# Towards a Mechanization of Cryptographic Protocol Verification

Dominique Bolignano

Dyade, B.P.105 78153 Le Chesnay Cedex France, Dominique.Bolignano@dyade.fr

**Abstract.** We revisit the approach defined in [2] for the formal verification of cryptographic protocols so as to allow for some mechanization in the verification process. In the original approach verification uses theorem proving. Here we show that for a wide range of practical situations and properties it is possible to perform the verification on a finite and safe abstract model.

## 1 Introduction

Formal verification of cryptographic protocols has recently received increased consideration due to the importance of cryptographic protocols in the design of new security or electronic commerce architectures. Many proof-based verification techniques have been proposed (see [2] for a discussion of this issue) to perform systematic analysis of large protocols. Model-checking based techniques have recently been applied [11, 7] to the verification of such protocols. Verification is performed on a finite model that corresponds to an abstraction of the initial specification. The verification is thus automatic. But the proof that such abstractions are safe and do not compromise the generality and accuracy of the verification process has not yet been formalized in the case of cryptographic protocols. In the case of electronic commerce protocols, for which the coherence of data (e.g. price, order or payment information, etc.) is critical, finding a safe abstraction is a particularly crucial issue. In this paper we propose a safe abstraction that can be incorporated into the framework proposed in [2] and further extended in [3]. Similar abstractions based on abstract interpretation techniques have been developed for the verification of temporal properties expressed using various branching-time temporal logics (e.g. [6, 4, 10, 5, 8, 9, 12]). Here we transpose some of the results of [4, 9, 12] to the verification of security properties. We also automate the construction of the abstract model and the translation of security properties into abstract ones for a large class of practical situations. As opposed to other uses of abstraction which typically guarantee the preservation of a whole logic or of a whole class of properties, here a specific abstraction function is selected for each given property and is thus only to guarantee the property at hand. The requirements are consequently much less demanding and the model reduction can be much more important. The proposed approach is currently being applied for the verification of large electronic commerce protocols.

## 2 Basics

Encryption is the transformation of data into a form unreadable by anyone without a secret decryption key. Decryption is the inverse function, which recreates the original data in its form prior to encryption. A cryptographic key system is said to be symmetric if, and only if, the same key can be used for both encryption and decryption. A cryptographic key system is said to be asymmetric when different keys are used for encryption and decryption. In this latter case, one of the key is only known by a particular principal and is known as the private key of this principal, whereas the other one is not confidential and is known as the public key of this principal. For illustration purposes we use a very simple two message key distribution protocol:

$$\boxed{(1)\ A \rightarrow S : (A, B)\ (2)\ S \rightarrow A : (K_B, B)_{K_S^{-1}}}$$

This protocol description can be read as follows: (1) $A$ sends a message to $S$ to tell him that he is $A$ and wants to get $B$'s public key $K_B$; no encryption is used; (2) $S$ replies to the request by sending $A$ $B$'s public key $K_B$; this message is encrypted with the $S$'s private key $K_S^{-1}$ which $S$ is the only one to know and which thus authenticates the producer (this kind of encryption is thus called a signature). Following the approach of [2] we first have to identify the different principals involved. Principals receive messages at one end and emit other messages at another end. Some principals will be considered to be "trustable" (i.e. to work according to their role in the protocol) and some not. Communications media are typically considered to be non-trustable, because messages can usually be intercepted, replayed, removed, or created by intruders. We will consider that this is the case in the following discussion. The set of untrustable principals is modelled as a single (black box) agent which is called the "external world" or, more concisely, the intruder. The intruder is modelled as a principal that may know some data initially and that will store and try to decrypt all data passed to him and thus in particular all information circulating on the communications media. The intruder will also be able to encrypt data to create new messages that will be sent to mislead other principals. But the intruder will be able to decrypt and encrypt data only with keys he knows. This modeling will in particular allow us to determine at any time which data are potentially known to the intruder under the chosen "trustability" hypothesis. The same protocol can be studied in terms of many different hypotheses. According to [2], the knowledge of the intruder is formalized as a set of data components. Data components range over domain $C$ and sets of data components over domain $S$. Data components can be:

- basic data, which may be (1) cryptographic keys which take their values in domain $KA$ (for asymmetric keys) or $KS$ (for symmetric ones), (2) other basic data which will take their values in domain $D$;
- data obtained by composition (1) using the pair operator which takes some data $c_1$ and some $c_2$ and returns the pair $(c_1, c_2)$, (2) or by encryption of some data $c$ using key $k$ which is noted $c_k$.

