Abstracting Cryptographic Protocols with Tree Automata *

David Monniaux
David.Monniaux@ens.fr
Laboratoire d'Informatique de l'École Normale Supérieure†
& SRI International, Computer science laboratory‡

June 13, 1999

Abstract

Cryptographic protocols have so far been analyzed for the most part by means of testing (which does not yield proofs of secrecy) and theorem proving (costly). We propose a new, abstract interpretation based, approach, using regular tree languages. The abstraction we use seems fine-grained enough to be able to certify some protocols. Both the concrete and abstract semantics of the protocol description language and implementation issues are discussed in the paper.

This paper is to be published in the proceedings of SAS '99, in Springer-Verlag's Lecture Notes in Computer Science series. The reader is invited to buy the proceedings of the conference.

1 Introduction

Our goal is to provide mathematical and algorithmic tools for the analysis of cryptographic protocols through abstract interpretation.

1.1 Verifying Cryptographic Protocols

Cryptographic protocols are specifications for sequences of messages to be exchanged by machines on a possibly insecure network, such as the Internet, to establish private or authenticated communication. These protocols can be used to distribute sensitive information, such as classified material, credit card numbers or trade secrets, or to create digital signatures.

Many cryptographic protocols have been found to be flawed; that is, there exists a way for an intruder that has gained partial or total control over the

*This work was partially funded by NSF grant CCR-9509931.

†ENS-LIENS, 45, rue d'Ulm , 75230 PARIS cédex 5, FRANCE; http://www.di.ens.fr

‡SRI-CSL, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, USA; http://www.csl.sri.com

communication network and is able to read, suppress and forge messages to trick the communicating machines into revealing some sensitive information or believing they have an authenticated communication, whereas they are actually communicating with the intruder. Several tools and techniques have therefore been devised for analyzing and verifying the security of cryptographic protocols.

A common feature of these techniques, including ours, is that they address the design of the protocol rather than the strength of the underlying cryptographic algorithms, such as message digests or encryption primitives. For instance, it is assumed that one may decrypt a message encrypted with a public key only when possessing the corresponding private key.

Whereas belief logics [3, 9, 8, 19, 20] try to deal with the rationale behind the design of a protocol, the other methods (theorem proving, model checking) are based on some kind of well-defined model of the computation [15]. The next part of this paper will describe the model we are considering.

Methods based on such models can be classified into two main categories:

1. Testing: Here a limited but wide set of possible attacks is generated and systematically tried against the protocol. The hope is that this set is wide enough so that any attack will be detected. In other words, a large subset of the space of reachable states of a certain configuration of a protocol is exhaustively explored by concrete model-checking. Efficient implementations have been devised [14, 17, 13].

2. Theorem proving: Here a semi-automated proof system is used; while such a method consumes lots of human resources, automation inside the tool can make it more bearable [18].

This paper intends to demonstrate how abstract interpretation techniques, and more particularly abstract model checking of infinite state computation systems, can be applied to the problem of analyzing cryptographic protocols. To our knowledge, this is the first time that an abstract domain has been proposed for cryptographic protocols.

A salient point of our approach is that is fully automatic from the protocol description to the results. Contrary to some other methods that use abstraction, but require the user to design himself an abstraction or manually help a program to compute invariants, our method requires no user input except the description of the protocol and the cryptographic primitives involved.

1.2 Abstract Interpretation

Abstract interpretation [5, 6] is a generic theory for the analysis of computation systems. Its basic idea is to use approximations in ordered domains in a known direction (lower or upper), to get reliable results. This order relation is preserved throughout monotonic operators.

Here we shall approximate transition systems. We consider a transition relation r on a "concrete" state space Σ. We also consider an "abstract" transition relation $r^\sharp$ on an "abstract" state space $\Sigma^\sharp$. An abstraction relation $a \subseteq \Sigma \times \Sigma^\sharp$ links the two spaces. By $a^{-1}(X^\sharp)$ where $X^\sharp \subseteq \Sigma^\sharp$, we mean $\{x \in \sigma \mid \exists x^\sharp \in X^\sharp \; a(x, x^\sharp)\}$.

$$\begin{tikzcd}
\sigma_0 \arrow[r, "r"] \arrow[d, "a"'] & \sigma_1 \arrow[r] \arrow[d, dashed] & \cdots \arrow[r] & \sigma \arrow[d, dashed] \\
\sigma_0^\sharp \arrow[r, dashed, "r^\sharp"'] & \sigma_1^\sharp \arrow[r, dashed] & \cdots \arrow[r, dashed] & \sigma^\sharp
\end{tikzcd}$$

Figure 1: The abstract transition relation follows the concrete one.

