Handling exp, × (and Timestamps) in Protocol Analysis*

Roberto Zunino and Pierpaolo Degano

Dipartimento di Informatica, Università di Pisa, Italy
{zunino, degano}@di.unipi.it

Abstract. We present a static analysis technique for the verification of cryptographic protocols, specified in a process calculus. Rather than assuming a specific, fixed set of cryptographic primitives, we only require them to be specified through a term rewriting system, with no restrictions. Examples are provided to support our analysis. First, we tackle forward secrecy for a Diffie-Hellman-based protocol involving exponentiation, multiplication and inversion. Then, a simplified version of Kerberos is analyzed, showing that its use of timestamps succeeds in preventing replay attacks.

1 Introduction

Process calculi [16] have been extensively used for cryptographic protocol specification and verification, exploiting formal methods. Several of these calculi (e.g. Spi [2]), however, use a specific set of cryptographic primitives, which is often entwined with the definition of the process syntax and semantics, e.g. by introducing pattern matching on encrypted messages. On the one hand, this simplifies the presentation of the calculus; also the verification tools only need to consider a given set of primitives. On the other hand, protocols using different primitives cannot be specified in the calculus as it is: one has to suitably extend it and to adapt existing tools to cope with the extensions. Of course, the new tools also needs new, adapted soundness proofs.

The applied pi calculus [1] instead does not fix the set of primitives. Its processes can exchange arbitrary terms, that are considered up to some equivalence relation. This relation can be defined by the user through an equational theory. In this scenario, adding primitives is done by adding the relevant equations to the theory, without changing the syntax of the processes. In other words, the applied pi effectively separates the semantics of the processes, which is fixed, from the semantics of the terms, which is user-defined.

In this paper, we present a technique for the static analysis of protocols specified in a (slight) variant of the applied pi. In our calculus, term equivalence is instead specified through an *arbitrary* rewriting system $\mathcal{R}$. Indeed, we do not put any restrictions on $\mathcal{R}$: it needs neither to be confluent nor terminating.

* Partly supported by the EU within the FETPI Global Computing, project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 413–427, 2006.

Our technique borrows from the control flow analysis (CFA) approach [19, 5] and from the algorithms for non-deterministic finite tree automata (NFTA) [10, 21]. As in the CFA, we extract a number of constraints from a protocol specification, expressed as a process. Then, we solve the constraints using the *completion* algorithm in [23], which turns out to be very similar to the one in [10, 21]. The result is a NFTA $\mathcal{F}$ describing a language which is an over-approximation of the set of terms exchanged by the protocol, in *all* their possible equivalent forms according to $\mathcal{R}$. Finally, the automaton $\mathcal{F}$ can be inspected to check a number of security properties of the protocol. Essentially, ours is a reachability analysis.

Exploiting this technique, we analyzed protocols using both standard cryptographic primitives, such as encryptions and signatures, as well as more "problematic" primitives such as exponentials and XOR. Exponentials are hard to deal with because their equational theory has many equations, and therefore equivalent terms may assume very different shapes. As a consequence, it is hard to find an accurate over-approximation for them. The literature often reports on studies carried out assuming only a few equations. For instance, from the web page of the AVISPA project [3] one sees examples with three equations for exp and inversion $(\bullet)^{-1}$ over finite-fields, analyzed through the tool in [6]. In the example of Sect. 4.1, we consider exp, $\times$, 1, and $(\bullet)^{-1}$, axiomatizing their interactions with twelve equations. Yet, our implementation of the presented analysis was able to prove the *forward secrecy* property for a protocol based on the Diffie-Hellman key exchange [8].

Our technique also offers a limited treatment of time. Here we report on the success of our tool for the verification of a simplified version of the Kerberos protocol [20, 18], involving timestamps. In our specification, we allow the disclosure of an old session key, mimicking a secret leak. The tool was able to prove the secrecy of messages exchanged in newer sessions, confirming the protocol is resilient of replay attacks with the compromised old key.

Finally, our technique allows for some composition of results. Albeit with some limitations, it is possible to analyze the components of a system independently, and then merge the results later to derive a sound analysis for the whole system.

A related approach to ours is in [4]. However, they only consider certain equational theories, e.g. without associativity, and define a semi-algorithm to obtain rewriting rules with "partial normal forms." They then use ProVerif to check processes equivalent, thus establishing security properties. Also, some decidability results for (a significant fragment of) the exponential theory are in [15]. Other applications of NFTA to security can be found in [12, 11]. There, protocols are specified through rewriting, rather than process calculi. Another interesting work is by Goubault [13], dealing with exponentials through rewriting. There, however, only exponentials with a fixed base are considered. Monniaux in [17] also uses NTFA for verifying protocols, when crypto primitives can be expressed through left-linear rewritings. Finally, there is an earlier analysis for the applied pi calculus in [22]. However, it only applies to free terms, subject to no rewriting.

Summary. In Sect. 2 we introduce background and notation. We present our calculus in Sect. 3, defining its dynamic semantics in Sect. 4. The same section

has the Diffie-Hellman example. Sect. 5 describes the static analysis and its application to Diffie-Hellman and Kerberos. In Sect. 6 we discuss compositionality.