Messages that are exchanged over communication media are of type $C$. The domains $S$ and $C$ are formalized as:

$$\boxed{\begin{array}{c} C = C_K|(C,C)|B \\ B = K|D \\ K = KA|KS|K^{-1} \end{array}}$$
*figure (1)*

$$\boxed{S = C \cup S|\emptyset}$$
*figure (2)*

$C$ is in fact defined modulo (i.e. quotiented by) the two axioms $\forall k.k \in KS \Rightarrow k^{-1} = k$, and $\forall k.k \in KA \Rightarrow (k^{-1})^{-1} = k$. Similarly $\cup$ is an ACUI operator with neutral $\emptyset$ (i.e. associative, commutative, unitary and idempotent). It is used to describe "flat" sets. The pair operator is used to represent reversible constructors such as the sequence, set, or aggregate constructors. The fact that a given data component $c$ can be derived from the intruder knowledge $s$ is formalized and axiomatized in [2] and is noted: $c$ *known_in* $s$. In the sequel, we will adopt the following conventions: variables $s, s', s'', ..., s_1, s_2, ...$ take their values in $S$ by default; variables $k, k', k'', ..., k_1, k_2, ...$ take their values in $K$.

## 3 Formalizing the Protocol

We then need to formally specify the protocol itself. This specification consists in the description of the role of each trustable agent. The formal specification of the protocol consists of a set of atomic actions. The sending and reception of a message are not synchronous. Consequently the transmission of a message is considered as two atomic actions, one for sending and one for receiving. More precisely, the formalization is based on the chemical reaction paradigm [1]: a system is described as a set of atomic actions which may be applied repeatedly, in any order and whenever their pre-condition holds. Our modeling of the key distribution protocol will thus distinguish 4 different kinds of atomic actions. These actions will be identified using the labels drawn from $\mathcal{A} = \{1_A, 1_S, 2_S, 2_A\}$. Each of the 4 labels $n_X$ of $\mathcal{A}$ stands for one action: principal $X$ sends or receives message $n$. The system is defined as a pair $(s_0, r)$ where $s_0$ is the initial global state, and $r$ is a relation binding the global state before applying an action to the global state after applying the action. The relation $r$ is defined using a predicate or logic formula $p$, defined on $(\mathcal{S} \times (\mathcal{A} \times C) \times \mathcal{S})$ where the domain for global states $\mathcal{S}$ is defined as the Cartesian product, $\mathcal{S}_A \times \mathcal{S}_S \times \mathcal{S}_I$, of local state domains, i.e. $\mathcal{S}_A$ and $\mathcal{S}_S$ for the two trustable principals, $A$ and $S$, and $\mathcal{S}_I$ for the intruder. By definition $p(s, (l, m), s')$ is true if and only if the global state $s$ is modified into $s'$ upon firing the action labelled $l$ for sending or receiving of message $m$. The set $\mathcal{S}_I$ is the domain $S$ of data components defined in the previous section. Intuitively the state of the intruder is the set of data components that have been listened to on the communication line and that the intruder may use to build new messages. The state of a trustable principal is defined as an aggregate or a tuple describing the value of each local state variable. We will use tuples to simplify the presentation. The local state of the key server $S$ is a triple containing a key directory mapping principal identifiers to public keys, the value of the last principal for which the public key was requested when relevant and the value of the program counter. The third state variable is useful in the case where control

constraints have to be specified. The local-state of $A$ is a pair containing the directory of known keys and the local program counter. The directory held by $A$ is empty initially and is updated each time a new association is received from $S$. The directory held by $S$ is never changed. For more conciseness in the sequel, we drop the program counter information from the local state of $A$ and from the local state of $S$. The formula $p$ is thus expressed as the disjunction of 4 sub-formulae, i.e. one for each action:

$$
\begin{aligned}
&p\,((d_A,(d_S,x),s_I),(l,m),(d'_A,(d'_S,x'),s'_I)) = \\
&\quad (l = 1_A \wedge m = (A,id) \wedge s'_I = s_I \cup m \wedge d_A = d'_A \wedge d_S = d'_S) \vee \\
&\quad (l = 1_S \wedge m = (\_,x') \wedge m\ \mathit{known\_in}\ s_I \wedge s'_I = s_I \wedge d_A = d'_A \wedge d_S = d'_S) \vee \\
&\quad (l = 2_S \wedge m = (k,x)_{K_S^{-1}} \wedge s'_I = s_I \cup m \wedge (x,k) \in d_S \wedge d_A = d'_A \wedge d_S = d'_S) \vee \\
&\quad (l = 2_A \wedge m = (k,id)_{K_S^{-1}} \wedge m\ \mathit{known\_in}\ s_I \wedge s'_I = s_I \wedge d_A = d'_A \cup (id,k) \wedge d_S = d'_S)
\end{aligned}
$$