For instance, Σ could be $\wp(\mathbb{Z})$ and $\Sigma^\sharp$ the set of (possibly empty) intervals of $\mathbb{Z}$ (given by their bounds). The abstraction relation, in that example, is the following:

$$\forall X \in \wp(\mathbb{Z})\ a(X, [\alpha, \beta]) \iff X \subseteq [\alpha, \beta].$$

We require that the two relations satisfy the following simulation condition[1] (see Fig. 1.):

$$\forall x, y \in \Sigma, x^\sharp \in \Sigma^\sharp,\ r(x, y) \wedge a(x, x^\sharp) \Rightarrow \exists y^\sharp \in \Sigma^\sharp\ r^\sharp(x^\sharp, y^\sharp) \wedge a(y, y^\sharp).$$

This implies that for all σ_0 and $\sigma_0^\sharp$ so that $a(\sigma_0, \sigma_0^\sharp)$, noting $A_0 = \{\sigma \mid \sigma_0 \rightarrow_r^* \sigma\}$ and $A_0^\sharp = \{\sigma^\sharp \mid \sigma_0^\sharp \rightarrow_{r^\sharp}^* \sigma^\sharp\}$, $A_0 \subseteq a^{-1}(A_0^\sharp)$.

We are only interested in safety properties; in the concrete model we are considering here, liveness properties can't be obtained, since the intruder can deny any network service by just stopping network transmission. To prove that a property P holds for all elements in A_0, it is sufficient to show that it holds for all elements in $r^{-1}(A_0^\sharp)$. That will be the basic idea of our method of analysis.

2 Concrete Model

There exists no standard model for cryptographic protocols, although there is progress in the design of common notations with well-defined semantics [16, 10]. We therefore had to provide a sensible model of what a cryptographic protocol is. We chose a simple syntax and a simple semantics, appropriate to describe the interactions between a fixed number of machines, or **principals** (subtler models, like the spi-calculus [1], could perhaps be used for successful analyses, but they are significantly more complex than ours).

2.1 Terms, Rewrite Systems, and Notations

Let us consider a signature [11, p. 249] [4, preliminaries] $\mathcal{F}$ and the **free algebra** of **terms** $T(\mathcal{F})$ on that signature. Messages exchanged on the network are elements of that algebra. We will also consider the algebra $T(\mathcal{F}, \mathcal{X})$ of terms with variables in $\mathcal{X}$. When $t \in T(\mathcal{F}, \mathcal{X})$, $(X_i)_{i \in i}$ is a family of variables, $(x_i)_{i \in i}$ is a family of terms, we write $t[x_i/X_i]$ the term obtained by parallel substitution of X_i by x_i in t. We note $FV(t)$ the set of free variables of t.

Let us also consider a notion of "possible computation"; this notion is defined by a function $\mathcal{K} : \wp(T(\mathcal{F})) \rightarrow \wp(T(\mathcal{F}))$ that computes the closure of a subset of $T(\mathcal{F})$ by the following operations:

[1]Readers coming from a type theory background may see it as a kind of subject reduction property.

- a subset $\mathcal{O}$ of the function symbols found in $\mathcal{F}$; that is, if the symbol f belongs to the subset $\mathcal{O}_n$ of elements of $\mathcal{O}$ of arity n, then for all n-tuple $(x_i)_{1\leq i\leq n}$ of elements of $\mathcal{K}(X)$, then $f(x_1,\ldots,x_n)$ belongs to $\mathcal{K}(X)$;
- a set $\mathcal{R}$ of rewrite rules [11, p. 252] over $T(\mathcal{F})$ of a certain kind described in the next paragraph.

So an element x of $T(\mathcal{F})$ is deemed to be "possibly computable" from $X \subseteq T(\mathcal{F})$ if $x \in \mathcal{K}(X)$. We write $\wp(T(\mathcal{F}))_{\mathcal{K}}$ the fixpoints of $\mathcal{K}$.

We require that the rules in $\mathcal{R}$ be of the following form: $a \to x$, where a is a term with variables over the signature $\mathcal{F}$ and x is a variable so that x appears exactly once in a. We will call such systems **simplification systems**.

2.1.1 Example

We shall consider the following signature $\mathcal{O}_C$:

$$\mathcal{O} = \{\mathsf{pair}(,);\mathsf{proj1}();\mathsf{proj2}();\mathsf{encrypt}(,);\mathsf{decrypt}(,);\mathsf{pk\_encrypt}(,);\mathsf{pk\_decrypt}(,)\}$$
$$\mathcal{O}_C = \mathcal{O} \cup \{\mathsf{public}(),\mathsf{private}()\}$$

and the following rewrite rules:

- $\mathsf{proj1}(\mathsf{pair}(x,y)) \to x$,
- $\mathsf{proj2}(\mathsf{pair}(x,y)) \to y$,
- $\mathsf{decrypt}(\mathsf{encrypt}(x,k),k) \to x$,
- $\mathsf{pk\_decrypt}(\mathsf{pk\_encrypt}(x,\mathsf{public}(k)),\mathsf{private}(k)) \to x$.

