluded that

$$f(PL_{A_i}^e) \xrightarrow{ev} f(PL_{A_j}^e)$$

that is, ev can also happen in the abstract system, and the states of the abstract and refined protocol logics after ev are bound by $f(\cdot)$.

Moreover, after event ev, adversary knowledges remain unchanged in both systems, so $AK_j^{e\text{-}pm} \supseteq AK_j^e$ holds, and it can be concluded that R holds after ev.

The same reasoning also applies for the $running$ and $finished$ events.

case $ev = leak.M$ Since this event is engaged by the adversary process, and not by the protocol logic, if ev can happen in the refined system, then M must be in the adversary knowledge, that is $M \in AK_i^e$. Consequently, by the assumption $AK_i^{e\text{-}pm} \supseteq AK_i^e$, it follows that $M \in AK_i^{e\text{-}pm}$ too, and ev can happen in the abstract system too. Moreover, the state of the protocol logic and the adversary knowledge remain unchanged after this event in both refined and abstract processes, so relation R still holds after it.

case $ev = send.A.B.M$ If this event can happen in the refined system, it can also happen in the abstract system, because, as with the $claimSecret$ event,

$$PL_{A_i}^e \xrightarrow{ev} PL_{A_j}^e$$

implies

$$f(PL_{A_i}^e) \xrightarrow{ev} f(PL_{A_j}^e)$$

While the reasoning about the protocol logic states is the same as explained in the $claimSecret.A.B.M$ case, the reasoning about adversary knowledges changes. After the event ev happens in the refined system, we have

$$AK_j^e = AK_i^e \cup \{M\}$$

and similarly, in the abstract system,

$$AK_j^{e\text{-}pm} = AK_i^{e\text{-}pm} \cup \{M\}$$

Since, by hypothesis, $AK_i^{e\text{-}pm} \supseteq AK_i^e$ holds, then $AK_j^{e\text{-}pm} \supseteq AK_j^e$ holds too.

The cases that remain to be considered are when ev_s starts with a $receive.B.A.M$ event. This event is followed by the τ steps coming from the sequence of the pairs of $priv\_send_A.(y,a) \rightarrow priv\_receive_A.N$ prefixes that have been added during refinement after the *receive* event. An external state is reached only after all these τ steps have been completed and, after a *receive* event, the protocol logic is obliged by construction to execute the sequence of private actions before any further external action. Then, there is a single external state that can be reached by the protocol logic after a *receive* event. Moreover, being the protocol logic synchronized with the adversary, the latter can execute only internal actions, i.e. *leak* actions, while the protocol logic is executing the internal steps that follow a *receive* event. These *leak* events do not change the global state. In conclusion, the only event sequences that start with a *receive* event and that lead to an external state are those that, after the *receive* event, have a sequence of τ events, corresponding to all the private actions that follow the *receive* event in the protocol logic, with interleaved *leak* events generated by the adversary. All these sequences lead to a single external state. If the refined protocol logic cannot execute one of the private actions that follow a *receive* event, the protocol logic gets stuck and no external state is ever reached. So, the only case that must be considered in order to check (30) is when all the private actions are executed.

By using the distributive property of the after operator, again we analyze protocol logics first, and then adversary knowledges.

In the refined system we have

$$((PL^e_{A_i} \parallel DEC_A) \setminus priv_A)/ev_s = (PL^e_{A_j} \parallel DEC_A) \setminus priv_A$$

The state of DEC_A does not change because after each pair of $priv\_send_A$ and $priv\_receive_A$ events, it returns to its initial state, and we are under the hypothesis that all private events happen.

Let

$$receive.B.A.T \rightarrow priv\_send_A.(y_1,a_1) \rightarrow priv\_receive_A.T_1 \rightarrow \ldots \rightarrow priv\_send_A.(y_n,a_n) \rightarrow priv\_receive_A.T_n \rightarrow PL^e_{A_j}$$

be the sequence of prefixes that are executed in $PL^e_{A_i}$ when ev_s occurs.

By the definition of DEC_A it follows that the data exchanged between the protocol logic and DEC_A in each pair of events $priv\_send_A.(M,a), priv\_receive_A.N$ must satisfy equations $N = d_A(a,M)$ and $N \neq \text{ATOM } \mathcal{E}$. Then, if all private events can occur in the refined system, the previous $receive.B.A.T$ event must have bound variables in T in such a way that $T_i = d_A(a_i, y_i)$ for all $1 \leq i \leq n$. But, by (20), this also implies that

$$e_A(a_i, T_i) = e_A(a_i, d_A(a_i, y_i)) = y_i$$

By the definition of $f(\cdot)$, we have that the abstract protocol logic in state $f(PL^e_{A_i})$ is ready to execute

$$receive.B.A.T^* \rightarrow f(PL^e_{A_j})$$

where $T^* = T\left[{}^{e_A(a_1,T_1)}/_{y_1}\right] \ldots \left[{}^{e_A(a_n,T_n)}/_{y_n}\right]$, i.e. where T^* is T with the same binding of variables that occurs in the concrete system when all private events occur.

Then, if in the refined system an event $receive.B.A.M$ followed by all the subsequent private events can occur, the same event can occur in the abstract system too, and

$$f(PL^e_{A_i}) \overset{receive.B.A.M}{\rightarrow} f(PL^e_{A_j})$$

With respect to the adversary, since the event $receive.B.A.M$ can happen in the refined system, it follows that $M \in AK^e_i$. Since $AK^{e\text{-}pm}_i \supseteq AK^e_i$, it follows that the event can happen in the abstract system too, because $M \in AK^{e\text{-}pm}_i$. Moreover, since after a *receive* event adversary knowledges remain unchanged, it also follows that $AK^{e\text{-}pm}_j \supseteq AK^e_j$.

In order to complete the proof of the simulation relation, we have to show that *leak* events interleaved in ev_s can happen in the abstract system too. This descends from the fact that neither the *receive* event nor the τ events change the adversary knowledge. Then, the occurrence of a $leak.N$ event in ev_s implies that $N \in AK^e_i$, which, in turn, by the hypothesis $AK^e_i \subseteq AK^{e\text{-}pm}_i$, implies $N \in AK^{e\text{-}pm}_i$, which finally means that $leak.N$ can be executed by the adversary in the abstract system too.

The weak simulation relation that has been proved implies that any trace of the refined system $P^e_A \parallel ADV(AK^e_0)$ that leads to an external state is also a trace of the abstract system $P^{e\text{-}pm}_A \parallel ADV(AK^{e\text{-}pm}_0)$. However it is still possible that a trace that leads to an internal state in the refined system is not a trace of the abstract system. By the cases that have just been analyzed, it can be realized that a trace of the refined

system can lead to an internal state only when the last action performed by the protocol logic is a *receive*. Moreover, in this trace, the last *receive* event can be followed only by *leak* events. The longest prefix of this trace that leads to an external state is the one that is obtained by removing the last *receive* event and the subsequent *leak* events, and the proof previously given ensures that this is also a trace of the abstract system. Moreover, since the *receive* event does not change the adversary knowledge, we can conclude that any *leak* event following the *receive* event in the refined system could also occur, both in the refined and in the abstract systems, before the *receive* event. Then, we have that even when a trace tr of the refined system is not a trace of the abstract system, the abstract system can still execute a trace tr^* that differs from tr only in the last *receive* event. This lets us conclude that lemma 4.5, which involves processes with hidden *send* and *receive* events, holds. □