The first action (i.e. $1_A$) describes $A$ sending a pair composed of the identification of $A$ and of the identification of the principal $id$ for which the public key is requested. Each sending of a message $m$ increases the knowledge of the intruder, i.e. $s'_I = s_I \cup m$. The value of $id$ is not constrained in any way. This allows $A$ to request any public key he wishes. The second action (i.e. $1_S$) describes $S$ receiving a pair of data. This pair can be the pair just sent by $A$ or any pair of data known by the intruder. The second situation is only meaningful if this can go undetected by $A$: here there is no particular checking other than on the form of the message. The third action (i.e. $2s$) describes $S$ sending a pair composed of the public key of $d$ and of the identifier $d$ stored previously[1]. Receiving a message $m$ does not change the state of the intruder (i.e. $s'_I = s_I$), but the message should be deducible from the knowledge of the intruder (i.e. $m$ *known_in* $s_I$). The fourth action (i.e. $2_A$) implicitly specifies that the received message is to be signed using $K_s^{-1}$.[2]

## 4 Proving security properties

Most security properties are safety properties[3]. They mainly rely on the fact that the intruder does not know some private data or is not able to construct the expected message. This is in both cases formalized as an invariant property, $\neg(c\ \mathit{known\_in}\ s_I)$, where $c$ stands for the private data in the first case and is the message to construct in the second, and where $s_I$ is the data collected by the intruder. Confidentiality properties which are the simplest security properties, correspond to the situation where $c$ is either a key or a basic

[1] As we have decided to represent data constructors such as the sequence, set, or aggregate constructors using the pair operators, sets and set operators are supposed to be coded in a Lisp-like manner.

[2] Because we have chosen not to store the value of the chosen $id$ in local state of $A$, there is no possibility here for $A$ to check that the key he receives is the key he requested.

[3] The only liveness property is denial of service, which current cryptographic protocols do not guarantee.

data (e.g. a nonce, a credit card number, etc.). As an example, $K_S^{-1}$ should remain unknown to the intruder. This is written $\neg(K_S^{-1}\ known\_in\ s_I)$. But some security properties cannot be written so as to fit into the general form above. As an illustration we will use in the sequel two representative invariant properties drawn from [2]. The first one will be referred to as invariant (1): $\forall\ k,\ x.\ (k,x) \in d_A \Rightarrow (k,x) \in d_S$, i.e. the directory of $A$ should always be coherent with the master directory held by $S$. The second one will be referred to as invariant (2): $\forall k, x.(k,x)_{K_S^{-1}}\ known\_in\ s_I \Rightarrow (x,k) \in d_S$ i.e. any data of the form $(k,x)_{K_S^{-1}}$ that the intruder can replay or produce corresponds to a valid identifier-key association.

## 5 Using a finite state machine

One of the keys to mechanization is to transform a system model into a model that only uses a finite number of keys and basic data values: these keys and basic data are defined as part of a finite subset $B_0$ of $B$. The corresponding subsets of $C$ and $S$ will be noted $C_0$ and $S_0$. The transformation is defined using a function $h : B \rightarrow B_0$ which will intuitively associate each element of $B$ to one of its representative in $B_0$. We then define the homomorphic extension $\hat{h} : S \rightarrow S_0$ of $h$ (i.e. $\hat{h}((c_1,c_2)) = (\hat{h}(c_1),\hat{h}(c_2))$, $\hat{h}(c_k) = \hat{h}(c)_{h(k)}$, etc.).

Given a model $M = (s_0, r)$, let us consider a finite abstract model $M_a$ such that for any finite run $s_0 \stackrel{(l_1,m_1)}{\longrightarrow} s_1 ... \stackrel{(l_k,m_k)}{\longrightarrow} s_k$ of $M$ then $\hat{h}(s_0) \stackrel{(l_1,\hat{h}(m_1))}{\longrightarrow} \hat{h}(s_1) ... \stackrel{(l_k,\hat{h}(m_k))}{\longrightarrow} \hat{h}(s_k)$ is a run of $M_a$: in other words, $M_a = (\hat{h}(s_0), r_a)$, with $r_a$ such that $(s,(l,m),s') \in r \Rightarrow (\hat{h}(s),(l,\hat{h}(m)),\hat{h}(s')) \in r_a$. Let us note $\mathcal{R}(M)$ (resp. $\mathcal{R}(M_a)$) the set of reachable states of a transition system $M$ (resp. $M_a$). By construction $\hat{h}(\mathcal{R}(M)) \subseteq \mathcal{R}(\hat{h}(M_a))$.

Thus in order to prove that an invariant property $inv$ holds on $M = (s_0, r)$ it is sufficient, (a) to provide $r_a$ such that $(s,(l,m),s') \in r \Rightarrow (\hat{h}(s),(l,\hat{h}(m)),\hat{h}(s')) \in r_a$, (b) to provide an invariant property $inv_a$ on $M_a = (\hat{h}(s_0), r_a)$ such that $\forall s.inv_a(\hat{h}(s)) \Rightarrow inv(s)$, and (c) to check $inv_a$ on $M_a = (\hat{h}(s_0), r_a)$. This can be seen as a direct reformulation in our framework of results presented in [9, 12][4] and based on ideas and theoretical results presented in [4]. The steps (a) and (b) generate proof obligations that should be discharged using formal provers, whereas step (c) can be performed using model checking techniques. This was already the case in [9]. The benefit of this approach comes from the fact that both kinds of proof obligations are much simpler to perform than the proof of invariant $inv$ for the initial concrete model $M$.