2 Background and Notation

A non-deterministic finite tree automaton (NFTA) $\mathcal{A}$ is determined by its finite set of states $\mathcal{Q} = \{\mathtt{@a}, \mathtt{@b}, \ldots\}$ and its set of transitions. Transitions have the form $\mathtt{@q} \to T$, where T is a generic term built using function symbols and states in $\mathcal{Q}$. For example, we consider the following $\mathcal{A}$:

$$\begin{array}{lll} \mathtt{@a} \to 0 & \mathtt{@a} \to 1 & \mathtt{@a} \to 2 \\ \mathtt{@b} \to \mathsf{nil} & \mathtt{@b} \to \mathsf{cons}(\mathtt{@a}, \mathtt{@b}) & \mathtt{@c} \to \mathsf{fst}(\mathtt{@b}) \end{array}$$

In the above the function symbols are $0, 1, 2, \mathsf{nil}$ (nullary), fst (unary) and cons (binary). States $\mathcal{Q}$ are $\{\mathtt{@a}, \mathtt{@b}, \mathtt{@c}\}$. Each state $\mathtt{@q}$ has an associated language $[\mathtt{@q}]_{\mathcal{A}}$, given by the set of the state-free terms reachable through transitions. For example, we have $\mathtt{@b} \to \mathsf{cons}(\mathtt{@a}, \mathtt{@b}) \to \mathsf{cons}(\mathtt{@a}, \mathsf{cons}(\mathtt{@a}, \mathtt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(\mathtt{@a}, \mathtt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(1, \mathtt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(1, \mathsf{nil})) = T$, and therefore $T \in [\mathtt{@b}]_{\mathcal{A}}$.

A term rewriting system $\mathcal{R}$ is a set of rewriting rules, having the form $L \Rightarrow R$, where L, R are terms built using function symbols and variables. For example, the usual rewriting rules for pairs are:

$$\mathsf{fst}(\mathsf{cons}(X, Y)) \Rightarrow X \qquad \mathsf{snd}(\mathsf{cons}(X, Y)) \Rightarrow Y$$

In [23] an algorithm is described for computing the $\mathcal{R}$-*completion* of an automaton $\mathcal{A}$. The result is another automaton $\mathcal{F}$ such that its languages 1) include those of $\mathcal{A}$, and 2) are closed under rewriting. Formally, $\mathcal{F}$ is such that whenever $\mathtt{@q} \to^*_{\mathcal{A}} \to^*_{\mathcal{R}} T$ also $\mathtt{@q} \to^*_{\mathcal{F}} T$ for any $\mathtt{@q}, T$. For instance, completing the $\mathcal{A}$ above, we obtain an $\mathcal{F}$ such that $\mathtt{@c} \to^*_{\mathcal{F}} 1$. A very similar algorithm was presented in [10]. Once such an $\mathcal{F}$ is computed, it is possible to verify properties about the languages of $\mathcal{A}$ up-to rewriting by inspecting their over-approximations in $\mathcal{F}$.

For our purposes, we also want $\mathcal{F}$ to satisfy a set of *intersection constraints* $\mathcal{I}$, provided as an input to the algorithm. These constraints have the form $\mathtt{@a} \cap \mathtt{@b} \subseteq \mathtt{@c}$, meaning that the intersection of the languages $[\mathtt{@a}]_{\mathcal{F}}$ and $[\mathtt{@b}]_{\mathcal{F}}$ must be included in $[\mathtt{@c}]_{\mathcal{F}}$. The algorithm in [23] was adapted to handle $\mathcal{I}$ and is the basis for our analysis tool.

The time complexity of the completion algorithm is polynomial (assuming that the depth of each left hand side in any rewriting rule is constant).

3 Syntax

Our process calculus is a simplified version of the applied pi calculus [1], in that processes exchange values using a global public network channel. Values are simply represented as terms, up to the equivalence specified by a rewriting system $\mathcal{R}$. We write $\mathcal{T}$ for the set of terms. We also use $\mathcal{X}$ as a set of variables. The syntax of our calculus is rather standard.

$$\pi ::= \mathsf{in}\ x \mid \mathsf{out}\ M \mid [x = y] \mid \mathsf{let}\ x = M \mid \mathsf{new}\ x \mid \mathsf{repl} \mid \mathsf{chk}$$
$$P ::= \mathsf{nil} \mid \pi\ .\ P \mid (P|P)$$

We now briefly describe our calculus: its semantics will be given in Sect. 4. Intuitively, nil is a process that performs no actions; $\pi.P$ executes the prefix π and then behaves as P; $P_1|P_2$ runs concurrently the processes P_1 and P_2. Prefixes perform the following actions: $\mathsf{in}\ x$ reads a term from the network and binds x to it; $\mathsf{out}\ M$ sends a term to the network; $[x = y]$ compares the term bound to x and y and stops the process if they differ; $\mathsf{let}\ x = M$ simply locally binds x to the value of M; $\mathsf{new}\ x$ generates a fresh value and binds x to it; repl spawns an unlimited number of copies of the running process, which will run independently; chk is a special action that we use to model certain kinds of attacks, which we will address in Sect. 4.