2.2 Concrete Semantics

Let us consider a finite set $\mathcal{P}$ of principals. Each principal $p \in \mathcal{P}$ has a finite set R_p of registers, each containing an element of $T(\mathcal{F}) \cup \{\perp\}$ — the $\perp$ element meaning "uninitialized" — and a program x_p to execute. The program is a finite sequence (possibly empty) of commands, which can be of the three possible types:

- $!t$, read as "output t", where $t \in T(\mathcal{F}, R_p)$;
- $r \simeq t$, read as "match register r against t", where $r \in \{1,\ldots,r_p\}$ and $t \in T(\mathcal{F}, R_p \cup \bar{R}_p)$; by r we shall mean "the current contents of register r" and by $\bar{r}$ we shall mean "store matched value into register r". $\bar{R}_p = \{\bar{r} \mid r \in R_p\}$ is a copy of R_p.
- $?r$, read as "input register r", where $r \in R_p$.

We shall write $h :: t$ the sequence whose head is h and tail t, and ε the empty sequence. The local state of a principal is therefore the content of its registers and the program it has yet to execute. The global state is the tuple (indexed by $\mathcal{P}$) of the local states, together with the state of the intruder, which is an element of $\wp(T(\mathcal{F}))_{\mathcal{K}}$. The set of global states is noted Σ.

We define the semantics of the system by a nondeterministic transition relation $\to$. Let S and S' be two global states. We note $S.p$ the local state of the principal p in S and $S.I$ the intruder knowledge in S. In a local state L, we note $L.r$ the contents of register r and $L.P$ the program. The definition of the transition relation is the following: $S \to S'$ if there exists $p_0 \in P$ so that:

- for all $p \in \mathcal{P}$ so that $p \neq p_0$, $S'.p = S.p$;
- $S.p_0.P = h :: \tau$ and either
  - $h = ?r_0$ and
    - for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$,
    - $S'.p_0.r_0 \in S.I$
    - $S'.p_0.P = \tau$
  - $h = !t$ and
    - for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$,
    - $S'.I = \mathcal{K}(S.I \cup \{t[S.p_0.r/r \mid r \in R_{p_0}]\})$
    - $S'.p_0.P = \tau$
  - $h = r \simeq t$ and either
    - there exists an unifier for the variables in $\bar{R}_{p_0}$ between $t[S.p_0.r/r \mid r \in R_{p_0}]$ and $S.p_0.r$; then
      - for all $\bar{r} \in \bar{R}_{p_0} \setminus FV(t)$, $S'.p_0.r = S.p_0.r$
      - $t[S.p_0.r/r \mid r \in R_{p_0}, S'.p_0.r/\bar{r} \mid r \in R_{p_0}] = S.p_0.r$
      - $S'.p_0.P = \tau$
    - such an unifier does not exist; then
      - for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$
      - $S'.p_0.P = \varepsilon$

3 Tree Automata and Operations on Them

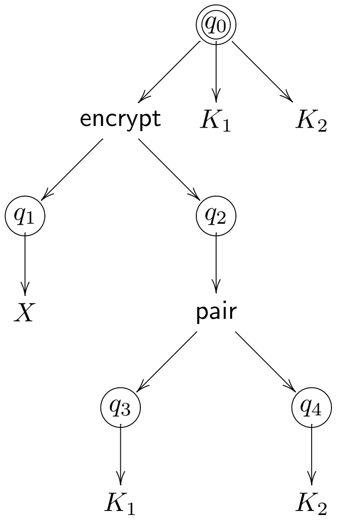
Regular languages, implemented as finite automata, are a well known domain abstracting sets of words on an alphabet. Here, we abstract sets of terms on a signature by regular tree languages, and we consider the generalization to n-ary constructors of finite automata: tree automata [4].

Please note that the algorithms presented here are given mainly as proofs that the functions described are computable. There are several ways to implement the same functions, and efficient implementations are likely to be more complex than the simple schemes given here.

3.1 Tree Automata

We use non-deterministic top-down tree automata [4, §1.6] to represent subsets of $T(\mathcal{F})$; an automaton is a finite representation the subsets of terms it recognizes. A top-down tree automaton over $\mathcal{F}$ is a tuple $A = \langle Q, q_0, \Delta \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and Δ is a set of rewrite rules[2]

[2]The reader should not confuse these rewrite rules, meant as a notation for the tree automaton, with the rewrite rules in $\mathcal{R}$.



