Abstracting Cryptographic Protocols with Tree Automata *

David Monniaux

http://www.di.ens.fr/%7Fmonniaux,
Laboratoire d'Informatique, École Normale Supérieure,
45 rue d'Ulm , 75230 PARIS cédex 5, FRANCE

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 approach, based on abstract interpretation and 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.

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 (often called **principals**) 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. Cryptographic protocols in everyday use on personal machines include

* This work was partially done at SRI International, Computer science laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, USA, under NSF grant CCR-9509931.

the Secure Socket Layer (SSL), used for secure World-Wide-Web transactions, and Secure Shell (SSH), used for the secure remote control of machines.

An intruder having gained partial or total control over the communication network may launch an attack on the protocol. We distinguish two classes of attacks: **passive attacks**, where an intruder passively listens to the communication network, and **active attacks** where the intruder can suppress messages and send its own messages. On the practical side, it should be noted that passive attacks on local area networks are generally very simple to mount, just requiring running adequate software on a machine connected to the network, which will then listen to the network and log any data packet transmitted. Active attacks are more elaborate and difficult to mount, since in their primitive setting ("man in the middle attack") they involve cutting a running network line and plugging an intruding machine in the middle. Yet active attacks can be performed with less intrusive means such as "DNS spoofing" [26], which makes a machine obtain a wrong Internet Protocol address for a remote host; the machine then communicates with an intruder believing it is actually the remote server it meant to communicate with. It is therefore important to consider both passive and active attacks when analyzing the security of a cryptographic protocol. Our analysis thus considers the worst case, where the intruder is able to read or suppress any messages sent on the network and forge any message that it can realistically concoct.

Many analyses techniques for cryptographic protocols have been proposed; the next sub-section will give a short survey of them. 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. Even with those restrictions, many protocols have been shown to be flawed.

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 for confidentiality and other safety properties. To our knowledge, this is the first time that an abstract domain has been proposed for cryptographic protocols.

Our method tests safety properties (i.e. that the protocol cannot reach certain undesirable states). An obvious safety property, and the only one that is implemented in our analyzer, is secrecy: we prove that the intruder cannot receive certain information. Other safety properties that can be dealt with using our method include some variants of authentication: we can prove that some principal can reach an "accepting" state only if some data it holds is the correct one.

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 Comparison to Related Works

Burrows, Abadi and Needham proposed to analyze cryptographic protocols using a logic of belief, now known as the "BAN logic" [10]. Several derivatives of this logic have followed [22,21,42,43]. All those systems provide a means to formalize the high-level reasoning that stands behind the protocols. Such informal reasoning often uses steps such as "Principal B receives a message signed with K_a, and K_a is a secret key only owned by A; therefore, this message must have been emitted by A at some point in the past.". The goals of those systems is to make such proofs more rigorous.

There are two important problems with logics of belief. The first is that it is essentially a manual approach. While there exist tools [38,28,16,40] which automatically check whether the purported conclusion follows from the hypotheses of the protocol description using the rules in the logic, the formalization of the reasoning of the protocol inside the logic is a manual, cumbersome and error-prone act, even though some of the available decision procedures can be modified so as to provide some hints as to missing or incorrect hypotheses to the user [38]. The second is that most of those logics do not have clear semantics. Some semantics have nevertheless been provided for some logics of belief [4], including some more recent work on strand spaces [44], yet designing a sound and usable semantics for those logics remains largely a work-in-progress.

In contrast with these logics, our method is fully automatic and operates on the actual definition of the protocols, not on some difficult-to-establish justification of them. Furthermore, our analysis is proven to be sound with respect to some simple formal semantics of the protocols.

The limitations of logics of belief clearly show a need for analyses based on the actual executions of the protocol, and not on some justification of it. Dolev and Yao proposed a formal model of cryptographic protocols where cryptographic primitives (encryption, decryption, signature...) obey a set of algebraic properties (such as: decrypting some encrypted piece of data using the same key with which it was encrypted yields the original, unencrypted piece of data). Our analysis is set in a particular formalization of the Dolev-

Yao model; it expresses the protocols and the properties of the primitives in a simple language and gives them a precise semantics.

Many analyses of protocols have been expressed within the Dolev-Yao model. The first ones used Prolog or Prolog-like state exploration techniques [35]. Later contributions have focused on implementing state-space exploration techniques model checkers, whether general-purpose [30,32] or special-purpose [41]. While often efficient at finding bugs, those approaches often cannot guarantee that there exists no sequence of actions from the intruder that can exhibit the faulty behavior. To work around this limitation in some cases, some criteria for the completeness of a finite state space have been proposed [31]. Alternatively, one can supplement the automatic search with inductive proofs [34].

Summarily, those approaches explore a subset of all possible attacks in the model, and thus cannot yield sound security proofs without some additional work. On the other hand, they can yield an actual trace of attack. In contrast, our method explores a superset of all possible attacks and thus can yield secrecy or other safety proofs.

More recently, several symbolic approaches to the analysis of cryptographic protocols have been proposed, among which ours [37] and Jean Goubault's [25], both based on regular sets of terms represented by automata while some others [20,8] consider other similar symbolic representations. These methods aim at circumventing the limitations of model-checking by using finite representations of infinite sets, coupled with some safe approximations (i.e. the analysis somehow overestimates the power of the adversary).

Another model for cryptographic protocols is Abadi and Gordon's spi-calculus [6,3], an extension to cryptographic protocols of the π-calculus, a process algebra. Several type systems suitable for proving security properties have been proposed [1,2,5]. These type systems allow proving that the data-flow in the protocol is secure, that is, that no sensitive information can flow to insecure channels. However, they restrict the class of protocols that can be considered and require extensive type annotations and manual derivation of typing proofs. This approach is certainly powerful since it can deal with very complex communication networks and protocols; on the other hand, those analyses are essentially manual while ours is automatic.