Note that match $[x = y]$ is only allowed between variables. This is actually not a restriction, since matching between arbitrary terms, e.g. $[M = N].P$ can be expressed by $\mathsf{let}\ x = M\ .\mathsf{let}\ y = N\ .[x = y].P$.

As usual, the *bound* variables in a process are those under a let,new, or in prefix; the others are *free*. A process with no free variables is *closed*.

Given a process, we use addresses $\theta \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}\}^*$ to point to its subprocesses. Intuitively, n chooses the continuation P for a process $\pi.P$, while l and r choose the left and right branch of a parallel $P_1|P_2$, respectively. An address θ is a concatenation of these selectors, singling out the subprocess $P@\theta$ as defined below. We write ϵ for the empty string.

$$P@\epsilon = P \qquad (P_1|P_2)@\theta\mathsf{l} = P_1@\theta$$
$$\pi.P@\theta\mathsf{n} = P@\theta \qquad (P_1|P_2)@\theta\mathsf{r} = P_2@\theta$$

4 Dynamic Semantics

Given a closed process P, we define its semantics through a multiset rewriting system [14, 7]. A state is a multiset σ of parallel threads. Each thread is formed by an environment $\rho \in \mathcal{X} \to \mathcal{T}$ and a continuation address θ singling out a subprocess of P. We write such a thread as $\langle \rho, \theta \rangle$. Intuitively, $\langle \rho, \theta \rangle$ runs the process $P@\theta$ under the bindings in ρ. The initial state is $\langle \emptyset, \epsilon \rangle$.

We extend ρ homomorphically to terms: $\rho(M)$ replaces variables in M with the value they are bound to in ρ. Also, as a handy convention, if $P@\theta = P_1|P_2$, we write $\langle \rho, \theta \rangle$ for the multiset $\{\langle \rho, \mathsf{l}\theta \rangle, \langle \rho, \mathsf{r}\theta \rangle\}$, or its further expansion, so that threads in the state never have continuation addresses θ' such that $P@\theta'$ has the form $P_1|P_2$.

Our semantics is given by the rules in Fig. 1. Local rules only care about one or two elements of the current state: these elements are rewritten independently of the rest of the state, which does not change. All the rules fire a prefix, advancing the current continuation address θ to $\mathsf{n}\theta$, except for rule Rew.

Rule Comm performs communication between threads. Rule Out outputs a term to the external environment. Since $\mathsf{out}\ M$ may be handled by either Comm or Out, there is no guarantee that outputs have a corresponding input; instead,

Local Rules

$$\textsf{Comm}\ \frac{P@\theta_1 = \textsf{in}\ x\ .P' \qquad P@\theta_2 = \textsf{out}\ M\ .P'' \qquad \rho_1' = \rho_1[x \mapsto \rho_2(M)]}{\langle \rho_1, \theta_1\rangle, \langle \rho_2, \theta_2\rangle \xrightarrow{\textsf{comm}\ \theta_1, \theta_2, \rho_2(M)} \langle \rho_1', \textsf{n}\theta_1\rangle, \langle \rho_2, \textsf{n}\theta_2\rangle}$$

$$\textsf{Out}\ \frac{P@\theta = \textsf{out}\ M\ .P'}{\langle \rho, \theta\rangle \xrightarrow{\textsf{out}\ \theta, \rho(M)} \langle \rho, \textsf{n}\theta\rangle}$$

$$\textsf{Match}\ \frac{P@\theta = [x = y].P' \quad \rho(x) = \rho(y)}{\langle \rho, \theta\rangle \xrightarrow{\tau} \langle \rho, \textsf{n}\theta\rangle} \qquad \textsf{Let}\ \frac{P@\theta = \textsf{let}\ x = M\ .P'}{\langle \rho, \theta\rangle \xrightarrow{\tau} \langle \rho[x \mapsto \rho(M)], \textsf{n}\theta\rangle}$$

$$\textsf{Repl}\ \frac{P@\theta = \textsf{repl}.P'}{\langle \rho, \theta\rangle \xrightarrow{\tau} \langle \rho, \theta\rangle, \langle \rho, \textsf{n}\theta\rangle} \qquad \textsf{Rew}\ \frac{\rho(x) \to_{\mathcal{R}} M}{\langle \rho, \theta\rangle \xrightarrow{\tau} \langle \rho[x \mapsto M], \theta\rangle}$$

Global Rules

$$\textsf{New}\ \frac{P@\theta = \textsf{new}\ x\ .P' \quad \hat{x} = \textsf{genFresh}()}{\sigma, \langle \rho, \theta\rangle \xrightarrow{\tau} \sigma, \langle \rho[x \mapsto \hat{x}], \textsf{n}\theta\rangle} \qquad \textsf{Chk}\ \frac{P@\theta = \textsf{chk}.P'}{\sigma, \langle \rho, \theta\rangle \xrightarrow{\textsf{chk}} \langle \rho, \textsf{n}\theta\rangle}$$

Fig. 1. Multiset Rewriting Rules

they may simply cause a *barb*, i.e. an action observed only by the external environment. Note that there is no rule for input, and therefore processes can never receive a value from the environment – for studying security issues our processes will explicitly contain an adversary.