(a) The tree. The circled node represent the states, the others the symbols

$$
\begin{array}{lll}
q_0(\mathsf{encrypt}(x,y)) & \rightarrow_A & \mathsf{encrypt}(q_1(x), q_2(y)) \\
q_0(K_1) & \rightarrow_A & K_1 \\
q_0(K_2) & \rightarrow_A & K_2 \\
q_1(X) & \rightarrow_A & X \\
q_2(\mathsf{pair}(x,y)) & \rightarrow_A & \mathsf{pair}(q_3(x), q_4(y)) \\
q_3(K_1) & \rightarrow_A & K_1 \\
q_4(K_2) & \rightarrow_A & K_2
\end{array}
$$

(b) The set Δ of rewrite rules.

Figure 2: An automaton $(\{q_0, \dots, q_4\}, q_0, \Delta)$ on the signature $\mathcal{O}_\mathcal{C}$ with added constants $\{X, K_1, K_2\}$ recognizing $\{\mathsf{encrypt}(X, \mathsf{encrypt}(K_1, K_2)), K_1, K_2\}$.

over the signature $\mathcal{F} \cup Q$ where the states are seen as unary symbols. The rules in Δ must be of the following type:

$$q(f(x_1, \dots, x_n)) \rightarrow_A f(q_1(x_1), \dots, q_n(x_n))$$

where $n \geq 0$ $f \in \mathcal{F}_n$, $q, q_1, \dots, q_n \in Q$, $x_1, \dots, x_n$ being variables. When $n = 0$, the rule is therefore of the form $q(a) \rightarrow a$. Defining

$$L_q(a) = \{t \in T(\mathcal{F}) \mid q(t) \rightarrow_A^* t\},$$

we denote by $L(a) = L_{q_0}(A)$ the language recognized by A.

We actually will be using a narrower subclass of tree automata, which we be referred to as **special automata**, over $\mathcal{F}$; we shall note the set of these automata $\mathcal{A}_\mathcal{F}$. Namely, we will require that the set Δ of rewrite rules defining the automaton can be partitioned between two subsets:

- rules of the form $q(f(x_1, \dots, x_n)) \rightarrow f(q(x_1), \dots, q(x_n))$ where $q \in Q$ and $f \in \mathcal{O}_n$; we require that if there exists $n \geq 0$ and $f \in \mathcal{O}_n$ so that $q(f(x_1, \dots, x_n)) \rightarrow f(q(x_1), \dots, q(x_n)) \in \Delta$ then $\forall n \geq 0, \forall f \in \mathcal{O}_n$, $q(f(x_1, \dots, x_n)) \rightarrow f(q(x_1), \dots, q(x_n)) \in \Delta$;
- rules of the form $q(f(x_1, \dots, x_n)) \rightarrow f(q_1(x_1), \dots, q_n(x_n))$ where $q, q_1, \dots, q_n \in Q$; we require the directed graph (Q, E) whose vertices are the states and the arrows are of the form $q \rightarrow_E q_i$, $1 \leq i \leq n$ for all the rules of the above form to be a tree.

This suggests a representation of such an automaton by a tree (see an exemple Fig. 2). Such a tree has two kind of nodes:

states that have:

- an (unordered and possibly empty) list of children, which are all symbol nodes;
- a boolean flag;

symbols that have an ordered list of children; there are as many as children as the arity of the symbol.

The symbolics in terms of rewrite rules of such a tree are the following:

- $\begin{tikzcd}[row sep=small] & & q_1 \\ q \arrow[r] & s \arrow[ur] \arrow[dr] & \vdots \\ & & q_n \end{tikzcd}$ where q, $q_1, \ldots, q_n$ are states and s is a n-ary symbol stands for the rewrite rule $q(s(x_1, \ldots, x_n)) \to_A s(q_1(x_1), \ldots, q_n(x_n))$;
- the flag on a state q, when true (represented by $\begin{tikzcd} q \arrow[loop right, "\mathcal{O}"] \end{tikzcd}$), means the set of rules $\{q(s(x_1, \ldots, x_n)) \to_A s(q(x_1), \ldots, q(x_n)) \mid s \in \mathcal{O}_n, n \in \mathbb{N}\}$.

Implementing the special automata as such trees allows for easy sharing of parts of the data structures.

3.2 Substitution and Matching

We extend canonically our definition of substitution of terms into terms into a definition of substitution of languages (sets of terms) into terms with variables. We furthermore overload this substitution notation to also consider a substitution function on automata so that for all term t and automata A_i, $L(t[A_i/X_i]) = t[L(A_i)/x_i]$. Such a substitution function, using only special automata, can be easily defined by induction on t.

Now we consider the reverse problem: given a language L and a term with variables t, give the set of solutions of $L = t[x_i/X_i]$. Such a solution is a family (L_i) of languages so that $L = t[L_i/X_i]$. We thus consider a function *match* so that if A is an automaton and t a term with variables, $match(A,t)$ is a finite subset of $FV(t) \to \mathcal{A}_{\mathcal{F}}$ and for any solution S in this set, $L = L(t[S_i/X_i])$. A computational definition follows.

