A Method for Automatic Cryptographic Protocol Verification (Extended Abstract)

Jean Goubault-Larrecq

G.I.E. Dyade & Projet Coq, Inria, France (Jean.Goubault@dyade.fr)

Abstract. We present an automatic, terminating method for verifying confidentiality properties, and to a lesser extent freshness properties of cryptographic protocols. It is based on a safe abstract interpretation of cryptographic protocols using a specific extension of tree automata, $\vee$-parameterized tree automata, which mix automata-theoretic techniques with deductive features. Contrary to most model-checking approaches, this method offers actual security guarantees. It owes much to D. Bolignano's ways of modeling cryptographic protocols and to D. Monniaux' seminal idea of using tree automata to verify cryptographic protocols by abstract interpretation. It extends the latter by adding new deductive abilities, and by offering the possibility of analyzing protocols in the presence of parallel multi-session principals, following some ideas by M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi.

1 Introduction

It is now well-known that secure cryptographic algorithms (see e.g., [17]) do not suffice in providing system-wide security guarantees, and that one has to be careful in designing cryptographic *protocols*, namely sequences of exchanges of messages purporting to achieve the communication of some piece of data, keeping it confidential or ensuring some level of authentication, to name a few properties of interest [6].

Successful attacks against cryptographic protocols are usually silly, in the sense that they are purely *logical* and do not exploit any weakness in the underlying cryptographic algorithms (e.g., encryption); they are nonetheless difficult to spot. To avoid logical faults, several methods have been designed, based on modal logics of beliefs ([6] and successors), on complexity theory [3] (for specific protocols), on process-algebraic techniques [2], on type disciplines [1], on model-checking [12, 13], or on deductive techniques [14, 4, 16]. While model-checking techniques are fully automated and have been used to find attacks, they cannot directly give actual security guarantees—although reductions to finite-state cases manage to do so in well-behaved cases [18]. On the other hand, the deductive techniques have been designed to give security guarantees, but mechanization is in general partial, as fully automated proof search in general does not terminate. In any case, abstract interpretation (see [8]) can help prepare the grounds for each style of verification. In fact, abstract interpretation alone suffices to verify protocols, as D. Monniaux shows [15], using tree automata to model the set of messages that intruders may build. F. Klay and T. Genet [10] also propose to use tree automata, this time to model the whole protocol itself. Each of the latter two approaches has advantages and disadvantages, but they are automatic, terminate and aim indeed at giving security guarantees, contrarily to standard model-checking tools.

Our goal is to present yet another automated technique for guaranteeing the absence of logical faults in cryptographic protocols, which uses tree automata as well. Our contribution is twofold. First, instead of using standard tree automata, we use a refinement ($\vee$PTAs) allowing us to mix enumerative techniques (automata) with deductive techniques (BDDs [5]). The latter will notably help us in modeling freshness and initial states of intruder knowledge. Our $\vee$PTAs will also be much smaller than standard tree automata, improving the efficiency of verification markedly. Second, we extend the simulation of protocol runs to the case of *parallel multi-session principals*, e.g., key servers, an important case of unbounded parallelism, using ideas from [9].

For space reasons, this paper is only an overview. Moreover, we concentrate on secrecy because it is so fundamental; authentication can be dealt with by simple extensions of the framework presented here, following [10] for example. We describe $\vee$PTAs in Section 2, and use them to represent and compute states of knowledge in Section 3. We report on practical experience with these techniques in Section 4, showing its practical value, and shedding light on its strengths and weaknesses. We conclude in Section 5.

2 Terms, Formulae, $\vee$-Parameterized Tree Automata

Let $\mathcal{T}$ be a set of so-called *types* τ. Let $\mathcal{F}$ a set of so-called *function symbols*. A first-order *signature* Σ over $\mathcal{F}$ is a map from $\mathcal{F}$ to the set of expressions of the form $\tau_1 \times \ldots \times \tau_n \to \tau$, where $n \in \mathbb{N}$ and $\tau_1, \ldots, \tau_n, \tau$ are types.