1.3 Abstract Interpretation

Abstract interpretation [14,15] is a generic theory for the analysis of computation systems. Its basic idea is to use approximations in ordered domains in

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

Fig. 1. The abstract transition relation follows the concrete one.

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 $\{x \in \sigma \mid \exists x^\sharp \in X^\sharp \; a(x, x^\sharp)\}$.

For instance, Σ could be $\wp(\mathbb{Z})$, where $\wp(X)$ notes the power-set of X, 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 here 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 Informal Semantics

The first difficulty in formal analyses of cryptographic protocol is the choice of the formalism in which to express the protocol and a suitable semantics.

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

Although specification languages such as CAPSL [36] or CLAP [24] and other notations [11] have been developed, it is difficult to find a model for which analysis is tractable and that does not introduce intolerable inaccuracies.

In this section, we shall give an informal account of our version of the Dolev-Yao model, as well as a description of the input syntax of our implemented analyzer.

2.1 Our version of the Dolev-Yao Model

The most common approach to analyzing cryptographic protocols is to abstract away the actual cryptographic primitives(public or private key encryption, hash functions...) being used. Those primitives are only characterized by some algebraic properties reflecting their ideal behavior. Our analysis scheme must then base its deductions on those properties.

For instance, we consider a symmetric encryption primitive $\texttt{encrypt}(X, K)$ taking a piece of data X and a key K. Symmetric key encryption means that the same key that has been used for encrypting the message can be used to decrypt it. This means that if an intruder owns $\texttt{encrypt}(X, K)$ and the key K, then it can obtain X. Furthermore, our encryption is supposed to be strong enough so that the only way to get X from $\texttt{encrypt}(X, K)$ is to own the key K.

This example suggests that we should consider the messages exchanged in the analyzed system to be terms over a signature consisting of all the primitives. For instance, a protocol making use of symmetric encryption is modeled with messages built on the following algebra: $\texttt{encrypt}(\cdot, \cdot)$ (encryption) and $\texttt{pair}(\cdot, \cdot)$ (pairing).

Such models of cryptographic protocols have been introduced by Dolev and Yao [11]. The specification for a protocol then contains:

- the naming and arities of primitives
- the properties of those primitives
- the protocol steps.

Our analysis method restricts the class of properties that can be expressed and the transition rules for the protocol steps.

It has to be noted that even with those restrictions, it is possible to specify a protocol whose security is undecidable [11, §4.2.1].

2.2 *Primitives (CONSTRUCTORS clause)*

Here is an example of a specification of primitives:

```
CONSTRUCTORS
{
  pair(P,P);
  proj1(N);
  proj2(N);

  encrypt(P,N);
  decrypt(P,N);

  private Na;
}
```

Primitives may be specified either as public (no keyword) or private (`private` keyword). A public primitive can be freely used by the intruder ; for instance, the encryption primitive may freely be used by the intruder, since we assume that he knows which cryptographic primitives we use and that he has a working implementation of them. This is consistent with Kerckhoffs' rule that the secrecy must reside entirely in the key (not in the secrecy of the encryption mechanism or algorithm) [39, 1.1]. On the other hand, private primitives reflect informations or operations that the intruder does not know : for instance, some secret constants are specified as 0-ary primitives.

The letters `P` and `N` in the arity are used to implement some elementary sanity check for the actions of the principals. A principal may only use pattern matching to extract patterns from fields designed by `P` (pattern), not from fields designed by `N` (no pattern). Indeed, if a principal knows a key `K`, it can match an incoming piece of data against `encrypt`$(x,$`K`$)$ and extract x, but it would be absurd for it to match an incoming piece of data against `encrypt`$($`X`$,k)$ and extract the key k. Those letters have no actual role in the analysis, they are just meant to exclude some obvious errors in the specification in the front-end of the analyzer.

2.3 *Rewriting Rules (RULES clause)*

In our case, the rules are :

```
proj1(pair(x,y)) => x;
proj2(pair(x,y)) => y;
decrypt(encrypt(x, k), k) => x;
```

Those rules mean respectively :

- that the intruder may freely project the left side of a pair it owns ;
- that the intruder may freely project the right side of a pair it owns ;
- that the intruder may decrypt an encrypted message it owns, provided it also owns the corresponding key.

It is possible to model several encryption systems in the same protocol : one only has to specify several primitives `encrypt1`, `encrypt2` etc... It is also possible to model a public-key cryptosystem as follows :

```
publicDecrypt(publicEncrypt(x, k), privateKey(k)) => x
```

where `privateKey` is a private unary constructor. When x is a public key, `privateKey`(x) is the corresponding private key. Of course, `privateKey` is a `PRIVATE` constructor.

2.4 The Intruder; Completion Rules

All communication goes through the intruder. When a principal outputs a message, it sends it to the intruder. When a principal inputs a message, it gets it from the intruder. This model thus assumes that the intruder has entire control of the communication network and can modify or concoct messages at will. Of course, it is impossible to prove that a protocol completes its task in such a model, since the intruder can just stop transmitting any message. On the other hand, we can establish properties such as the secrecy of some piece of information, or whether principals always fail before reaching a certain point of the protocol if fed improper information by the intruder.

The intruder is limited in its computations. It has a set of initially known pieces of data, and enriches that set as it receives messages. It completes that set of messages using :

- the publicly available primitives (§2.2),
- the rewriting rules (§2.3).