We define $match_l(A,t)$, where $A = \langle Q, q_0, \Delta \rangle$ is an automaton and $t \in T(\mathcal{F}, \mathcal{X})$, recursively over the structure of t. Its value is a finite subset of $FV(t) \to \mathcal{A}_{\mathcal{F}}$.

- if $t = s(t_1, \ldots, t_n)$ where s is an n-ary symbol, then
$$\begin{aligned} match_l(A,t) = \{\lambda x \in \mathcal{X}\{\cup\{p_i \mid \forall 1 \le i \le n \ p_i \in match_l(\langle Q, q_i, \Delta\rangle, t_i)\} \\ \mid r : q(s(x_1, \ldots, x_n)) \to_A s(q_1(x_1), \ldots, q_n(x_n)) \in \Delta\}; \end{aligned}$$
- if $t \in \mathcal{X}$ then $match_l(\langle Q, q_0, \Delta\rangle, t) = \{[x \mapsto q_0]\}$.

The interesting property of this function is that for all linear[3] term $t \in T(\mathcal{T}, \mathcal{X})$, for all automaton $A = \langle Q, q_0, \Delta \rangle$, calling $x_1, \ldots, x_n$ the variables in t, for all

[3] A term is said to be **linear** if all variables have at most one occurrence in it.

terms $t_1, \ldots, t_n \in T(\mathcal{T})$, then $t[t_i/x_i, \ldots, t_n/x_n] \in L(A)$ if and only if there exists p in $match_l(A,t)$ so that for all i, $t_i \in L_{p(x_i)}(A)$). Informally, that means that this function returns the set of matches of the term against the automaton, giving for each match and for each variable the states in which this variable is to be recognized in that match.

We then construct a function *match* that has the same property, except that it does not constrain the terms to be linear.

$$match(A,t) = \{f \in match_l(A,t) \mid \forall x \in \mathcal{X} \bigcap_{q \in f(x)} L_q(A) \neq \emptyset\}.$$

The definition of $match_l$ translates into an algorithm on automata defined by trees as above. Then *match* is defined, using an effective test of whether the languages of several automata intersect [4, §1.7].

3.3 The $\mathcal{K}^\sharp$ Function on Automata

We want a function $\mathcal{K}^\sharp$ so that $\mathcal{K}(L(A)) \subseteq L(\mathcal{K}^\sharp(A))$ for all special automaton A. Actually, we shall give such a function so that there is $\mathcal{K}(L(A)) = L(\mathcal{K}^\sharp(A))$.

We will use a notion of position in a term [11, p. 250] as a sequence of positive integers describing the path from the root of the term to that position; ε will be the root position. $pos(t)$ is the set of positions in term t. By $t_{|p}$ we shall denote the subterm of t rooted at position t. We define the similar notions for trees.

Now we define $completion(A, \mathcal{R})$ (see Fig. 3 for an example) where A is a special automaton and $\mathcal{R}$ is a simplification system by induction on the structure of A: calling q_0 the initial state of A and calling $C_1, \ldots, C_n$ the children states of q_0, that is, the states two nodes away from q_0:

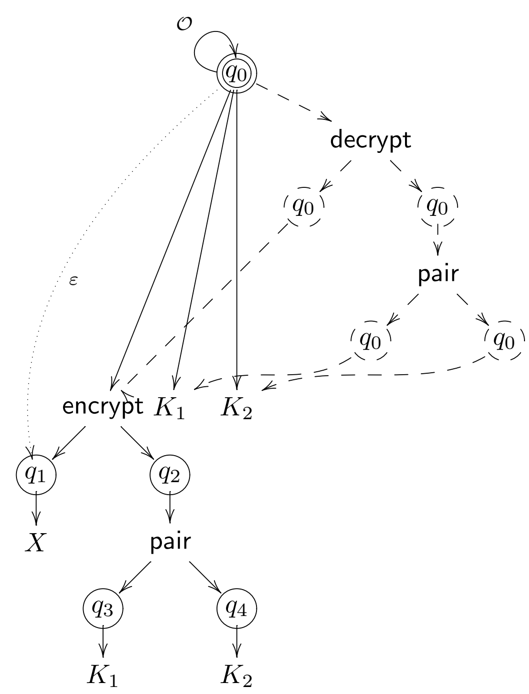
construct A', obtained by replacing in A the subtree starting from $C_1, \ldots, C_n$ by their image by $a \mapsto completion(a, R)$
repeat
  for $a \to x \in \mathcal{R}$ **do**
    for $f \in match(A', a)$ **do**
      if the following subtree is not already present, modulo state renaming **then**
        copy $A'_{|f(x)}$, replacing the state $f(x)$ by q_0 {adds a child to q_0}
      end if
    end for
  end for
until no new subtree is added to A'
return A'

Termination of this algorithm is ensured by the following property, proved by induction on the structure of A: the set of subtrees of $completion(A, \mathcal{R})$ is, modulo state renaming, the set of subtrees of A. The repeat-until loop only inserts subtrees that were already present in A modulo state renaming, and thus terminates, since there are only a finite number of them and it never inserts twice the same.