Let $\mathcal{X}_\tau$, for each type τ, be pairwise disjoint non-empty sets, disjoint from $\mathcal{F}$, and $\mathcal{X}$ be $(\mathcal{X}_\tau)_{\tau \in \mathcal{T}}$. The set $T_\tau(\Sigma, \mathcal{X})$ of *terms of type* τ is the smallest set containing $\mathcal{X}_\tau$ and such that for each $f \in \mathcal{F} = \mathrm{dom}\,\Sigma$, if $\Sigma(f) = \tau_1 \times \ldots \times \tau_n \to \tau$ and $t_1 \in T_{\tau_1}(\Sigma, \mathcal{X})$, $\ldots$, $t_n \in T_{\tau_n}(\Sigma, \mathcal{X})$, then $f(t_1, \ldots, t_n)$ is in $T_\tau(\Sigma, \mathcal{X})$. We write f instead of $f()$.

We use propositional formulae, up to logical equivalence, to represent (some) sets of terms. Let $\mathcal{A}_\tau$, for each type τ, be a set of so-called *logical variables* of type τ. The intent is that each logical variable α of type τ denotes a set of terms of type τ. *Propositional formulae* F of type τ are defined by the grammar :

$$F ::= A \mid F \wedge F \mid F \vee F \mid \neg F \mid \mathbf{0} \mid \mathbf{1}$$

where A ranges over $\mathcal{A}_\tau$. Formulae F are interpreted as sets $[\![F]\!]\rho_\tau$ in *environments* ρ, where ρ is any family $(\rho_\tau)_{\tau \in \mathcal{T}}$ of maps ρ_τ from $\mathcal{A}_\tau$ to $T_\tau(\Sigma, \mathcal{X})$, by interpreting $\mathbf{0}$ as $\emptyset$, $\mathbf{1}$ as $T_\tau(\Sigma, \mathcal{X})$, $\wedge$ as intersection, $\vee$ as union, $\neg$ as complement.

To deal with term structure, we define the following variant of tree automata. Compared to ordinary tree automata [7], ours integrate propositional formulae at states, and the states are typed (the latter helps in practice limit the size of automata, and does not restrict the generality of the approach). To simplify the following definition, extend Σ to $\mathcal{F} \cup \bigcup_{\tau \in \mathcal{T}} \mathcal{X}_\tau$ by letting $\Sigma(x) \hat{=} \to \tau$ for every $x \in \mathcal{X}_\tau$.

Let $\mathcal{Q}$ be a set of so-called *states* q. We assume that each state q has a *type* τ_q, and that $\mathcal{Q}$ contains infinitely many states of each type.

An *$\vee$-parameterized tree automaton, or $\vee$PTA, of type* τ_0, $\mathfrak{A}$, is a 4-tuple $(\mathfrak{Q}, \mathfrak{F}, \mathfrak{R}, \mathfrak{B})$, where $\mathfrak{Q}$ is a finite subset of $\mathcal{Q}$, whose elements are the *states of* $\mathfrak{A}$, $\mathfrak{F} \subseteq \mathfrak{Q}$ is the set of *final states*, $\mathfrak{B}$ maps each state $q \in \mathfrak{Q}$ to a formula of type τ_q, and $\mathfrak{R}$ is a set of rewrites rules $f(q_1, \ldots, q_n) \to q$, the *transitions*, where $f \in \mathcal{F} \cup \bigcup_{\tau \in \mathcal{T}} \mathcal{X}_\tau$

is such that $\Sigma(f) \hat{=} \tau_{q_1} \times \ldots \times \tau_{q_n} \to \tau_q$ ("transitions respect types")—in case f is a variable of type τ, this means $n = 0$, $\Sigma(x) = \to \tau$.

Ordinary tree automata are just $\vee$PTAs without the $\mathfrak{B}$ component (or equivalently, where $\mathfrak{B}$ maps each state to the class of $\mathbf{0}$.) The semantics of $\vee$PTAs is given by defining when a $\vee$PTA $\mathfrak{A} \hat{=} (\mathfrak{Q}, \mathfrak{F}, \mathfrak{R}, \mathfrak{B})$ *recognizes a term t in an environment ρ at a state q*; this is so if and only if $t \in [\![\mathfrak{B}(q)]\!]\rho_{\tau_q}$, or t is of the form $f(t_1, \ldots, t_n)$, and there is a transition $f(q_1, \ldots, q_n) \to q$ in $\mathfrak{R}$ such that t_j is recognized by $\mathfrak{A}$ in ρ at q_j for each j, $1 \leq j \leq n$. A term t is *recognized by $\mathfrak{A}$ in ρ* if it so at some final state of $\mathfrak{A}$.