2.5 The initial and secure knowledge specifications (`INTRUDER_KNOWLEDGE` and `SECRETS` clauses)

The initial knowledge of the intruder is specified in the `INTRUDER_KNOWLEDGE` clause (actually, the initial knowledge of the intruder is the completion of the knowledge specified by the intruder by the rules — see §2.4).

2.6 Principals (`PRINCIPAL` clauses)

Each principal runs its part of the protocol. The usual notation is to identify principals by a letter such as A or B, or S for servers. If there are several instances of the protocol running concurrently or successively, there are several instances ($A_1, A_2, \ldots,\ B_1, B_2, \ldots$) of each of the principals described in the protocol description; alternatively, the same principal may only have one instance (a single server may serve several sessions). For instance, protocols featuring a server are generally modeled with a common server principal for all sessions, while the other principals have one instance per session. Such distinction between instances of the same principal is important if they are supposed to generate unique identifiers or random numbers in each instance (see 2.7).

2.6.1 Registers (`REGISTER` clause)

Each principal has a finite number of registers, each containing no value or a term. For the sake of clarity, registers are given names reflecting the data a correct execution of the protocol should store in that register. For instance, `kab` is the name of a register meant to contain the communication key between principals A and B. Semi-formal specifications of protocols often call with the same identifier (say K_{ab}) two different things:

- an unique constant generated at run-time for each session of the protocol,
- registers in one or more principals meant to contain that constant.

Of course, while in a correct run of the protocol those registers indeed contain that constant after some point of the protocol, an intruder might well concoct messages so that some principal "thinks" some value is a correct communication key, store it into the appropriate register and then use it as a legitimate key. It is therefore important to distinguish between those two meanings.

2.6.2 Specification of actions (`PROGRAM`)

A principal may at a given moment, depend on the current instruction it has to execute:

- Input a message into a register (`?`*register*). The message is actually input from the intruder (who, in our model, controls the network) ; the message obtained from the intruder can be any piece of data that the intruder can compute (§2.4).

  For instance, if the intruder initially knew `X` and received `Y` from a principal, it can compute `pair(X,Y)` and send it to the principal that executes

`?r`. That principal will then store `pair(X,Y)` into register `r`.
- Output a message, made from constants and register contents (`!`*message*). In our notation, the message is a term containing constructors, constants and register names. The principal computes the actual message by replacing the register names by the current values of the registers. The message is then output to the intruder, which uses it to increase its knowledge.

  For instance, if register `data` contains `X` and register `kab` contains `Kab`, then `! encrypt(data, kab)` will output `encrypt(X, Kab)` to the intruder.
- Match the contents of a register against a pattern (*register* `=~` *pattern*). The pattern can specify constants, current contents of registers (*register*) or destination registers (`$`*register*). If the pattern fails, the principal considers that the protocol has failed and stops.

  This action is complex to describe formally, but really simple informally. Let us suppose that register `r` contains `encrypt(X, Kab)` and that register `kab` contains `Kab`. After the principal execute `r =~ encrypt($data, kab)`, register `data` will contain `X`. If `r` had contained `encrypt(X, Kas)`, the match would have failed and the principal would have stopped executing.

The execution of the protocol steps is sequential. If a match step fails, the principal ceases to execute the protocol (*fail-stop* behavior [23]).

The behavior of all principals is interleaved: at a given time, any principal ready to execute an action can execute it. This leads to many possible interleaving of actions.

2.7 Nonces and key generation

Many protocols call for the use of *nonces*; a nonce is a number that is generated once for each session of the protocol. A nonce can be a session key or any number that is unique to a session. A nonce is supposed to be *fresh*; that is, it has not been used in a previous session of the protocol (or at least it is very unlikely that is has been used before).

We model nonces by unique constants that cannot be unified with any other constant. If a nonce is unique to a session, we use one constant per session. So, for instance, if a server S generates a key K_{ab} then we will use a constant `Kab` to represent the key.

2.8 Limitations of this model

There are pitfalls to this approach:

(1) Some properties of the primitives may have been ignored. For instance, some primitives are malleable [17]; for instance, an intruder can transform some encrypted text $\texttt{encrypt}(X, K)$ into some other text $\texttt{encrypt}(Y, K)$ without knowing the key K. It would be necessary to reflect such behavior in the rules describing the primitives.
(2) Such a model do not take probabilistic leaks of information into account [9]. We do not take into account attacks using statistical inference.
(3) Our model has "subliminal channels". It is possible to hide information in complex ways, such as encrypting something n times with the same key and using the result as a means to transmit n. We do not handle such methods of transmission, which anyway are not used in most protocols.
(4) Representing concatenation with pairing is not very accurate. In real-life protocols, concatenation is associative. Furthermore, a real-life intruder might split data in ways that are not allowed by this model, such as taking only one byte of a 32-bit value.

    As we shall see later, it is difficult to accurately model associativity properties in term algebras. Our implementation therefore does not take associativity into account. Even with this limitation, our method found a "bug" in a well-known protocol (§7.2). Whether or not the bug exists in the formal model depends on how n-tuples have been split.

2.9 An example: the Otway-Rees protocol

In this section, we shall see on a simple example (the Otway-Rees protocol [10, §4]) how to model a protocol in our model. We shall again use the input syntax of our simple analyzer.