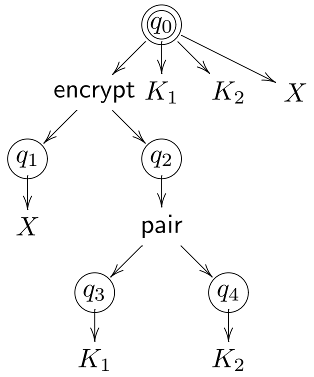
We then define

$$\mathcal{K}^\sharp(A) = completion(A^{\mathcal{O}}, \mathcal{R})$$

where $A^{\mathcal{O}}$ is A where the flag on the initial state has been set to true.



(a) Before the completion. The dashed subtree is an expansion of paths going through the loops on q_0, for the sake of clarity. The dotted line is the ε-transition we are adding.



(b) After the completion. Not to use real ε-transitions, we add the children of q_1 to the children of q_0.

Figure 3: Completion of the automaton from Fig. 2 by the rewrite rule $\mathsf{decrypt}(\mathsf{encrypt}(x, k), k) \to x$.

4 Abstract Model

The above concrete model has an annoying feature that makes it difficult to analyze: the infinite nondeterminism of the intruder (the knowledge of the intruder is an infinite set). We suppress that difficulty by "folding" together all branches of the nondeterminism of the intruder. This approximation is safe, in the sense that it always overestimates what the intruder knows. What then remains is a system of bounded nondeterminism, corresponding to the various possible interleavings of the principals. As the number of principals is finite, that gives a finite state space (although the number of interleavings grows fast with the number of principals).

4.1 Abstract Semantics

An abstract global state $S^\sharp \in \Sigma^\sharp$ is made of a tree automaton $S^\sharp.I$ representing the knowledge of the intruder, and the local states $(S^\sharp.p)_{p\in\mathcal{P}}$. Each local state $S^\sharp.p$ is made of a program sequence $S^\sharp.p.P$, with the same definition as in the concrete semantics, and a family $(S^\sharp.p.r)_{r\in r_p}$ of automata.

We define the semantics of the system by a nondeterministic transition relation $\to^\sharp$. Let $S^\sharp$ and ${S'}^\sharp$ be two global states. The definition of the transition relation is the following: $S^\sharp \to^\sharp {S'}^\sharp$ if there exists $p_0 \in \mathcal{P}$ so that:

- for all $p \in P$ so that $p \neq p_0$, ${S'}^\sharp.p = S^\sharp.p$;
- $S^\sharp.p_0.P = h :: \tau$ and either
  - $h = ?r_0$ and
    - for all $r \in R_{p_0}$ so that $r \neq r_0$, ${S'}^\sharp.p_0.r = S^\sharp.p_0.r$,
    - ${S'}^\sharp.p_0.r_0 = S.I$
    - ${S'}^\sharp.p_0.P = \tau$
  - $h = !t$ and
    - for all $r \in R_{p_0}$ so that $r \neq r_0$, ${S'}^\sharp.p_0.r = S^\sharp.p_0.r$,
    - ${S'}^\sharp.I = \mathcal{K}^\sharp(S^\sharp.I \cup t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])$
    - ${S'}^\sharp.p_0.P = \tau$
  - $h = r \simeq t$ and either
    - $\mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}]) \neq \emptyset$ then
      - for all $\bar{r} \in \bar{R}_{p_0} \setminus FV(t)$, ${S'}^\sharp.p_0.r = S^\sharp.p_0.r$
      - for all $\bar{r} \in FV(t)$, ${S'}^\sharp.p_0.r = \cup\{M.\bar{r} \mid M \in \mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])\}$[4]
      - ${S'}^\sharp.p_0.P = \tau$
    - $\mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}]) = \emptyset$; then
      - for all $r \in R_{p_0}$, $S'\sharp.p_0.r = S\sharp.p_0.r$
      - ${S'}^\sharp.p_0.P = \varepsilon$

[4] Replacing this condition by

$$\exists M \in \mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])\ \forall \bar{r} \in FV(t)\ {S'}^\sharp.p_0.r = M.r$$

yields a less coarse abstract model, which still has the good property that nondeterminism is finite and traces length are bounded. The model we use is clearly an abstraction of this more precise model.