We can compute unions of $\vee$PTAs exactly, and give upper approximants of their intersections by a standard automaton product construction. (This construction gives an exact result in the case of *normal* $\vee$PTAs to be described later.) We can always test whether an $\vee$PTA is *definitely empty*, i.e. whether it cannot recognize any term under any environment ρ: create a Boolean variable ne_q for each state of the $\vee$PTA, produce the clause ne_q if $\mathfrak{B}(q)$ is not equivalent to $\mathbf{0}$ (for each q), the clause $ne_{q_1} \wedge \ldots \wedge ne_{q_n} \Rightarrow ne_q$ for each transition $f(q_1, \ldots, q_n) \to q$ with $\mathfrak{B}(q)$ equivalent to $\mathbf{0}$, and $\neg ne_q$ for each final state q; if the resulting set of clauses is satisfiable, then the given $\vee$PTA is definitely empty; to check it, we use BDDs [5] to represent sets $\mathfrak{B}(q)$ and unit resolution to solve the resulting set of Horn clauses.

We define *assumptions* to be maps $\mathfrak{H}$ from types τ to formulae of type τ. The environment $\rho \hat{=} (\rho_\tau)_{\tau \in \mathcal{T}}$ *satisfies* $\mathfrak{H}$, written $\rho \models \mathfrak{H}$, if and only if $[\![\mathfrak{H}(\tau)]\!]\rho_\tau$ is the set of all terms of type τ, for every type τ. For any two formulae F and G of type τ, we write $F \cap G = \emptyset$ the assumption mapping τ to $\neg(F \wedge G)$ and every other type to $\mathbf{1}$. Given a finite family of assumptions $\mathfrak{H}_i$, $i \in I$, their *conjunction* maps every type τ to $\bigwedge_{i \in I} \mathfrak{H}_i(\tau)$. We reason on $\vee$PTAs $\mathfrak{A}$ modulo assumptions $\mathfrak{H}$ by *reducing* $\mathfrak{A}$, replacing $\mathfrak{B}(q)$ by $\mathfrak{B}(q) \wedge \mathfrak{H}(\tau_q)$ for each state q to get a new $\vee$PTA $\mathfrak{A}_{|\mathfrak{H}}$: under any environment ρ satisfying $\mathfrak{H}$, $\mathfrak{A}$ and $\mathfrak{A}_{|\mathfrak{H}}$ recognize the same terms, and if $\mathfrak{A}_{|\mathfrak{H}}$ is definitely empty, then for no environment ρ satisfying $\mathfrak{H}$, $\mathfrak{A}$ recognizes any term.

3 Messages, What Intruders Know, and Simulating Protocol Runs

To be more specific, our set $\mathcal{T}$ of types contains the type `msg` of *messages*; the type `msglist` of tuples of messages, which we shall use to build argument lists to the `t` tupling operator below; the type `K` of *raw keys*, e.g. integers of some fixed sizes used to build actual keys, of type `key`, which we assume to be in $\mathcal{T}$ as well; the type `D` of *raw data*, e.g. integers, reals, strings, etc. $\mathcal{T}$ may contain other types, which we do not care about. The *basic signature* Σ_0 is:

`symk, asymk1, asymk2` : `K` $\to$ `key`	`k` : `key` $\to$ `msg`
`d` : `D` $\to$ `msg`	`c` : `msg` $\times$ `key` $\to$ `msg`
`sk` : `msg` $\times$ `msg` $\to$ `key`	`t` : `msglist` $\to$ `msg`
`pubk, privk` : `msg` $\to$ `key`	`nil` : $\to$ `msglist`
$* : \to$ `key`	`cons` : `msg` $\times$ `msglist` $\to$ `msglist`

The `symk` constructor builds symmetric keys from raw keys, `asymk1` and `asymk2` build the two parts of asymmetric keys; `sk` returns a long-term session key shared between the two principals in argument, `pubk` and `privk` return their argument's public and

private keys respectively. Any actual key is a message, as represented by the explicit conversion symbol `k`. Similarly, we use `d` to convert raw data to messages. The symbol `c` is used to build *ciphertexts*: $\mathtt{c}(M, K)$ is the result of encrypting the *plaintext* M with key K. The special key $*$ is used to model the hash code of M as $\mathtt{c}(M, *)$. Finally, any list of messages can be made into a message, using the tupling constructor `t` that takes a list of messages, of type `msglist`, in argument: the latter are built using the standard Lisp constructors `nil` and `cons`. For legibility we shall abbreviate $\mathtt{cons}(M_1, \ldots, \mathtt{cons}(M_n, \mathtt{nil}) \ldots)$ as $[M_1, \ldots, M_n]$.