Rule Match, allows a process to continue only if x and y are bound to the same term. Rule Let simply updates ρ with the new binding. Rule Repl allows for spawning a new copy of P'. In Rew, the thread rewrites the term bound by x, thus performing an internal computation step; note that these internal steps may lead a matching to succeed.

Global rules instead look at the whole state. Rule New is not completely standard, and it generates a *fresh* value $\hat{x}$ for the variable x. Here we postulate that 1) a constant (nullary function) symbol $\bar{x}$ exists for each variable bound by new in P, and 2) two function symbols val, next exist, subject to no rewriting in $\mathcal{R}$. Note that we only need a finite number of such $\bar{x}$, since there are only finitely many variables in P and we have *no* α-conversion. When rule New is applied, the $\hat{x}$ is generated by a genFresh() primitive, which we assume to choose among $\textsf{val}(\bar{x}), \textsf{val}(\textsf{next}(\bar{x})), \textsf{val}(\textsf{next}(\textsf{next}(\bar{x}))), \ldots$. To make it possible to track new-generated values to their new prefix in P, we require that all new-bound variables are distinct, and therefore so are their related constants $\bar{x}$. Note that this representation prevents an adversary *Adv* to deduce any instance of $\hat{x}$ from other instances he knows, even if *Adv* can use val and next.

Rule Chk is peculiar: when a chk prefix is fired *all* the other threads are aborted, and the thread continues its execution alone. For simplicity, we admit only one firing of chk. We use this special prefix to model some kind of attacks. For instance, suppose we want to study the case in which the adversary learns some secret term S, maybe by corrupting some participant to the protocol. A straightforward way to model this attack would be simply adding out S to the

protocol, disclosing S. While this would work, in many cases giving this kind of power to the adversary might allow for trivial attacks. Instead, to keep the game fair, we could restrict the interaction between the adversary and the participants after the disclosure of S. For example, we could imagine that it would take a long time for the adversary to obtain S, and meanwhile the participants have terminated the protocol run, either normally or because of a time-out.

A possible usage of chk is the following. The adversary, after having learnt S, is only allowed to run alone, and possibly use this new knowledge to decrypt messages it learnt in the past. In our calculus, we model this scenario as

$$(Proto|Adv)|\mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ S\ .\mathsf{nil}|Adv)$$

Usually, the process Adv is chosen independently of the protocol, modeling the capabilities of any adversary, as we shall do in our examples.

Note that we include the adversary process twice. First, the adversary can interact with the protocol. Later, when chk is fired, the adversary can learn S and go on with its computation, without being able to communicate with the protocol participants. Since we want to allow the adversary to keep its knowledge across the chk firing, we simply save it in the variable $know$ before the chk, and make it again available to the adversary later on. Note that, while $know$ is only a single term, it can be a cons-list of all the terms known by the adversary. Therefore $know$ actually can bring all the old adversary knowledge into the new world, provided we have a primitive for pairing. In the next section, we show such a use of chk.

Another interesting use of chk we found is for modeling *timestamps*, as we will show in Sect. 5.3.

4.1 Diffie-Hellman Example

We consider the following key-exchange protocol, based on Diffie–Hellman [8].

1. $A \to \mathsf{all} : \mathsf{g}$
2. $A \to B : \{\mathsf{g}^{\mathsf{a}}\}_{\mathsf{k1}}$
3. $B \to A : \{\mathsf{g}^{\mathsf{b}}\}_{\mathsf{k2}}$
4. $A \to B : \{\mathsf{m}\}_{\mathsf{g}^{\mathsf{ab}}}$
5. $\ldots$
6. $A \to \mathsf{all} : \mathsf{k1}, \mathsf{k2}$

Initially, the principals A and B share two long term secret keys k1, k2, and agree on a public finite field $\mathsf{GF}[p]$ (where p is a large prime), and public generator g of $\mathsf{GF}[p]^*$. In the second step, principal A generates a nonce a and sends B the result of $\mathsf{g}^{\mathsf{a}} (\bmod p)$, encrypted with the key k1. In the third step, B does the same, with its own nonce b and key k2. Since both principals know the long term keys, they can compute $(\mathsf{g}^{\mathsf{b}})^{\mathsf{a}} = \mathsf{g}^{\mathsf{ab}} = (\mathsf{g}^{\mathsf{a}})^{\mathsf{b}} \ (\bmod p)$ and use this value as a session key to exchange the message m in the fourth step.

We study the robustness of this protocol against the active Dolev–Yao [9] adversary (such an adversary has full control over the public network, can reroute, discard or forge messages; further, he can apply any algebraic operation to terms learnt before). The adversary we use runs all the available operations in a non-deterministic way. Doing this, its behaviour encompasses that of any arbitrary Dolev-Yao adversary.

More in detail, we are interested in the *forward secrecy* of the message m. That is, we want m to be kept secret even though later on the long term keys k1, k2 are disclosed (last step).