In the cryptography literature, the Otway-Rees protocol is described as follows:

Message 1	$A \to B$	$M; A; B; \{N_a; M; A; B\}_{K_{as}}$
Message 2	$B \to S$	$M; A; B; \{N_a; M; A; B\}_{K_{as}}; \{N_b; M; A; B\}_{K_{bs}}$
Message 3	$S \to B$	$M; \{N_a; K_{ab}\}_{K_{as}}; \{N_b; K_{ab}\}_{K_{bs}}$
Message 4	$B \to A$	$M; \{N_a; K_{ab}\}_{K_{as}}$

$\{X\}_K$ is a notation for "X encrypted with the key K". The following paragraphs will explain the other notations.

The Otway-Rees protocol features three principals: one server S and two clients A and B. A and B each have a shared key, K_{as} and K_{bs} respectively, for communication with the server S. Those two clients wish to communicate together using a shared key K_{ab}, on which they have to agree. They use S as a middleman so as to agree on a shared secret key K_{ab}. Of course, it is hoped that when they do so, no intruder can get hold of that key K_{bs} (**secrecy**) and that both machines, if completing the protocol successfully, get the correct key K_{bs} (correct key distribution). We suppose of course that both keys K_{as} and K_{bs} are kept secret from the intruder.

This protocol, as many others, makes use of **nonces**, also known as **counfounders**. These are random pieces of data that are generated for each run of the protocol by some of the principals. They are meant to foil some attacks such as replaying parts of an older transaction (**replay attacks**); here, principal A generates nonces M and N_a and principal B generates nonce N_b.

As said in §2.7, we model both nonces and keys using private constants (i.e. 0-ary constructors that are not supposed to be initially known by the intruder and are not guessable by him). There is, however, a distinction to be made between two meanings for the same notation in the above description of the protocol:

- in some cases, a name (say, "N_a") is a constant initially known by the principal that emits the message; it is therefore to be treated as a symbolic constant `Na`;
- in other cases, the same name N_a denotes a value that the principal has previously acquired from the network and stored into a register (called `na`); in a normal (without intrusion) run of the protocol, this value should be equal to `Na`.

We model only one run of the protocol; that is, we simulate only three machines A, B and S. n simultaneous multiple runs of the protocol with the same server could be simulated by considering machines $A_1, \dots, A_n$ and $B_1, \dots, B_n$.

```
CONSTRUCTORS
{
  pair(P,P);
  proj1(N);
  proj2(N);

  encrypt(P,N);
  decrypt(P,N);

  private Kas;
  private Kbs;
  private Kab;
  private M;
  private Na;
  private Nb;
  private X;
  private A;
  private B;
  S;
}

RULES
{
  proj1(pair(x,y)) => x;
```

```
  proj2(pair(x,y)) => y;
  decrypt(encrypt(x, k), k) => x;
}

INTRUDER_KNOWLEDGE
{}

SECRETS
{ X
}

PRINCIPAL A
{
  REGISTERS { r, kab }
  PROGRAM {
    ! pair(A, pair(B, pair(M, encrypt(pair(pair(Na, M), pair(A, B)), Kas))));
    ? r;
    r =~ pair(B, pair(A, encrypt(pair(Na, $kab), Kas)));
    ! encrypt(X, kab);
  }
}

PRINCIPAL B
{
  REGISTERS { r, m, a, z, kab }
  PROGRAM {
    ? r;
    r =~ pair($a, pair(B, pair($m, $z)));
    !pair(B, pair(S, pair(a, pair(z, encrypt(pair(pair(Nb, m), pair(a, B)), Kbs)))));
    ? r;
    r =~ pair(S, pair(B, pair($z, encrypt(pair(Nb, $kab), Kbs))));
    ! pair(B, pair(a, z));
  }
}

PRINCIPAL S
{
  REGISTERS { m, r, a, b, na, nb }
  PROGRAM {
    ? r;
    r =~  pair($b, pair(S, pair($m, pair($a, pair($b, pair(
      encrypt(pair(pair($na, $m), pair($a, $b)), Kas),
      encrypt(pair(pair($nb, $m), pair($a, $b)), Kbs)))))));
    ! pair(S, pair(b, pair(m, pair(
      encrypt(pair(na, Kab), Kas),
      encrypt(pair(nb, Kab), Kbs)))));
  }
}
```

3 Formal Semantics

In this section, we give a formal semantics to the cryptographic protocols, corresponding to the informal semantics presented in the preceding section. We shall later prove the correction of our analysis with respect to this formal semantics.

3.1 Terms and Completion Rules

Let us consider a signature [27, p. 249] [13, preliminaries] $\mathcal{F}$ and the **free algebra** of **terms** $T(\mathcal{F})$ on that signature. A signature is simply a couple (Σ, a) where Σ is a set of **function names** and a is a function from Σ to the set of nonnegative integers $\mathbb{N}$ called the **arity**. The function names in the example of §2.9 are `pair`, `encrypt`, each of arity 2, and various constants (`Na`) of arity 0. The free algebra $T(\mathcal{F})$ of terms upon the signature is just the set of syntactic terms built using those function names (for instance, `encrypt(Na, Kas)`).

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, $(t_i)_{i \in I}$ is a family of terms, we write $t[t_i/X_i]$ the term obtained by parallel substitution of X_i by t_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 $\mathcal{K} : \wp(T(\mathcal{F})) \to \wp(T(\mathcal{F}))$ that computes the closure of a subset of $T(\mathcal{F})$ by the following operations:

- 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 rules 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: t`=>`x, where t is a term with variables and x is a variable appearing exactly once in t (**collapsing rules**).