But in order to perform step (c) we need to be able to perform the checking of $inv_a$ automatically on each reachable state of $M_a$. The problem here comes from the fact that even when a limited number of keys and of basic data components is used, the computations that the intruder may perform (or the data that he can generate) are unbounded: e.g. starting from $k$ the intruder can generate $k_k$,

[4] In [9, 12] this was proved for the AG operator of CTL.

$k_{k_k}$, etc. In this section we thus provide a decision procedure for the *known_in* predicate in the case where the parameters of *known_in* are explicit (described by extension using a variable free expression). The five basic operations that an intruder may use in order to exploit data were defined in [2] and referred to as $\gamma$, $\gamma'$, $\pi$, $\pi'$ and $\xi$ operations: $\gamma$ for the encryption of a known data component using a known key, $\gamma'$ for the decryption of a known data component using a known key, $\pi$ for the pairing of two known data components, $\pi'$ for the decomposition of a pair (i.e. obtaining the first or second projection); and $\xi$ for data extraction. Each action was formalized as a state transformation relation:

$$
\begin{aligned}
s \xrightarrow{\gamma} s' &\stackrel{def}{=} \{(s,s') | (\exists c, k \,.\, k \cup c \subseteq s \wedge s' = s \cup (c)_k\} \\
s \xrightarrow{\gamma'} s' &\stackrel{def}{=} \{(s,s') | (\exists c, k \,.\, k^{-1} \cup (c)_k \subseteq s \wedge\ s' = s \cup c\} \\
s \xrightarrow{\pi} s' &\stackrel{def}{=} \{(s,s') | (\exists c_1, c_2 \,.\, c_1 \cup c_2 \subseteq s \wedge s' = s \cup (c_1, c_2))\} \\
s \xrightarrow{\pi'} s' &\stackrel{def}{=} \{(s,s') | (\exists c_1, c_2 \,.\, (c_1, c_2) \subseteq s \wedge s' = s \cup c_1 \cup c_2)\} \\
s \xrightarrow{\xi} s' &\stackrel{def}{=} \{(s,s') | s' \subseteq s\}
\end{aligned}
$$

where $\subseteq$ is defined on $S \times S$ as: $s' \subseteq s \stackrel{def}{=} \exists s''.s' \cup s'' = s$. The exploitation of a given knowledge (i.e. a given set of data components) to deduce new information (i.e. new data) consists in the application of zero or more of the above operations in any order and any number of times. A set of data components $s'$ (or a single data components) is thus said to be deducible from a set of data component $s$ if and only if there exists a sequence of applications of the five basic operations which allows us to obtain $s'$ from $s$. Given any subset[5] $E = \{x_1, .., x_n\}$ of $\{\gamma, \gamma', \pi, \pi', \xi\}$, $\xrightarrow{E}$ is defined as the reflexive-transitive closure of the relation $\xrightarrow{x_1} \cup ... \cup \xrightarrow{x_n}$. The predicate *known_in* is defined as follows: $c$ *known_in* $s$ if and only if $s \xrightarrow{\{\gamma,\gamma',\pi,\pi',\xi\}} c$: $c$ *known_in* $s$ if and only if $c$ is deducible from $s$. We recall here some of the properties that were proved in [2] and that we will use in the sequel.

**Lemma 1.** *If* $E \subseteq \{\gamma, \gamma', \pi, \pi', \xi\}$ *then* $\forall s, s', s''.(s \xrightarrow{E} s') \Rightarrow (s \cup s'' \xrightarrow{E} s' \cup s'')$

**Proposition 2 Confluence.** *Relation* $\xrightarrow{E}$ *is confluent for any given subset* $E$ *of* $\{\gamma, \gamma', \pi, \pi'\}$: *i.e.* $s \xrightarrow{E} s_1$ *and* $s \xrightarrow{E} s_2$ *then there exists* $s_3$ *such that* $s_1 \xrightarrow{E} s_3$ *and* $s_2 \xrightarrow{E} s_3$. *Furthermore* $s_3 = s_1 \cup s_2$ *is always a solution.*

**Proposition 3.** *If* $s \xrightarrow{\{\gamma,\gamma',\pi,\pi',\xi\}} s'$, *then there exists* $s''$ *such that* $s \xrightarrow{\{\gamma,\gamma',\pi,\pi'\}} s''$ *and* $s'' \xrightarrow{\{\xi\}} s'$. *More generally for any non-empty subset* $E$ *of* $\{\gamma, \gamma', \pi, \pi'\}$ *if* $s \xrightarrow{E \cup \{\xi\}} s'$, *then there exists* $s''$ *such that* $s \xrightarrow{E} s''$ *et* $s'' \xrightarrow{\{\xi\}} s'$

**Proposition 4.** *If* $s \xrightarrow{i} s''$ *and* $s'' \xrightarrow{j} s'$, *with* $i \in \{\gamma, \pi\}$ *and* $j \in \{\gamma', \pi'\}$ *then there exists* $s'''$ *such that* $s \xrightarrow{\{j\}} s'''$ *and* $s''' \xrightarrow{i} s'$.