We define the algebra by adapting the rewriting rules for encryption, multiplication, exponentiation, and inversion from [15]:

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{dec}(\mathsf{enc}(X,K),K) \Rightarrow X & \\ \mathsf{fst}(\mathsf{cons}(X,Y)) \Rightarrow X & \mathsf{snd}(\mathsf{cons}(X,Y)) \Rightarrow Y \\ \times(1,X) \Rightarrow X & \mathsf{exp}(\mathsf{inv}(X),Y) \Rightarrow \mathsf{inv}(\mathsf{exp}(X,Y)) \\ \times(X,Y) \Rightarrow \times(Y,X) & \times(X,\times(Y,Z)) \Rightarrow \times(\times(X,Y),Z) \\ \mathsf{exp}(X,1) \Rightarrow X & \mathsf{exp}(1,X) \Rightarrow 1 \\ \mathsf{inv}(\mathsf{inv}(X)) \Rightarrow X & \mathsf{exp}(\mathsf{exp}(X,Y),Z) \Rightarrow \mathsf{exp}(X,\times(Y,Z)) \\ \mathsf{inv}(1) \Rightarrow 1 & \mathsf{exp}(\times(Y,Z),X) \Rightarrow \times(\mathsf{exp}(Y,X),\mathsf{exp}(Z,X)) \\ \times(X,\mathsf{inv}(X)) \Rightarrow 1 & \mathsf{inv}(\times(X,Y)) \Rightarrow \times(\mathsf{inv}(X),\mathsf{inv}(Y)) \end{array} \right\}$$

Note that the algebra $\mathcal{A}$ defined by the above rewriting rules is not the same algebra of $\mathsf{GF}[p]^*$. In fact, $\mathcal{A}$ satisfies more equations than the ones that hold in $\mathsf{GF}[p]^*$. For instance, operations in $\mathcal{A}$ do not specify which modulus is being used; e.g., inversion modulo n is simply written as $\mathsf{inv}(X)$ rather than $\mathsf{inv}(X,n)$. Therefore, we have *(a)* $\times(X,\mathsf{inv}(X)) = 1$ and *(b)* $\mathsf{exp}(Y,\times(X,\mathsf{inv}(X))) = Y$. However, *(a)* holds in $\mathsf{GF}[p]^*$ only if $\mathsf{inv}(X)$ is performed modulo p, while *(b)* holds only if $\mathsf{inv}(X)$ is performed modulo $\varphi(p) = p-1$ (where φ is the Euler function). In spite of $\mathcal{A}$ being not equal to the $\mathsf{GF}[p]^*$ algebra, the equations that hold in $\mathsf{GF}[p]^*$ do hold in $\mathcal{A}$.

We use the following process:

$$\begin{aligned} P =& \mathsf{new}\ g\ .(DY|\mathsf{new}\ k1\ .\mathsf{new}\ k2\ .(Proto|Chk)) \\ Chk =& \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ k1\ .\mathsf{out}\ k2\ .\mathsf{nil}|DY)) \\ Proto =& A|B \\ A =& \mathsf{repl}.\mathsf{new}\ a\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,a),k1)\ .\mathsf{in}\ x\ . \\ & \quad \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k2),a)\ .\mathsf{out}\ \mathsf{enc}(\mathsf{m},k)\ .\mathsf{nil} \\ B =& \mathsf{repl}.\mathsf{new}\ b\ .\mathsf{in}\ x\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,b),k2)\ . \\ & \quad \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k1),b)\ .\mathsf{in}\ n\ .\mathsf{out}\ \mathsf{hash}(\mathsf{dec}(n,k))\ .\mathsf{nil} \end{aligned}$$

$$\begin{aligned} DY =& \mathsf{repl}.((\mathsf{new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{nil}|\mathsf{out}\ g\ .\mathsf{out}\ 1\ .\mathsf{nil})| \\ & \mathsf{in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ \mathsf{enc}(x,y)\ .\mathsf{out}\ \mathsf{dec}(x,y)\ .\mathsf{out}\ \mathsf{exp}(x,y)\ . \\ & \mathsf{out}\ \times(x,y)\ .\mathsf{out}\ \mathsf{inv}(x)\ .\mathsf{out}\ \mathsf{cons}(x,y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ . \\ & \mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ .\mathsf{nil}) \end{aligned}$$

The specification P combines the protocol participants with the DY adversary. The principal B outputs the hash of the exchanged message, just as a witness. We also add a Chk process for the explicit disclosure of the secret keys: of course, this only happens after the chk prefix is fired.

We expect P to ensure the secrecy of the message m. Further, we expect this secrecy property to still hold even after the chk fires and thus the long term keys $k1, k2$ are disclosed. This is the *forward secrecy* property.

5 Static Semantics

Our analysis over-approximates the values that processes exchange at run-time. These sets of values result from solving a set of constraints generated from a given process P.