4.2 The Abstraction Relation

We define an abstraction relation $a \subseteq \Sigma \times \Sigma^\sharp$: for any S in Σ and $S^\sharp$ in $\Sigma^\sharp$

$$a(S, S^\sharp) \Leftrightarrow (S.I \subseteq L(S^\sharp.I)) \wedge \forall p \in \mathcal{P}\ (S.p.P = \varepsilon) \vee \begin{cases} S.p.P = S^\sharp.p.P \\ \forall r \in R_p\ S.p.r \in L(S^\sharp.p.r). \end{cases}$$

The correctness of our method relies on the fact that $\rightarrow^\sharp$ is an abstraction of $\rightarrow$ with respect to a, according to the definition in part 1.2. We prove it by a straightforward, albeit tedious, case analysis on the different operations.

4.3 Where the Abstract and Concrete Models do not Coincide

As we are dealing with an approximate model, it is important to know how much information the model actually loses. There exists a simple example in which our abstraction strictly overestimates the power of the intruder: a single principal A runs that very simple program
$?r$
$!\mathsf{decrypt}(r, K)$
and the intruder initially knows $\{\mathsf{encrypt}(X, K); \mathsf{encrypt}(Y, K)\}$, X, Y and K being constants initially unknown to the intruder. We want to know whether at the end of the "protocol", the intruder can get hold of the concatenation of X and Y. Straightforwardly, this is impossible in the concrete model, since the intruder has to *choose* what it sends to A, and cannot send both X and Y. However, using the abstract model, we cannot get this conclusion.

Is this overestimation of the power of the intruder relevant when dealing when real-life protocols? Our investigations on examples of protocols found in classic papers on the topic [3] did not show it was a problem; the above kind of example is largely considered academic by the cryptographic protocol community. Furthermore, an error that exists only in the approximation for n principals could well be a concrete error for a greater number of principals. For instance, with the above example, if we run two copies of A, the intruder really can get X and Y. For these reasons, we think that the approximation is fine enough.

5 Implementation Issues

Basing ourselves on the above theory, we implemented a protocol analyzer. This program takes as input the signature and the rewrite system defining the term algebra and a specification of the protocol.

5.1 The Program

Our program reads an input file containing:

- the signature of the algebra, divided between "public" and "private" constructors; private constructors (like keys) cannot be applied by the intruder;
- the rewrite system;

- the initial knowledge of the intruder;
- what the intruder wants to get hold of (set L);
- the programs run by the principals.

It then explores the interleavings of the principal actions, computing with the abstract operations, and displays the interleavings that seem to exhibit a security hole (where the abstract knowledge of the intruder contains an element of L.

5.2 Interleavings

It is not necessary to consider all possible interleavings. We only consider interleavings that are concatenations of sequences of the following form: inputs and matches by a principal, and outputs by the same principal. It is easy to see that any interleaving is equivalent (when it comes to the final knowledge of the intruder) to such an interleaving. Our implementation therefore only explores the interleavings of that form. This drastically reduces the computation time.

5.3 Implementation of the Automata

We tried two implementations of the automata:

- One was closely based on the operations described above on special automata. Elementary operations, especially because of the use of hashed sets to test for identical branches, are very fast. The problem is that special automata have no minimization property and the size of automata grows fast as the length of the traces grows.
- The other one was operating on minimal deterministic finite tree automata [4]. Here, it seems that the completion by the rewriting system (implemented as the insertion of ϵ-transitions and the final determinization of the automaton) is very slow.

We also investigated whether some available toolkits such as MONA [12] and BANE [7], but didn't succeed in using any of those for our particular needs. The MONA application programming interface is geared towards WS2S logic applications and handling already computed automata is difficult; on the other hand, BANE is more geared towards computations on sets of terms, but it seemed that some useful features were either missing or difficult to implement without knowing the internals of the library. We are also considering other possible implementations based on constraint solving [2].

The experimental results we obtained suggest the replacement of the rewriting system by completion through a system of rules, which is computationally less expensive. This needs some slight changes in the semantics. An implementation is under way.

6 Experimental Results

We used the above implementation on some examples, some of which academic samples, some of them real protocols from the standard papers on the topic.

6.1 Trials on Small Examples

We first experimented our analyzer (computing on special automata) on some small examples, among which:

- a single run of the Otway-Rees protocol [3];
- the "Test n" examples: n principals running each the program: $?r$
  $\mathsf{decrypt}(r, K_n)$

  the initial knowledge of the intruder being $\mathsf{encrypt}(\cdots(\mathsf{encrypt}(X, K_1),$
  $\ldots, a$
  $lowbreakK_n)$; the unknown piece of data the intruder tries to recover being X.