3.2 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 $T(\mathcal{F}) \cup \{\Omega\}$ — the Ω element meaning "uninitialized" — and a program x_p to execute. We shall actually ignore uninitialized elements, since protocols using uninitialized elements do not make real sense; our analysis therefore refuses protocols that make use of uninitialized registers using a simple syntactic check. The program is a finite sequence (possibly empty) of commands, which can be of the three possible types:

Output to network $!t$, read as "output t", where $t \in T(\mathcal{F}, R_p)$;

Pattern matching r_0 `=~` t, read as "match register r_0 against t", where $r_0 \in \{1, \ldots, r_p\}$ and $t \in T(\mathcal{F}, R_p \cup \$R_p)$; $\$R_p = \{\$r \mid r \in R_p\}$ is a copy of R_p obtained by prefixing each register name by a \$ sign.

The difference of meaning between r and $\$r$ is as follows:

- r at position π in the term notes the current value of register r; the sub-term at position π of the value in register r_0 is matched against the current value in register r;
- $\$r$ notes that the sub-term at position π of the value in register r_0 should be stored into register r.

Input from network $?r$, 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 $\to$ (nondeterminism arises because of the interleavings of the actions of the principals and because of the choices of the adversary). 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

  (Input from network) $h = ?r_0$ and
  - for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$,
  - $S'.p_0.r_0 \in S.I$
  - $S'.p_0.P = \tau$

  (Output to network) $h = !t$ and
  - for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$ (this does not change the state of any register),
  - $S'.I = \mathcal{K}(S.I \cup \{t[S.p_0.r/r \mid r \in R_{p_0}]\})$ (the intruder now knows t and uses it to increase its knowledge);
  - $S'.p_0.P = \tau$ (the principal p_0 has yet to execute the rest of its program).

  (Pattern matching) $h = r_0$`=~`t and either
  - there exists a substitution $\sigma_1 : \$R_p \to T(\mathcal{F})$ so that $r_0 = t[\sigma_1; \sigma_2]$ where σ_2 is the substitution $r \mapsto S.p_0.r$ and $t[\sigma_1; \sigma_2]$ is the result of the application to the term t of both substitutions σ_1 and σ_2; then

    for all $r \in R_{p_0}$ so that $\$r$ is not a free variable in t, $S'.p_0.r = S.p_0.r$ (registers not meant to receive results from the pattern

are left untouched);
for all $r \in R_{p_0}$ so that $\$r$ is a free variable in t, $S'.p_0.r = \sigma_1(r)$ (registers meant to receive results from the pattern get the desired results);
$S'_I = S_I$ (as there is no network communication involved, the intruder gets no new knowledge);
$S'.p_0.P = \tau$ (the principal p_0 has yet to execute the rest of its program);

· such a substitution does not exist; then
for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$;
$S'_I = S_I$ (as there is no network communication involved, the intruder gets no new knowledge);
$S'.p_0.P = \varepsilon$ (the principal considers that the protocol has failed for some reason and stops executing it — fail-stop behavior).

Please note that the pattern matching is not done modulo the rules defined in §3.1. This should not be a problem in most cases. Furthermore, it is possible to slightly complexify the semantics by requiring that the rewriting rules should be applied after each computation step. For the sake of brevity, we will not treat that extension in this paper.

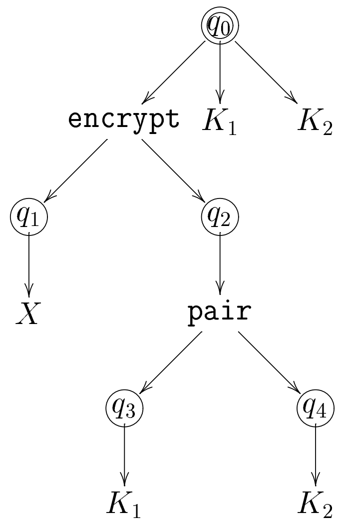
4 Tree Automata and Operations on Them

Regular languages, implemented as finite automata, are often used to abstract sets of words on an alphabet. Here, we abstract sets of terms on a signature by regular tree languages, and we consider the generalization of finite automata to n-ary constructors: tree automata [13].

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.

4.1 Special Tree Automata

We use non-deterministic top-down tree automata [13, §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



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

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

(b) The set Δ of rewrite rules.

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

rules [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:

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

where $n \geq 0$ $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n$ 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 $\mathcal{A}_\mathcal{F}$. A special automaton can be represented as 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;

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

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:

- $q \longrightarrow s \begin{array}{l} \nearrow q_1 \\ \quad \vdots \\ \searrow q_n \end{array}$ 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 $q \circlearrowleft \mathcal{O}$), 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.

A formal definition of the class of set automata is that the set Δ of rewrite rules defining a special automaton can be partitioned between two subsets:

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

4.2 Union

The union of two languages representable using special automata A and B is represented by the special automaton $A \sqcup B$ obtained by joining A and B at the root.

4.3 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 any 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 $t = s(t_1, \dots, t_n)$ where s is an n-ary symbol, then

$$\begin{aligned} match_l(A,t) = \{&\lambda x \in \mathcal{X}\{\cup\{p_i \mid \forall 1 \leq i \leq n\ p_i \in match_l(\langle Q, q_i, \Delta\rangle, t_i)\} \\ &\mid r : q(s(x_1, \dots, x_n)) \to_A s(q_1(x_1), \dots, q_n(x_n)) \in \Delta\}; \end{aligned}$$

- 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 $x_1, \dots, x_n$ the variables in t, for all terms $t_1, \dots, t_n \in T(\mathcal{T})$, then $t[t_i/x_i, \dots, t_n/x_n] \in L(A)$ 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.