We consider as our actual signature Σ any one of the form $\Sigma_0 \uplus \Sigma_1$, where Σ_1 is an unspecified collection of function symbols of signatures $\tau_1 \times \ldots \times \tau_n \to \tau$ where $\tau \notin \{\mathtt{key}, \mathtt{msg}, \mathtt{msglist}\}$. Leaving $\mathcal{T}$ and Σ partly unspecified allows us to deal with extensible types for raw keys and raw data.

We say that, for any keys K and K' (of type `key`), K' is an *inverse* of K if and only if $K = \mathtt{symk}(k)$ or $K = \mathtt{sk}(M_1, M_2)$ and $K' = K$; or $K = \mathtt{asymk1}(k)$ and $K' = \mathtt{asymk2}(k)$; or $K = \mathtt{asymk2}(k)$ and $K' = \mathtt{asymk1}(k)$; or $K = \mathtt{pubk}(M)$ and $K' = \mathtt{privk}(M)$; or $K = \mathtt{privk}(M)$ and $K' = \mathtt{pubk}(M)$. Note that $*$ has no inverse.

Intruders can read on any communication line, and collect what they read. Let E be a set of messages that the intruders have collected (this set might be infinite). These intruders can then forge new messages from E and send them to other principals. Following [4], we model intruders as a deductive system. Write $E \vdash\!\!\!\mapsto M$ the predicate "from the set E of messages, the intruders may deduce the message M", defined as follows (E, M denotes the union of E with $\{M\}$):

$$\frac{}{E, M \vdash\!\!\!\mapsto M}\,(Ax)$$

$$\frac{E \vdash\!\!\!\mapsto M \quad E \vdash\!\!\!\mapsto \mathtt{k}(K)}{E \vdash\!\!\!\mapsto \mathtt{c}(M, K)}\,(CryptI) \qquad \frac{E \vdash\!\!\!\mapsto \mathtt{c}(M, K) \quad E \vdash\!\!\!\mapsto \mathtt{k}(K') \quad (K' \text{ inverse of } K)}{E \vdash\!\!\!\mapsto M}\,(CryptE)$$

$$\frac{E \vdash\!\!\!\mapsto M_1 \quad \ldots \quad E \vdash\!\!\!\mapsto M_n}{E \vdash\!\!\!\mapsto \mathtt{t}([M_1, \ldots, M_n])}\,(TupleI) \qquad \frac{E \vdash\!\!\!\mapsto \mathtt{t}([M_1, \ldots, M_n])}{E \vdash\!\!\!\mapsto M_i}\,(TupleE_i), 1 \leq i \leq n$$

So intruders may replay messages (Ax), *construct* messages by encryption and tupling ($(CryptI)$, $(TupleI)$), and *extract* messages by decryption and field selection ($(CryptE)$, $(TupleE_i)$)—but they cannot crack ciphertexts. Then we may always assume without loss of generality that intruders do all extractions before any construction [4]. That is, let $Ded(E)$ be the set of messages *deducible* from E, i.e., those such that $E \vdash\!\!\!\mapsto M$ is derivable; let $Con(E)$ be the constructible ones (derivable using only (Ax), $(CryptI)$, $(TupleI)$), and $Ext(E)$ the extractible ones (derivable using only (Ax), $(CryptE)$, $(TupleE_i)$). Then $Ded(E) = Con(Ext(E))$.

We represent sets of messages E by $\vee$PTAs, more precisely by *normal* $\vee$*PTAs*, whose states q of type `msg`, `msglist` or `key` are such that $\mathfrak{B}(q)$ is equivalent to $\mathbf{0}$, and whose transitions $f(q_1, \ldots, q_n) \to q$ are such that f is in the basic signature Σ_0. In particular, computing intersections can be done exactly on normal $\vee$PTAs.

A central result is that for every normal $\vee$PTA $\mathfrak{A}$ of type `msg`, there is a normal $\vee$PTA that we call $Ded(\mathfrak{A})$ such that, if E is the set of terms recognized by $\mathfrak{A}$ in ρ,

then $Ded(\mathfrak{A})$ recognizes at least the terms of $Ded(E)$ in ρ. The idea is by constructing $Ded(\mathfrak{A})$ as $Con(Ext(\mathfrak{A}))$, where the semantics of Con and Ext are as expected.