[5] Only non empty subsets are useful in practice.

**Corollary 5.** *If* $s \xrightarrow{\{\gamma,\gamma',\pi,\pi'\}} s'$, *then there exists* $s''$ *such that* $s \xrightarrow{\{\gamma',\pi'\}} s''$ *and* $s'' \xrightarrow{\{\gamma,\pi\}} s'$.

We then define a first algorithm described using the chemical reaction paradigm. The algorithm is supposed to stop whenever a fixed point is reached[6]. The algorithm performs a decomposition of $s$ into components obtained by applying only decomposition operations $\gamma', \pi', \xi$. The returned value, $decomp(s)$, is by definition the last value of $s_1$:

$(Init)$ $s_1 := s$
$(Act1)$ If $\exists\ k, c.(k \subseteq s_1 \wedge c_{k-1} \subseteq s_1)$ Do $s_1 := s_1 \cup c$ End
$(Act2)$ If $\exists\ c, c'.((c, c') \subseteq s_1)$ Do $s_1 := s_1 \cup c \cup c'$ End

(Decomposition algorithm: *decomp*)

**Proposition 6.** *For any* $s$ *with an explicit value, the previous algorithm terminates and the returned value* $s_1$ *is the set of all data components* $c$ *such that* $s \xrightarrow{\{\gamma',\pi',\xi\}} c$.

*Proof.* The proof of termination is quite straightforward: each step consumes one sub-tree of the abstract syntactic tree that corresponds to the value of $s$. The fact that the returned value $s_1$ is the set of all data components $c$ such that $s \xrightarrow{\{\gamma',\pi',\xi\}} c$ is proved in two steps. First we prove that any component $c$ of $s_1$ is such that $s \xrightarrow{\{\gamma',\pi',\xi\}} c$, next we prove the converse. The first part is straightforward, and the second is a direct application of propositions 3 and 2.

Let us now consider the following algorithm which uses the previous one:

$(Init)$ $s_1 := decomp(s);\ s_1' := s' - s_1$
$(Act1)$ If $\exists\ c, k.c_k \subseteq s_1'$ Do $s_1' := (s_1' \cup c \cup k) - s_1 - c_k$ End
$(Act2)$ If $\exists\ c, c'.(c, c') \subseteq s_1'$ Do $s_1' := (s_1' \cup c \cup c') - s_1 - (c, c')$ End

(Decision Algorithm)

**Proposition 7.** *The previous algorithm is a decision procedure which takes two explicitly defined parameters* $s$ *and* $s' = \{c\}$ *and returns an empty set* $s_1'$ *whenever* $c$ *known_in* $s$ *and a non empty set otherwise.*

*Proof.* The proof for the termination of the algorithm is similar to that of proposition 6. The proof of correctness and the proof of completeness both use the same invariant property: $s_1 \cup s_1' \xrightarrow{\{\gamma,\pi,\xi\}} s'$. Thus when $s_1' = \emptyset$ then $s \xrightarrow{\{\gamma',\pi'\}} s_1 \xrightarrow{\{\gamma,\pi,\xi\}} s'$ which proves the correctness. If the last value of $s_1'$ is not $\emptyset$ then we can easily prove by contradiction using propositions 3, 2, 6 and corollary 5 that $s \cup s_1' \xrightarrow{\{\gamma,\gamma',\pi,\pi',\xi\}} s'$ but that $s \xrightarrow{\{\gamma,\gamma',\pi,\pi',\xi\}} s'$ is false: it is not possible to build $s'$ from $s$ without using an element of $s_1'$ (in fact all elements of $s_1'$ are necessary).[7]

[6] The algorithm can be implemented by only firing actions which *effectively* change the state, and by terminating when no more action can be fired.

[7] More detailled proofs can be found in the extended version of this paper.

## 6 Computing the abstract model and the abstract properties

Now we improve on the approach proposed in the previous section by bringing some automation for steps (a) and (b). In step (a), given a logical formula $p$ (e.g. $p : (\mathcal{S} \times (\mathcal{A} \times C) \times \mathcal{S}) \rightarrow Bool$ defined in section 3 we are looking for an abstract logical formula $p_a$ such that $\forall x.p(x) \Rightarrow p_a(\hat{h}(x))$ (i.e. we consider that $h$ is the identity function on labels of $\mathcal{A}$). In step (b) given a logical formula $p$ (e.g. $p$ is $inv$) we are looking for an abstract logical formula $p_a$ (i.e. $p_a$ is $inv_a$) such that $\forall x.p_a(\hat{h}(x)) \Rightarrow p(x)$. The main difference in both cases comes from the direction of the implication (i.e. it goes from concrete to abstract in the first case and from abstract to concrete in the second). The goal will be said to be negative in the first case and positive in the second. In both cases, formulae are supposed to be expressed in a very simple (typed) logical language defined on $C$ and $S$ using the operators and connectors of basic set theory, i.e. $\vee, \wedge, \in, \neg, \Rightarrow$, together with the predefined predicate *known_in*. For the sake of simplicity, quantifiers are omitted and free variables are considered to be universally quantified. The homomorphic extension of $\hat{h}$ can now be defined on this new language. In the sequel we will use the same notation $\hat{h}$ to refer to it because it generalizes the previous extension $\hat{h}$[8]. We first consider the following preliminary result:

**Proposition 8.** $\forall c, s.c\ known\_in\ s \Rightarrow \hat{h}(c)\ known\_in\ \hat{h}(s)$

*Proof.* $c$ *known_in* $s$ means by definition that $s \xrightarrow{\{\gamma,\gamma',\pi,\pi',\xi\}} c$, and thus that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} c$ where $a_1, a_2, \ldots, a_k$ are elements of $\{\gamma, \gamma', \pi, \pi', \xi\}$. $\hat{h}(c)$ has the same structure as $c$, and each component $c'$ of $s$ has its counterpart $\hat{h}(c')$ in $\hat{h}(s)$. In order to prove the proposition we just need to associate each step $s_i \xrightarrow{a_{i+1}} s_{i+1}$ with the corresponding step $\hat{h}(s_i) \xrightarrow{a_{i+1}} \hat{h}(s_{i+1})$.

We now describe the main steps of an algorithm for checking that given a formula $p$ and a goal (i.e. positive or negative), we can use $\hat{h}(p)$ as an abstract formula (i.e. $p_a = \hat{h}(p)$). The algorithm is based on the ten rules of figure (3) below and works as follows: we use the initial formula $p$ and its associated sign as an initial goal; at each step we try to match the current logical formula[9] and its associated sign to the *formula* and *sign* part of a rule; if one of the rules (1) to (4) is matched, then each matching sub-expression, i.e. the sub-expression matching $x$ and/or $y$, forms a new sub-goal that has to be checked recursively using the sign specified by the rule for the corresponding sub-expression[10]; if one of the rules (5) to (10) is applied no new sub-goal is generated; the checking stops when

[8] We will assume that $\hat{h}(true)$ (resp. $\hat{h}(false)$) is defined and that $\hat{h}(true) = true$ (resp. $\hat{h}(false) = false$).

[9] The symbol $a$ used in rules (9) and (10) can only match by convention a variable free element of $C$. The symbols $x$ and $y$ can match any formula.

[10] Each rule (1), (2), (3), (4), or (7) in fact specifies two rules: one for a positive sign and another for a negative one.

all sub-goals have been checked, or a sub-goal does not match any rule. In the first case the checking is said to be successful. The test, $\hat{h}^{-1}(a) = \{a\}$, can be checked automatically provided that the set $B_0^{injective} = \{b \in B_0 | \hat{h}^{-1}(b) = \{b\}\}$ (or a subset of it) is provided by some mean (typically $B_0^{injective}$ is provided by the user at the time a particular function $h$ is proposed): the test is positive if only if $a$ (which is variable free by definition) is only made of components of $B_0^{injective}$ (e.g. $a = (K_{any}, K_{any})_{K_{any}}$, with $B_0^{injective} = \{K_{any}\}$).

| | *Formula* | *Sign* | $x$ | $y$ |
|---|---|---|---|---|
| (1) | $x \wedge y$ | $+/-$ | $+/-$ | $+/-$ |
| (2) | $x \vee y$ | $+/-$ | $+/-$ | $+/-$ |
| (3) | $\neg x$ | $+/-$ | $-/+$ | |
| (4) | $x \Rightarrow y$ | $+/-$ | $-/+$ | $+/-$ |
| (5) | $x = y$ | $-$ | | |

| | *Formula* | *Sign* |
|---|---|---|
| (6) | $x \in y$ | $-$ |
| (7) | $true, false$ | $+/-$ |
| (8) | $c$ *known_in* $s$ | $-$ |
| (9) | $x = a$ *s.t.* $\hat{h}^{-1}(a) = \{a\}$ | $+$ |
| (10) | $a \in e$ *s.t.* $\hat{h}^{-1}(a) = \{a\}$ | $+$ |

*figure* (3)

**Proposition 9.** *The previous algorithms always terminates. Whenever it succeeds then taking $p_a$ to be $\hat{h}(p)$ we have $\forall x.p_a(\hat{h}(x)) \Rightarrow p(x)$ if the goal was positive and $\forall x.p(x) \Rightarrow p_a(\hat{h}(x))$ otherwise.*