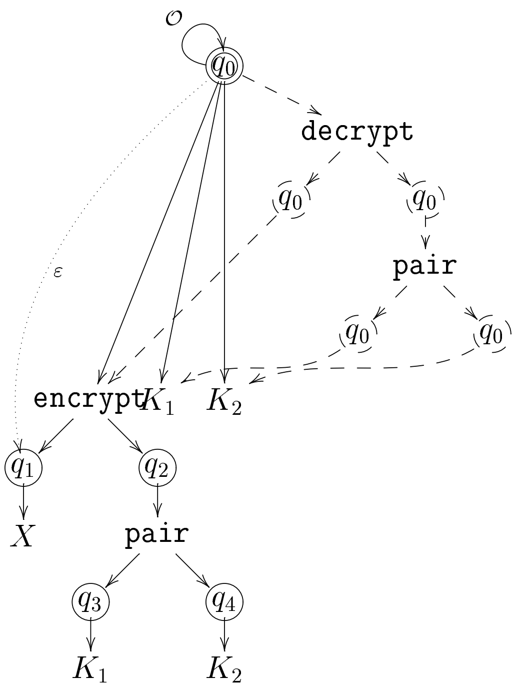
We then construct a function *match* that has the same property, except that it does not constrain the terms to be linear.

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

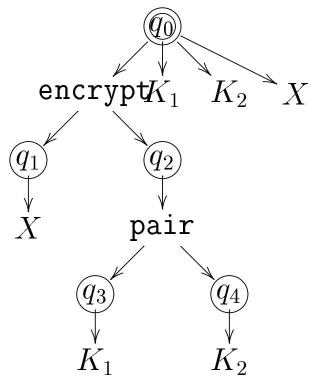
The definition of $match_l$ translates into an algorithm on automata defined by trees as above. Then *match* is defined, using an effective test of whether the languages of several automata intersect [13, §1.7].

The above property of $match_l$ induces the following property on *match*:
Lemma 1. *Given a term t with variables and a special automaton A, if there exists a substitution $\sigma : FV(t) \to T(\mathcal{F})$ of the free variables of the term t so that $t[\sigma] \in L(A)$, then there exists $M \in match(A,t)$ so that for each variable x, $\sigma(x) \in L(M(x))$.*

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



(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.

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

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

We want an abstraction of the function $\mathcal{K}$; that is, 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 [27, 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.

Lemma 2. *For any special automaton A, $\mathcal{K}(L(A)) \subseteq L(\mathcal{K}^\sharp(A))$*

Proof. $L(\mathcal{K}^\sharp(A))$ contains necessarily $L(A)$, since all the transitions present in A are present in $\mathcal{K}^\sharp(A)$. Furthermore $L(\mathcal{K}^\sharp(A))$ is stable by $\mathcal{K}$, since the completion algorithm terminated. Since $\mathcal{K}(L(A))$ is the least fixpoint of $\mathcal{K}$, the inclusion follows. □

5 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).

Informally, our abstract works as follows: instead of using an infinite set of terms to model the knowledge of the intruder, we use a finite representation of an infinite set of terms; instead of individual terms stored in principals' registers, we use finite representations of set of terms that could possibly be held in those registers. A single abstract state then stands for all the concrete states where the contents of the principals' registers belong to the respective abstract sets and where the intruder's knowledge is a subset of the respective abstract state.

5.1 The Abstract Domain and the Abstraction Relation

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.

A single abstract state may represent (infinitely) many corresponding concrete states. The links between abstract states and concrete states is defined by an abstraction relation (see §1.3).

Our abstraction relation $a \subseteq \Sigma \times Sigma^\sharp$: is defined as follows: for any S in Σ and $S^\sharp$ in $\Sigma^\sharp$

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

What this means intuitively is that we attach a regular set of terms, represented by a special automaton, to each part of the concrete state that stores a term. An abstract state is an abstraction of a concrete state if and only if all the terms of the concrete state belong to the respective sets of terms.

5.2 Abstract Semantics

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
  - $h = \texttt{?}r_0$ and
    for all $r \in R_{p_0}$ so that $r \neq r_0$, $S'^\sharp.p_0.r = S^\sharp.p_0.r$,
    $S'^\sharp.p_0.r_0 = S^\sharp.I$

$S'^\sharp.p_0.P = \tau$
$S'^\sharp.I = S^\sharp.I$

· $h = !t$ and
 for all $r \in R_{p_0}$ so that $r \neq r_0$, $S'^\sharp.p_0.r = S^\sharp.p_0.r$,
 $S'^\sharp.I = \mathcal{K}^\sharp(S^\sharp.I \sqcup t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])$
 $S'^\sharp.p_0.P = \tau$

· $h = r\texttt{=\~{}}t$, $S'^\sharp.I = S^\sharp.I$ and either
 $match(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}]) \neq \emptyset$ then
  for all $\$r \in \$R_{p_0} \setminus FV(t)$, $S'^\sharp.p_0.r = S^\sharp.p_0.r$
  for all $\$r \in FV(t)$, $S'^\sharp.p_0.r = \sqcup\{M(\$r) \mid M \in match(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])\}$ [4]
  $S'^\sharp.p_0.P = \tau$
 $match(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}]) = \emptyset$; then
  for all $r \in R_{p_0}$, $S'\sharp.p_0.r = S\sharp.p_0.r$
  $S'^\sharp.p_0.P = \varepsilon$

5.3 Proof of Correctness

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 1.3. This means that our method computes symbolically a **superset** of the set of reachable states.

Let us now prove this correctness condition. Let us consider a concrete transition $S \rightarrow S'$ (as defined in §3.2) and an abstract state $S^\sharp$ so that $S^\sharp$ is an abstraction of S (as defined in §5.1). Let p_0 be the principal that executes the transition and $S.p_0.P = h :: \tau$ the program it has to execute.