Building $Ext(\mathfrak{A})$ works by saturating the set $\mathfrak{F}$ of final states by the following two rules: for every transition $\mathtt{t}(ql) \to q$ where q is in $\mathfrak{F}$, add to $\mathfrak{F}$ all states q' of type `msg` reachable from ql by following `cons`-transitions backwards (rule $(TupleE_i)$); for every transition $\mathtt{c}(q', qk) \to q$ with $q \in \mathfrak{F}$, add q' to $\mathfrak{F}$ if for some transition $\mathtt{k}(qk') \to qf$ with $qf \in \mathfrak{F}$, qk' contains possible inverses of qk (rule $(CryptE)$): qk' *contains possible inverses* of qk when there are transitions $f_1(q_{11}, \ldots, q_{1n}) \to qk$ and $f_2(q_{21}, \ldots, q_{2n}) \to qk'$ such that q_{1j} and q_{2j} intersect possibly for every $1 \leq j \leq n$, where $f_1 = \mathtt{symk}$ and $f_2 = \mathtt{symk}$, or $f_1 = \mathtt{asymk1}$ and $f_2 = \mathtt{asymk2}$, etc. (see definition of inverse keys); two states q_1 and q_2 of the same type *intersect possibly* if and only if the intersection of $(\mathfrak{Q}, \{q_1\}, \mathfrak{R}, \mathfrak{B})$ and $(\mathfrak{Q}, \{q_2\}, \mathfrak{R}, \mathfrak{B})$ is not definitely empty.

To build $Con(\mathfrak{A})$, add two fresh states qm of type `msg` and ql of type `msglist` to $\mathfrak{A}$, mapped to $\mathbf{0}$ by $\mathfrak{B}$. Then for each transition $f(q_1, \ldots, q_n) \to q'$, where q' is in the set $\mathfrak{F}$ of final states of $\mathfrak{A}$, add a transition $f(q_1, \ldots, q_n) \to qm$, and add transitions $\mathtt{nil}() \to ql$, $\mathtt{cons}(qm, ql) \to ql$, and $\mathtt{t}(ql) \to qm$ (rule $(TupleI)$), and transitions $\mathtt{c}(qm, q) \to qm$ for every transition $\mathtt{k}(q) \to q'$ with q' final in $\mathfrak{A}$ (rule $(CryptI)$).

We simulate protocol runs by describing each principal as a small program. Programs are sequences of instructions, which may either create raw keys, create raw data (nonces), write expressions onto output channels, or read expressions from input channels while pattern-matching them (à la ML). We verify protocols by simulating all possible interleavings (modulo some partial order reductions). The Ded operator handles writes: writing a message M adds M to the set E of messages, and is abstracted by the computation of $Ded(\mathfrak{A})$, where $\mathfrak{A}$ is the normal $\vee$PTA abstracting E. Reads returns any message M such that $E \mid\!\!\!\rightarrow M$ is derivable: we abstract this by having the read instruction return the $\vee$PTA $\mathfrak{A}$ abstracting E itself as abstract value. Note that abstract values associated with each program variable denote sets of concrete messages, and are represented as normal $\vee$PTAs again. Pattern-matching is done in the abstract semantics just as in the concrete semantics, replacing equality tests between concrete messages M_1 and M_2 by tests that the $\vee$PTAs that abstract M_1 and M_2 have an intersection that is not definitely empty after reduction by the current set of assumptions $\mathfrak{H}$. Creating fresh raw data is done as follows. With each instruction creating raw data we associate a *freshness variable* $X \in \mathcal{A}_{\mathtt{D}}$; then we insist that $\mathfrak{H}$ be the conjunction of all assumptions $X \cap Y = \emptyset$ for every two distinct freshness variables X and Y, and possibly of other assumptions. ($\mathfrak{H}$ is fixed at the beginning of the simulation and never changes.) Then the abstract value of the variable containing the newly created data is the automaton $(\{q\}, \{q\}, \emptyset, \{q \mapsto X\})$ recognizing exactly those data in (the semantics of) X. Creating fresh keys is done similarly. Note that propositional variables are really needed here to deal with freshness of nonces and keys.