*Proof.* The proof of termination is straightforward. The proof of correctness is done by structural induction on $p$. This leads to one induction step per rule. For the positive side of rule (1) we have to prove that given any four logical formulae $q, q', q_a, q'_a$, that satisfy $\forall x.q_a(\hat{h}(x)) \Rightarrow q(x)$ and $\forall x.q'_a(\hat{h}(x)) \Rightarrow q'(x)$ then $\forall x.p_a(\hat{h}(x)) \Rightarrow p(x)$ where $p = q \wedge q'$ and $p_a = q_a \wedge q'_a$. For the negative side of rule (1) and for rules (2) to (7), the proof is similar. For rule (8) we just need to use proposition 8. For rule (9) (resp. for rule (10)) we use the fact that $\hat{h}^{-1}(a) = \{a\}$ to prove that $x' \in \hat{h}^{-1}(x) \Rightarrow x' = a$ (resp. that $e' \in \hat{h}^{-1}(e) \Rightarrow a \in e'$).

The algorithm will thus be used to automate steps (a) and (b) in all situations where the algorithm succeeds. In case of failure the unsatisfied sub-goals can still be discharged using theorem proving, or a new formula verifying the sub-goal can be proposed by the user. In practice it has always been quite easy to add new rules similar to rules (9) and (10) in the rare cases of failure of the algorithm. As an illustration of the use of the proposed algorithm let us first consider the first kind of security properties identified in section 4, i.e. $\neg(c$ *known_in* $s)$. The algorithm proceeds as follows: the initial goal is positive as it used for step (b); rule (3) is applied with a positive sign and with $x$ matching sub-expression $c$ *known_in* $s$; in the column for $x$, we find a negative sign associated to a positive goal (i.e. the sign of the goal is in the column "Sign", here on the left side of the column); thus a new negative sub-goal $c$ *known_in* $s$ is generated; this negative sub-goal matches rule (8) and the checking succeeds. Thus the only problem here is to find an adequate function $h$. We propose, $h$ such that $h(k) = if\ k = K_S^{-1} then\ K_S^{-1}\ else\ K_{other}$ and $\hat{h}(d) = D_{any}$, where $d$ is a typed variable that is supposed to range over domain $D$. For keys, $h$ will return either

$K_S^{-1}$ or $K_{other}$. The second key, $K_{other}$, will necessarily be part of the intruder knowledge for $\hat{h}(M)$ (i.e. $K_{other} \subseteq \hat{h}(s_I)$), but the first one $K_S^{-1}$ should not be deducible from $\hat{h}(s_I)$. It is private. The distinction between these two kinds of keys is essential here: at least one key must be private; the other ones which can be represented using (i.e. collapsed into) a single key $K_{other}$ may be known by the intruder. The same distinction between private and non private data is on the other hand not useful here for basic data. Thus $h$ will return the same value for all data in $D_{any}$, and this basic data does not need to be private (i.e. $D_{any}$ *known_in* $\hat{h}(s_I)$). After a few steps the knowledge of the intruder will thus typically be $\hat{h}(s_I) = K_{other} \cup D_{any} \cup (K, D)_{K_S^{-1}}$. The previous discussion is quite representative of issues that have to be considered during the identification of $h$. A misconception in $h$ results in the identification of non existent flaws, and can easily be fixed by reducing some of the collapses formalized by $h$.

Let us now consider the example of invariant (1): $\forall\ k,\ x.\ (k, x) \in d_A \Rightarrow (k, x) \in d_S$. First the formula is transformed into an equivalent quantifier free formula. This is done by introducing two constants (i.e. *eigenvariables*), let us say $K_{this}$ and $D_{this}$. Then we define $h$ such that $h(k) = \mathit{if}\ k = K_S^{-1}\ \mathit{then}\ \ K_S^{-1}\ \mathit{elseif}\ k = K_{this}\ \mathit{then}\ \ K_{this}\ \mathit{else}\ K_{other}$ and $h(d) = \mathit{if}\ d = D_{this}\ \mathit{then}\ \ D_{this}\ \mathit{else}\ D_{other}$. Thus $B_0^{injective} = \{D_{this}, K_{this}, K_S^{-1}\}$. The previous algorithm terminates successfully using rule (10) in particular, where $a$ matches $(D_{this}, K_{this})$. The same invariant $(D_{this}, K_{this}) \in d_A \Rightarrow (D_{this}, K_{this}) \in d_S$, can thus be used on $M$ and $\hat{h}(M)$. The automatic checking of the invariant completes successfully on $\hat{h}(M)$. Indeed if the $d_S$ directory is chosen to be both functional and injective then $\hat{h}(d_S) = (D_{this}, K_{this}) \cup (D_{other}, K_{other})$ and $\hat{h}(d_A) \subseteq \hat{h}(d_S)$ on all states of $\mathcal{R}(\hat{h}(M))$.

Now for invariant (2), i.e. $\forall k, x.(k, x)_{K_S^{-1}}$ *known_in* $s_I \Rightarrow (x, k) \in d_S$, we can use the same function $h$ as for invariant (1). For similar reasons, the algorithm terminates successfully and the abstract invariant, i.e. $(K_{this}, D_{this})_{K_S^{-1}}$ *known_in* $\hat{h}(s_I) \Rightarrow (D_{this}, K_{this}) \in d_S$, can be checked automatically on $\hat{h}(M)$.