We decided to represent these sets of values as the languages associates with the states of a finite tree automaton. Some of the constraints extracted from P can be expressed as transitions (e.g. $\{\mathsf{f}(\mathsf{g}(x), y) | x \in X \wedge y \in Y\} \subseteq Z$ becomes $\texttt{@z} \to \mathsf{f}(\mathsf{g}(\texttt{@x}), \texttt{@y})$), forming an automaton $\mathcal{A}$. The others are intersection constraints, and form a set $\mathcal{I}$, with typical element $\texttt{@a} \cap \texttt{@b} \subseteq \texttt{@c}$. Of course, we also require our sets of values be closed under rewritings in $\mathcal{R}$.

Our tool, supplied with $\mathcal{A}, \mathcal{I}, \mathcal{R}$, computes an automaton $\mathcal{F}$ such that its languages include those of $\mathcal{A}$, satisfy $\mathcal{I}$, and are closed under $\mathcal{R}$. Once done that, we can check a number of properties about P by simply inspecting $\mathcal{F}$.

We first give some intuition behind the construction of $\mathcal{A}$ and $\mathcal{I}$. Roughly, we follow the data-flow between processes depicted in Fig. 2. In the figure, the arrows towards/from processes represent inputs and outputs, respectively, while bullets represent the data-flow points which we focus on in our analysis. For each bullet, we compute an approximation for the set of values that flow through it.

More in detail, we generate a dedicated state of $\mathcal{A}$ for each bullet, and add transitions between states following the arrows in the figure. Formally, the states of $\mathcal{A}$ are:

- `@in`, `@out`, `@chk-in`, `@chk-out`;
- `@in-`$b\theta$ and `@out-`$b\theta$, for each θ such that $P@\theta = P_1 | P_2$, and $b \in \{\mathsf{l}, \mathsf{r}\}$;
- `@in-n`θ and `@out-n`θ, for each θ such that $P@\theta = \mathsf{repl}.P'$;
- `@inters-`θ, for each θ such that $P@\theta = [x = y].P'$;
- `@x` and `@x-val`, for each $\mathsf{new}\ x$ occurring in P;
- `@x`, for each $\mathsf{let}\ x = M$ and $\mathsf{in}\ x$ occurring in P.

We generate the transitions of the automaton $\mathcal{A}$ and the intersection constraints $\mathcal{I}$ using the *gen* function, recursively defined in Fig. 3. The expression *gen*

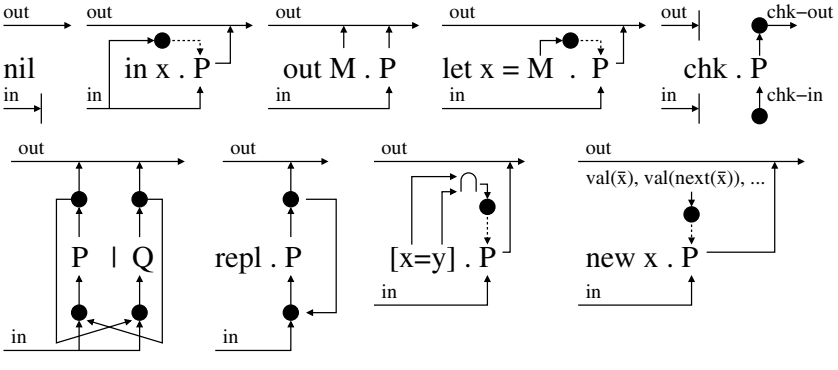

Fig. 2. Data Flow

$$gen(\theta, \mathsf{nil}, \zeta, in, out) = \emptyset$$
$$gen(\theta, \mathsf{in}\ x\ .P, \zeta, in, out) = (\texttt{@x} \to in), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out)$$
$$gen(\theta, \mathsf{out}\ M\ .P, \zeta, in, out) = (out \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta, in, out)$$
$$gen(\theta, (P|Q), \zeta, in, out) = (out \to \texttt{@out-l}\theta), (out \to \texttt{@out-r}\theta),$$
$$(\texttt{@in-l}\theta \to in), (\texttt{@in-l}\theta \to \texttt{@out-r}\theta), (\texttt{@in-r}\theta \to in), (\texttt{@in-r}\theta \to \texttt{@out-l}\theta),$$
$$gen(\mathsf{l}\theta, P, \zeta, \texttt{@in-l}\theta, \texttt{@out-l}\theta), gen(\mathsf{r}\theta, Q, \zeta, \texttt{@in-r}\theta, \texttt{@out-r}\theta)$$
$$gen(\theta, \mathsf{repl}.P, \zeta, in, out) = (out \to \texttt{@out-n}\theta), (\texttt{@in-n}\theta \to in),$$
$$(\texttt{@in-n}\theta \to \texttt{@out-n}\theta), gen(\mathsf{n}\theta, P, \zeta, \texttt{@in-n}\theta, \texttt{@out-n}\theta)$$
$$gen(\theta, \mathsf{let}\ x = M\ .P, \zeta, in, out) = (\texttt{@x} \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out)$$
$$gen(\theta, \mathsf{new}\ x\ .P, \zeta, in, out) = (\texttt{@x} \to \mathsf{val}(\texttt{@x-val})),$$
$$(\texttt{@x-val} \to \bar{x}), (\texttt{@x-val} \to \mathsf{next}(\texttt{@x-val})), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out)$$
$$gen(\theta, [x = y].P, \zeta, in, out) = (\zeta(x) \cap \zeta(y) \subseteq \texttt{@inters-}\theta),$$
$$gen(\mathsf{n}\theta, P, \zeta[x, y \mapsto \texttt{@inters-}\theta], in, out)$$
$$gen(\theta, \mathsf{chk}.P, \zeta, in, out) = gen(\mathsf{n}\theta, P, \zeta, \texttt{@chk-in}, \texttt{@chk-out})$$