Before we start the simulation, we need to describe the initial set of messages that the intruders know. So let K_0 and D_0 be propositional variables denoting the sets of raw keys that exist (i.e., have been created already), respectively raw data that exist at the start of the run. Let SSK_0, $SAK1_0$, $SAK2_0$ be variables denoting the sets of raw keys k such that $\mathtt{symk}(k)$, resp. $\mathtt{asymk1}(k)$, resp. $\mathtt{asymk2}(k)$ are initially unknown to the intruders. Let SD_0 be a variable denoting the set of raw data d such that $\mathtt{d}(d)$ is initially

unknown to intruders. Assuming for simplicity that every key $\mathtt{sk}(\dots)$ or $\mathtt{privk}(\dots)$ is initially unknown to intruders, and that all keys $\mathtt{pubk}(\dots)$ and $*$ are known, we build an $\vee$PTA $\mathfrak{A}_0$ recognizing the greatest set of terms M known to the intruders validating the secrecy assumptions above. Informally, this is done as follows. Create a state qd of all raw data assumed to exist and initially known; a state qk of all keys assumed to exist and initially known; a state qk^{-1} of all keys assumed to exist but that have no initially known inverse. Then the set E of terms M we look after is given by: M is either $\mathtt{d}(d)$ with d recognized at qd, or $\mathtt{k}(k)$ with k recognized at qk, or a tuple $\mathtt{t}([M_1, \dots, M_n])$ where each M_i is in E, or $\mathtt{c}(M, K)$, where either M is in E and K is any existing key, or M is any existing message and K is recognized at qk^{-1}. This description can be turned easily into an actual $\vee$PTA $\mathfrak{A}_0$.

We also extend the simulation to handle an unbounded number of copies of any given group of principals. This handles the case of so-called *parallel multi-session* principals S, such as key servers, which actually spawn a new thread after each connection request. (They behave as processes $!S$ in the π-calculus, i.e. they run an unbounded number of copies of S in parallel.) To deal with this case, we use an idea from [9]: such principals S are viewed as accomplices to intruders, and we model them by extending the $Ded(\mathfrak{A})$ automaton by new states and transitions to account for the added computing power that all the copies of S contribute to intruders. This is technical, but let us give a rough idea. First, we assume that each creation (of raw data, of raw keys) done by each copy of S actually returns some unspecified data in the denotation of the freshness variable associated with the creation instruction; so we confuse every copy of S, as far as freshness is concerned. Then, we assume that each instruction of any copy of S executes in any order. Next, we assume that each read succeeds, and pattern-matching is approximated in a crude way: for example, in a $\mathtt{read}\,\mathtt{t}([\mathtt{c}(x, K), y])$ which attempts to read a pair, put the second component in y, decrypt the first component with K and put the resulting plaintext in x, we simply estimate that the value of y will be anything known to intruders, and the resulting value of x will be anything that exists (possibly not known to intruders, because of the enclosing $\mathtt{c}$). We model this by enriching the automaton $Ded(\mathfrak{A})$ with two states, qkn recognizing all known messages, and qx recognizing all existing messages. Writes are then coded by merging these states with other states; e.g., writing $\mathtt{t}([x, \mathtt{c}(y, K)])$ with the same x and y as above implies that $\mathtt{t}([x, \mathtt{c}(y, K)])$ must be recognized at qkn, so that x and $\mathtt{c}(y, K)$ are recognized at qkn, because of $(TupleE_i)$. As far as x is concerned, this means losing any information on existing but unknown messages (merge the qx and qkn states). For $\mathtt{c}(y, K)$, everything depends on whether we assume K to have a known inverse or not: in the first case, then y must exist, otherwise it must become known to the intruders; in any case, since y was already assumed to be known, we do nothing here. In general, the problem of knowing whether K has a known inverse or not matters, and is solved by a fixpoint iteration, which converges because we only deal with finitely many key expressions.

4 Experimental Results

We have implemented these techniques using a bytecode compiler for HimML, a variant of Standard ML incorporating facilities for handling finite sets and maps elegantly and

efficiently [11]. We have then tested this implementation on standard cryptographic protocols [6], on a 166MHz Intel Pentium machine running Linux 2.0.30. Each of these protocols are three-party protocols, involving two principals A and B that wish to get a secret key K_{ab} by interacting with a key server S. All of these protocols were tested under an empty assumption $\mathfrak{H}$. Results and running times are as follows:

	S in mono-session			S in parallel multi-session		
Protocol	Result	Time (s.)	#Branches	Result	Time (s.)	#Branches
Needham-Schroeder shared key	p.f.	1.94	4	p.f.	1.56	3
Otway-Rees	OK	1.56	3	OK	1.56	3
Wide-Mouthed Frog	p.f.	0.34	2			
Yahalom	p.f.	1.17	4	p.f.	1.2	3
SimplerYahalom	OK	1.16	3	OK	1.52	3
Otway-Rees2	OK	3.54	4	OK	14.57	15

In the result column, "OK" means the protocol passed, "p.f." means that it contains a possible flaw. The "#Branches" column indicates how much non-determinism is involved in checking all relevant interleavings of the protocol. Times are in seconds, and total the whole exploration of all relevant interleavings; in other words, our tool does not just stop after the first possible flaw.