## 7 Conclusion

We have shown how to automate the formal verification of cryptographic protocols for a large variety of security properties. The proposed approach relies on the general theorem proving framework originally proposed in [2] and incorporates abstract interpretation inspired facilities, thus applying techniques developed for the verification of general temporal properties (i.e. [6, 4, 5, 9, 12]). We have first transposed work in [4, 9, 12] to the framework of verification of security properties proposed in [2]. This entails providing a decision procedure for the intruder's (unbounded) knowledge. But we have also significantly improved the mechanization proposed in [9] by providing an algorithm for computing the abstract model and the abstract properties, given an abstraction function. The algorithm may

fail to show that a particular sub-expression meets the sub-goal. In this situation, which is very rare in practice the user should then either prove manually that the problematic sub-expression indeed meets the sub-goal, or should provide a new sub-expression himself. An alternative, more restrictive, but probably more elegant approach would be to characterize the precise language for which the abstract property can be computed automatically and restrict the logic language that can be used for describing the protocol and for expressing security properties. We would then consider the checking algorithm of section 6 as a typing or static inference algorithm. In doing so we would obtain a complete mechanization in all cases once the abstraction function is provided. The approach that is proposed here for the verification of cryptographic protocols is somewhat more complex than the two model-checking based approaches proposed so far (i.e. [11] and [7]). This is mainly because the latter approaches do not encompass the first abstraction phase, and the user has to provide the simplified finite model directly. There is thus a risk that the informal abstraction step implicitly performed by the user is unsafe and compromises the result of the analysis itself (by validating problematic protocols). The main objective of the proposed approach is indeed to prove the absence of flaws, and not only to identify flaws. The two kinds of approaches are thus complementary in their objectives. In order to cope with the abstraction problem, some guidelines are provided in [11] and are informally justified for the writing of the finite model. For example, the number of different keys that the intruder may use is specified. This number is independent of the protocol or of the property at hand. Even if this is acceptable in practice for many authentication protocols it is a severe limitation for more general cryptographic protocols, as it is quite easy to exhibit protocols for which problematic scenarios require larger numbers of distinct keys. In the proposed approach, the number of distinct keys (featured by the size of the set $h(K)$, e.g. 2 and 3 in the examples of the previous section) will typically depend on the property and the protocol at hand. Finally we believe that some of the results presented here are quite general and could also be used with pure model checking approaches. In [11] and [7], for example, the number of internal steps that the intruder may perform in order to deduce new data from existing one is implicitly bounded so as to keep the model finite. In many cases (i.e. for many protocols) this decision could be justified formally using the model and the results of [2]. But the decision algorithm proposed in section 5 in fact suppresses the need for such limitation of the number of steps, and could be used in conjunction with approaches like [11] and [7]. The approach is currently being applied successfully for the verification of large electronic commerce protocols[11]. The abstracted models experimented so far have always been very small in terms of the number of states. This is mainly due to the fact that in the proposed approach an abstract function has only to preserve the particular property for which it is provided and not for a whole logic or a class of properties as it is the case in other approaches such as [11], [7], or [6, 10, 8] for example.

[11] *http://www.dyade.fr/actions/VIP/vip.html*

## References


1. J.-P. Banâtre and D. Le Métayer. Gamma and the chemical reaction model: ten years after. In *Coordination programming: mechanisms, models and semantics.* World Scientific Publishing, IC Press, 1996.
2. D. Bolignano. Formal verification of cryptographic protocols. In *Proceedings of the third ACM Conference on Computer and Communication Security*, 1996.
3. D. Bolignano. Towards the Formal Verification of Electronic Commerce Protocols. In *Proceedings of the 10 th IEEE Computer Security Foundations Workshop.* IEEE, June 1997.
4. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
5. Rance Cleaveland, Purush Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *Proceedings of SAS'95.* LNCS, 1995.
6. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL$^*$, $\exists$CTL$^*$ and CTL$^*$. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, IFIP Transactions, Amsterdam, June 1994. North-Holland/Elsevier.
7. G.Leduc, O. Bonaventure, E. Koerner, L. Léonard, C. Pecheur, and D. Zanetti. Specification and verification of a ttp protocol for the conditional access to services. In *Proceedings of the 12th Workshop on the Application of Formal Methods to System Development (Univ Montreal)*, 1996.
8. S. Graf. Verification of a distributed cache memory by using abstractions. In *Workshop on Computer-Aided Verification, CAV'94, Stanford.* LNCS 818, Springer Verlag, jun 1994.
9. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
10. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design Volume 6, Issue 1*, 1995.
11. G. Lowe. An attack on the needham-schroeder public-key protocol. In *Information Processing Letters*, 1995.
12. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.