Since $S^\sharp$ is an abstraction of S, for all $p \in \mathcal{P}$, for all $r \in R_p$, $S.p.r \in L(S^\sharp.p.r)$. For $p \neq p_0$, $S'.p.r = S.p.r$ and $S'^\sharp.p.r = S^\sharp.p.r$, thus $S'.p.r \in L(S'^\sharp.p.r)$. Let us now establish the remaining relations. There are several cases, depending on the instruction being executed:

(Input from network) $h = ?r_0$; then

- for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$; by the abstraction relation, $S.p_0.r \in L(S^\sharp.p_0.r)$ and thus $S'.p_0.r \in L(S'^\sharp.p_0.r)$;

---

[4] Replacing this condition by

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

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.

- $S'.p_0.r_0 \in S.I$; but $S'^\sharp.p_0 = S^\sharp.I$ and $S.I = L(S^\sharp.I)$, thus $S.p_0.r_0 \in L(S'^\sharp.p_0.r_0)$;
- $S'.p_0.P = \tau = S^\sharp.p_0.P$;

(Output to network) $h = !t$ and

- for all $r \in R_{p_0}$, $S'.p_0.r = S.p_0.r$; by the abstraction relation, $S.p_0.r \in L(S^\sharp.p_0.r)$ and thus $S'.p_0.r \in L(S'^\sharp.p_0.r)$;
- $S'.I = \mathcal{K}(S.I \cup \{t[S.p_0.r/r \mid r \in R_{p_0}]\})$; since $t[S.p_0.r/r \mid r \in R_{p_0}] \in L(t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])$ (Lemma 1) and $S.I \in L(S^\sharp.I)$ then $S.I \cup \{t[S.p_0.r/r \mid r \in R_{p_0}]\} \in L(S^\sharp.I \sqcup t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])$, and thus, using Lemma 2, $S'.I \in L(S'^\sharp.I)$.
- $S'.p_0.P = \tau = S^\sharp.p_0.P$;

(Pattern matching) $h = r_0$=~t and then either:

- there exists a substitution $\sigma_1 : \$R_p \to T(\mathcal{F})$ so that $r_0 = t[\sigma_1; \sigma_2]$ where σ_2 is the substitution $r \mapsto S.p_0.r$;
  - for all $r \in R_{p_0}$ so that $\$r$ is not a free variable in t, $S'.p_0.r = S.p_0.r$; since in this case $S'^\sharp.p_0.r = S^\sharp.p_0.r$, it follows that $S'.p_0.r \in L(S'^\sharp.p_0.r)$;
  - for all $r \in R_{p_0}$ so that $\$r$ is a free variable in t, $S'.p_0.r = \sigma_1(r)$; but then there exists $M \in \mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])$ so that $\sigma_1(r) \in L(M(\$r))$ (Lemma 1) and thus a fortiori $\sigma_1(r) \in L(\sqcup\{M(\$r) \mid M \in \mathit{match}(S^\sharp.p_0.r, t[S^\sharp.p_0.r/r \mid r \in R_{p_0}])\})$, which means that $S'.p_0.r \in L(S'^\sharp.p_0.r)$;
- such a substitution does not exist; then $S'.p_0.P = \varepsilon$;

in any case,

- $S'.p_0.r_0 \in S.I$; but $S'^\sharp.p_0 = S^\sharp.I$ and $S.I = L(S^\sharp.I)$, thus $S.p_0.r_0 \in L(S'^\sharp.p_0.r_0)$;
- $S'.p_0.P = \tau = S^\sharp.p_0.P$;

5.4 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 $\{\mathtt{encrypt}(X, K); \mathtt{encrypt}(Y, K)\}$, 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.

Let us consider the concrete model. The intruder may send $\mathtt{encrypt}(X, K)$ or $\mathtt{encrypt}(Y, K)$, in which case it obtains respectively $\mathtt{decrypt}(\mathtt{encrypt}(X, K), K) =$

X or $\texttt{decrypt}(\texttt{encrypt}(Y, K), K) = Y$ after the second step, but it may not do so at the same time. The intruder has to choose whether it wants to get X or Y. It is not possible for it to get both X and Y, and it may therefore not compute the concatenation $\texttt{pair}(X, Y)$.

Let us now execute the abstract analysis by hand, step-by-step. After the first step, we know that register r may contain any of $\{\texttt{encrypt}(X, K); \texttt{encrypt}(Y, K)\}$. After the first step, the abstract intruder knowledge gets augmented by $\{\texttt{decrypt}(\texttt{encrypt}(X, K), K); \texttt{decrypt}(\texttt{encrypt}(Y, K), K)\}$. Apply rewriting rules to this set yields that the abstract intruder knowledge contains both X and Y. The abstract intruder can then compute the concatenation $\texttt{pair}(X, Y)$.

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 [10] 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, Y): it can obtain X from A_1, then obtain Y from A_2, then compute (X, Y). For these reasons, we think that the approximation is fine enough.

6 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.

6.1 The Protocol Analyzer

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).

6.2 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. This reduction is similar to the *partial order reduction* techniques used in model-checking [12, chapter 10] to speed up analyzes of models of concurrent asynchronous systems.

6.3 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 [13]. 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 [29] and BANE [18], 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 [7].

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, leading to a semantics similar to Goubault's [25].

7 Experimental Results

We used the first two implementations cited in §6.3 on some examples, some of which academic samples, some of them real protocols from the standard papers on the topic.