Fig. 3. Extraction of $\mathcal{A}, \mathcal{I}$ from a process

$(\theta, P', \zeta, in, out)$ generates the transitions and intersection constraints for $P' = P@\theta$, a subprocess of P[1]. The *static environment* $\zeta \in \mathcal{X} \to \mathcal{Q}$ keeps track of which state of $\mathcal{A}$ is used to approximate the sets of values that can be dynamically bound to each variable in scope. The *in* and *out* parameters define the states for the approximation of the values that can be received and sent by P', respectively. Initially, *gen* is called as $gen(\epsilon, P, \emptyset, \texttt{@in}, \texttt{@out})$ to generate $\mathcal{A}, \mathcal{I}$ for the whole process P.

No productions are generated for nil. For $\mathsf{in}\ x\ .P'$, we generate a new state @x, and a transition from it to *in* to include inputs in its language. Then, we update ζ (the dotted line in Fig. 2) by binding x to @x, and proceed recursively with the continuation P'. Outputs as $\mathsf{out}\ M\ .P'$ generate a transition from the *out* state to $\zeta(M)$, the term obtained by replacing all the variables in M with their corresponding states; we then proceed recursively for P'. For example, the generated transitions for $P = \mathsf{in}\ x\ .\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{out}\ \mathsf{g}(x)\ .\mathsf{nil}$ are $\texttt{@x} \to \texttt{@in}, \texttt{@out} \to \mathsf{f}(\texttt{@x}), \texttt{@out} \to \mathsf{g}(\texttt{@x})$. Note that each output *contributes* to the language of @out by adding transitions to those already generated. This is depicted in Fig. 2 by the *out* arrow going straight from left to the right and collecting possible outputs from below. As seen in the figure, this happens for all processes, except for chk.

Parallel processes such as $P|Q$ are handled by creating four dedicated states for input and output of the left and right branch, then adding transitions to cross-connect inputs and outputs as in Fig. 2. Replication $\mathsf{repl}.P$ is done in a similar fashion, with a loopback transition.

For let bindings, we simply create a new state for the approximation of the bound value, and update ζ accordingly. A $\mathsf{new}\ x\ .P'$ causes the generation of

[1] The parameter P' of *gen* is actually redundant since it is determined by θ, but its presence allows for a simple definition.

transitions for the language $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$ using the two states `@x` and `@x-val`; then, we update ζ to bind x to this language.

A match $[x = y].P'$ creates a new state `@inters-`θ for the (approximation of the) intersection of values hold by x and y, together with the associated intersection constraint; in the analysis of P' we use this new state for both $\zeta(x)$ and $\zeta(y)$.

When a chk is fired, the continuation runs in an isolated world, therefore in the analysis we simply reset in, out to new independent states and proceed recursively. Note that ζ is not changed, and that bound variables bring their values into the new world (e.g. $\mathsf{in}\ x\ .\mathsf{chk.out}\ x\ .\mathsf{nil}$).

Note that our analysis generates no transitions for states `@in` and `@chk-in`: their language is therefore empty. In fact, top-level processes receive no value from their environment; this reflects the absence of an input rule.

Matching and Precision. Consider the following process:

$$P_1 = \mathsf{out}\ \mathsf{cons}(0,0)\ .\mathsf{out}\ \mathsf{cons}(1,1)\ .\mathsf{nil}\ \mid \\ \mathsf{in}\ x\ .\mathsf{let}\ f = \mathsf{fst}(x)\ .\mathsf{let}\ z = 0\ .[f = z].\mathsf{out}\ \mathsf{snd}(x)\ .\mathsf{nil}$$

At run-time, the last $\mathsf{out}\ \mathsf{snd}(x)$ can output 0, only. However, our analysis of the match $[f = z]$ does only refine the approximation of f and z, and not that of x. Therefore, in the analysis, a single state is used for the values of x before and after the match. The result of the analysis is that the last out may output either 0 or 1.

A more precise result can be obtained by using instead the following pattern:

$$P_2 = \mathsf{out}\ \mathsf{cons}(0,0)\ .\mathsf{out}\ \mathsf{cons}(1,1)\ .\mathsf{nil}\ \mid \\ \mathsf{in}\ x\ .\mathsf{let}\ y = \mathsf{cons}(0, \mathsf{snd}(x))\ .[x = y].\mathsf{out}\ \mathsf{snd}(x)\ .\mathsf{nil}$$

Here the analysis of the match refines the approximation of x itself, and therefore deduces that the last output can only be 0. We will use this style of matching in our examples.