Here are some recorded times and memory footprints:

Example	Pentium II 450 MHz	Sun Ultra 1	Memory used
Otway-Rees, 1 run	21 s	106 s	10 Mb
Test 6	3 s	11 s	1 Mb

Alas, while other protocols [3, 9], when using similarly small number of principals, have been easy to analyze using the program, bigger examples (like two parallel runs of the Otway-Rees protocol) have made the computation times explode.

6.2 An Interesting Point on the Otway-Rees Protocol

An early trial of our program on the Otway-Rees protocol [3] yielded surprising results. This protocol features a principal A running:

$$!\mathsf{pair}(A, \mathsf{pair}(B, \mathsf{pair}(M, \mathsf{encrypt}(\mathsf{pair}(N_a, \mathsf{pair}(M, \mathsf{pair}(A, B))), K_{as}))))$$
$$?r$$
$$r \simeq \mathsf{pair}(B, \mathsf{pair}(A, \mathsf{encrypt}(\mathsf{pair}(N_a, \bar{k}_{ab}), K_{as})))$$
$$!\mathsf{encrypt}(X, k_{ab})$$

The secret piece of data is X. After these four steps, the intruder can indeed get X in the following way: at step 2, the intruder sends $\mathsf{pair}(B, \mathsf{pair}(A, \mathsf{encrypt}(\mathsf{pair}(N_a, \mathsf{pair}(A, B)), K_{as})))$, built from pieces of the message output by A at step 1. A will then use $\mathsf{pair}(A, B)$ as k_{ab}. On the other hand, reorganizing the output from step 1, replacing $\mathsf{pair}(N_a, \mathsf{pair}(M, \mathsf{pair}(A, B)))$ by $\mathsf{pair}(\mathsf{pair}(N_a, M), \mathsf{pair}(A, B))$, prevents this attack, and the analyzer then concludes that the protocol is safe.

Whether or not the bug described above is relevant in real implementations depends on how certain primitives, notably pairing, are implemented. Models taking associativity and commutativity into account could perhaps be more suitable for analyses of such properties.

7 Conclusions and Prospects

We proposed a model based on tree automata to abstract cryptographic protocols. We implemented our algorithms and were able to successfully and correctly analyze some small instances (2 principals and 1 server) of well-known protocols and test examples. Our abstraction is fine-grained enough to yield successful result on real-life protocols.

On the other hand, the complexity of the computation quickly rises as the number of simulated machines grows. It also seems difficult to accommodate well certain properties like associativity or commutativity of operators. A more annoying drawback of our current concrete model is that the number of sessions and principals is fixed. We intend to investigate how to extend our approach to a model allowing an arbitrary number of sessions to be created, such as the spi-calculus [1].

It seems that the inefficiencies in our present implementation are essentially caused by the need to work with the rewriting system. We are currently working on a related semantics removing the need of such system (replacing it by pattern matchings) and we hope that performance will then be more adequate.

References

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. Research report 149, Compaq Systems Research Center, Palo Alto, CA, USA, Jan 1998.

[2] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[3] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Equipment Corporation, Systems Research Centre, February 1989.

[4] Hubert Comon, Max Dauchet, Remi Gilleron, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available through the WWW. In preparation.

[5] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 mars 1978.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.

[7] Manuel Fähndrich. *BANE: Analysis Programmer Interface.* Computer Science Departmene, University of California at Berkeley, 1998.

[8] Li Gong. *Cryptographic Protocols for Distributed Systems.* PhD thesis, University of Cambridge, Cambridge, England, April 1990.

[9] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about belief in cryptographic protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 234–248, Oakland, California, May 1990. IEEE Computer Society, IEEE Computer Society Press.

[10] Jean Goubault-Larrecq. Clap, a simple language for cryptographic protocols. available on the WWW, 1999.

[11] Jean-Pierre Jouannaud and Nachum Dershowitz. Rewrite systems. In Jan van Leuween, editor, *Handbook of Theoretical Computer Science, volume B*. Elsevier, The MIT Press, 1990.

[12] Nils Klarlund and Anders Møller. *MONA version 1.3: User Manual*. BRICS, University of Aarhus, 1998.

[13] G. Lowe and B. Roscoe. Using CSP to detect Errors in the TMN Protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

[14] W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-SCS-97-139, Carnegie Mellon University, May 1997.

[15] Catherine Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 1995. To appear.

[16] Jonathan K. Millen. CAPSL: Common authentication protocol specification language. available on the WWW.

[17] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *IEEE Symp. Security and Privacy*, pages 141–153, Oakland, 1997.

[18] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[19] P. Syverson. Adding time to a logic of authentication. In *1st ACM Conference on Computer and Communications Security*, pages 97–101, 1993.

[20] P. Syverson and P. C. van Oorschot. On unifying some cryptographic protocol logics. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28, May 1994.