Note that the Needham-Schroeder protocol was found to be flawed, and indeed our tool finds the standard attack where the intruder plays the second part of the session alone against B, without A or S participating at all. The Yahalom protocol was found to be flawed, too: whether or not our tool has found an attack remains to be examined; indeed, reading attacks off $\vee$PTAs is not an easy task! But, as noticed in [6], the Yahalom protocol is a very subtle one, and requires strong assumptions. (By the way, our tool only detects flaws in B's behaviour, so we are guaranteed that A at least cannot be fooled.) On the other hand, the SimplerYahalom protocol (an improved version of the Yahalom protocol given in [6]) is found to be correct by our tool, confirming the opinion of op.cit. that this second version is easier to show correct than the original one.

The last line of the table shows a simulation of two sessions of the Otway-Rees protocol in sequence: OtwayRees2 simulates a principal A_2 playing the role of A twice in a row (with A's identity, and trying to communicate with the same B twice), a principal B_2 that plays the role of B twice in a row (with B's identity, but without checking that its peer is the same A in both sessions), and a server S. The time taken by our tool is still very reasonable, although there should be many more interleavings than for OtwayRees. We are saved by the fact that several interleavings are impossible: our tool discovers that some `reads` must block (abstract pattern-matching fails).

The worst-case complexity of our algorithms is daunting: abstract pattern-matching in particular takes exponential time and produces $\vee$PTAs of exponential size. Nonetheless, the nice news is that verification of actual protocols is quite fast on average, while still maintaining a high level of accuracy.

5 Conclusion

We hope to have convinced the reader that automatic verification of cryptographic protocols was now possible, including some limited form of deduction, and allowing us to prove properties like "M is definitely secret at program point p, whatever the initial

messages known to the intruder, provided that assumption $\mathfrak{H}$ is verified". Our technique is natural, provides actual secrecy guarantees—and to a lesser extent freshness guarantees—, and works fast in practice.

Acknowledgments

Many thanks to Dominique Bolignano, David Monniaux, and Mourad Debbabi.

References

1. M. Abadi. Secrecy by typing in cryptographic protocols. *Journal of the Association for Computing Machinery*, 1998. Submitted.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.
3. M. Bellare and P. Rogaway. Provably secure session key distribution–the three party case. In *27th ACM Symposium on Theory of Computing (STOC'95)*, pages 57–66, 1995.
4. D. Bolignano. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communication Security*, 1996.
5. R. E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, C35(8):677–692, 1986.
6. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, 426(1871):233–271, 1989.
7. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on `http://www.grappa.univ-lille3.fr/tata/`, 1997.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. Correct version at `http://www.dmi.ens.fr/~cousot/COUSOTpapers/JLP92.shtml`.
9. M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. Formal automatic verification of authentication cryptographic protocols. In *1st IEEE International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE, 1997.
10. T. Genet and F. Klay. Rewriting for cryptographic protocol verification (extended version). Technical report, CNET-France Telecom, 1999. Available at `http://www.loria.fr/~genet/Publications/GenetKlay-RR99.ps`.
11. J. Goubault. HimML: Standard ML with fast sets and maps. In *5th ACM SIGPLAN Workshop on ML and its Applications*, 1994.
12. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, pages 147–166. Springer Verlag LNCS 1055, 1996.
13. W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-SCS-97-139, Carnegie Mellon University, 1997.
14. C. A. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 1995.
15. D. Monniaux. Abstracting cryptographic protocols with tree automata. In *6th International Static Analysis Symposium (SAS'99)*. Springer-Verlag LNCS 1694, 1999.
16. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
17. B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.
18. S. D. Stoller. A bound on attacks on authentication protocols. Technical Report 526, Indiana University, 1999. Available from `http://www.cs.indiana.edu/hyplan/stoller.html`.