7.1 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 [10];
- the "Test n" examples: n principals running each the program: $?r$ $\mathsf{decrypt}(r, K_n)$
  the initial knowledge of the intruder being $\mathsf{encrypt}(\cdots(\mathsf{encrypt}(X, K_1),$ $\ldots, a$ $lowbreak K_n)$; the unknown piece of data the intruder tries to recover being X.

Alas, while other protocols [10,22], 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 become too large.

7.2 An Interesting Point on the Otway-Rees Protocol

An early trial of our program on the Otway-Rees protocol (§2.9) yielded some unexpected results. This protocol features a principal A running:

```
! pair(A, pair(B, pair(M, encrypt(pair(pair(Na, M), pair(A, B)), Kas))));
? r;
r =~ pair(B, pair(A, encrypt(pair(Na, $kab), Kas)));
! encrypt(X, kab);
```

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 $\texttt{pair}(B, \texttt{pair}(A, \texttt{encrypt}(\texttt{pair}(N_a, \texttt{pair}(A, B)), K_{as})))$, built from pieces of the message output by A at step 1. A will then use $\texttt{pair}(A, B)$ as k_{ab}. On the other hand, reorganizing the output from step 1, replacing $\texttt{pair}(N_a, \texttt{pair}(M, \texttt{pair}(A, B)))$ by $\texttt{pair}(\texttt{pair}(N_a, M), \texttt{pair}(A, B))$, 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.

8 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.

The main drawback of our method is the high number of interleavings to consider, which limits the number of simultaneous sessions to be analyzed in practice. It is nevertheless possible to use further abstraction to analyze larger numbers of sessions provided we have a suitable *widening operator* [15]. Our idea of upper-approximate sets of terms was further refined by Genet and Klay [20] through the use of *stable sets* which enable the approximation of an unbounded number of sessions.

We hope that further progress on the algorithmics for abstract sets of trees [19,33] will make the abstract analysis of cryptographic protocols, and thus automated proofs, more tractable.

References

[1] Martín Abadi. Secrecy by typing in security protocols. In *14th Symposium on Theoretical Aspects of Computer Science (STACS'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[2] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[3] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73, Warsaw, Poland, July 1997. Springer-Verlag.

[4] Martín Abadi and Mark R. Tuttle. A semantics for a logic of authentication. In Luigi Logrippo, editor, *10th Annual ACM Symposium on Principles of*

Distributed Computing, pages 201–216, Montréal, Québec, Canada, August 1991. ACM Press.

[5] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures (FoSSaCS'01)*, volume 2030 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.

[6] 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.

[7] 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.

[8] Robert Amadio and Denis Lugiez. On the reachability problem in cryptographic protocols. Technical Report 3915, INRIA, 2000.

[9] M. Bellare and P. Rogaway. Provably secure session key distribution–the three party case. In *27th ACM Symposium on Theory of Computing (STOC)*, pages 57–66, 1995.

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

[11] Cervesato, Durgin, Lincoln, Mitchell, and Scedrov. A meta-notation for protocol analysis. In *CSFW 12: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[12] Edmund M. Clarke, Jr, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

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

[14] 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.

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

[16] Dan Craigen and Mark Saaltink. Using EVES to analyze authentication protocols. Technical Report TR-96-5508-05, ORA Canada, Ottawa, March 1996.

[17] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In ACM, editor, *Proceedings of the twenty third annual ACM Symposium on Theory of Computing, New Orleans, Louisiana, May 6–8, 1991*, pages 542–552. IEEE Computer Society Press, 1991. Full version available from authors.

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

[19] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Rewriting Techniques and Applications (RTA-98)*, volume 1379 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[20] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In D. McAllester, editor, *Automated Deduction (CADE-17)*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

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

[22] 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 Press.

[23] Li Gong and Paul Syverson. Fail-stop protocols: An approach to designing secure protocols. In *5 th International Working Conference on Dependable Computing for Critical Applications*, September 1995.

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

[25] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In Dominique Méry Beverly Sanders, editor, *Beverly Sanders , Dominique M Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA 2000)*, number 1800 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[26] Sinéad Hanley. DNS overview with a discussion of DNS spoofing. http://www.sans.org/infosecFAQ/DNS/DNS.htm.

[27] 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.

[28] D. Kindred and J. M. Wing. Fast, automatic checking of security protocols. In *Second USENIX Workshop on Electronic Commerce*, pages 41–52, Oakland, California, November 1996. USENIX.

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

[30] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.

[31] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *11th Computer Security Foundations Workshop*. IEEE, 1998.

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

[33] Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation.* PhD thesis, École Polytechnique, 2000.

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

[35] J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proceedings of the 1984 Symposium on Security and Privacy (SSP '84)*, pages 134–141, Los Angeles, Ca., USA, April 1990. IEEE Computer Society Press.

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

[37] David Monniaux. Abstracting cryptographic protocols with tree automata. In *Sixth International Static Analysis Symposium (SAS'99)*, number 1694 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[38] David Monniaux. Decision procedures for the analysis of cryptographic protocols by logics of belief. In *12th Computer Security Foundations Workshop.* IEEE, 1999.

[39] Bruce Schneier. *Applied Cryptography.* Wiley, second edition, 1996.

[40] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 87–100, Berlin, July 13–17 1997. Springer.

[41] Dawn Song. Athena: A new efficient automatic checker for security protocol analysis. In *12th Computer Security Foundations Workshop*. IEEE, 1999.

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

[43] 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.

[44] Paul Syverson. Towards a strand semantics for authentication logics. *Electronic Notes in Theoretical Computer Science*, 20, 1999.