5.1 Subject Reduction

Here, we establish the soundness for our analysis.

First, we define an address compatibility relation $\sim$ over addresses. Roughly speaking, $\theta_1 \sim \theta_2$ means that at run-time a thread running $P@\theta_1$ could communicate with a thread running $P@\theta_2$. The actual $\sim$ over-approximates run-time communication, and simply checks if the two addresses point to processes either at different branches of the same parallel, or under the same replication. We also take into account the presence of the chk prefix, since its continuation cannot interact with previously spawned threads.

More in detail, we say that chk *occurs* between θ and $\theta'\theta$ iff for some θ_a, θ_b we have $\theta' = \theta_a\theta_b$ and $P@\theta_b\theta = \mathsf{chk}.P'$. Then, the address compatibility relation $\sim$ is the minimum relation such that

- if chk does neither occur between $\mathsf{l}\theta$ and $\theta_l\mathsf{l}\theta$, nor between $\mathsf{r}\theta$ and $\theta_r\mathsf{r}\theta$, then $\theta_l\mathsf{l}\theta \sim \theta_r\mathsf{r}\theta$
- if $P@\theta = \mathsf{repl}.P'$ and chk does neither occur between θ and $\theta_1\theta$, nor between θ and $\theta_2\theta$, then $\theta_1\theta \sim \theta_2\theta$

The following lemma ensures that the relation $\sim$ actually encompasses all runtime communications. Its proof can be done by induction on the number of computation steps.

Lemma 1. *If* $P \rightarrow^* \xrightarrow{\mathsf{comm}\ \theta_1,\theta_2,M}$, *then* $\theta_1 \sim \theta_2$.

Input and output states related to compatible addresses satisfy the following inclusion property. The proof of this lemma is by structural induction on P.

Lemma 2. *If* $\theta_1 \sim \theta_2$, *then we have* $[\texttt{@out-}\theta_1]_{\mathcal{F}} \subseteq [\texttt{@in-}\theta_2]_{\mathcal{F}}$, *provided these states exist.*

The following theorem ensures that our analysis is sound, relating the dynamic semantics to the static one.

Theorem 1 (Subject Reduction). *Given* P, *let* $\mathcal{F}$ *be the automata resulting from the analysis. Assume* $\langle \emptyset, \epsilon \rangle \rightarrow^* \xrightarrow{\alpha} \sigma, \langle \rho, \theta \rangle$.

1. $\forall x \in \mathsf{dom}(\rho).\ \rho(x) \in [\texttt{@x}]_{\mathcal{F}}$
2. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was not fired before* α, *then* $M \in [\texttt{@out}]_{\mathcal{F}}$
3. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was fired before* α, *then* $M \in [\texttt{@chk-out}]_{\mathcal{F}}$

Proof (Sketch). By induction on the number of computation steps. First, we consider property (1): for this, we only need to check the rules that update the environment ρ.

When the Comm rule is applied, yielding to $\mathsf{comm}\ \theta_1, \theta_2, \rho_2(M)$, by Lemma 1 we have $\theta_1 \sim \theta_2$. We look for the transitions for $\mathsf{in}\ x$ and $\mathsf{out}\ M$ generated by $gen()$. These transitions have the form $\texttt{@x} \rightarrow in$ and $out \rightarrow \zeta(M)$, where $in = \texttt{@in-}\theta_i$ and $out = \texttt{@out-}\theta_o$. The addresses θ_i, θ_o, in general, are not the same as θ_1, θ_2, but they are strictly related so that we have also $\theta_i \sim \theta_o$. By Lemma 2, $\rho_1'(x) = \rho_2(M) \in [\texttt{@out-}\theta_o]_{\mathcal{F}} \subseteq [\texttt{@in-}\theta_i]_{\mathcal{F}} \subseteq [\texttt{@x}]_{\mathcal{F}}$, provided that $\zeta(M)$ is a correct approximation of $\rho_2(M)$. For this last proof obligation, we note that when there is no match involving variables in M, we have $\zeta(x) = \texttt{@x}$, so inductive hypothesis and structural induction on M suffices. Otherwise, if there is a match, we have $\zeta(x) = \texttt{@inters-}\theta_m$ for some x occurring in M. Here we first proceed by structural induction on P, obtaining $\rho_2(x) \in [\texttt{@inters-}\theta_m]_{\mathcal{F}}$, and then continue as for the no-match case. This shows that property (1) is preserved by Comm.

We now tackle property (1) for the other rules. The Let case is straightforward: we generated the transition $\texttt{@x} \rightarrow \zeta(M)$, so we have $\rho(M) \in [\texttt{@x}]_{\mathcal{F}}$. Rule New also poses no problem, because the fresh term returned by genFresh() is chosen among the terms in the language of @x. Finally, environment updates by rule Rew are harmless, the languages of $\mathcal{F}$ being closed under rewritings.

For properties (2,3), only rule Out may cause $\mathsf{out}\ \theta, M$. Here, structural induction on P is sufficient to show that $M \in [\texttt{@out-}\theta']_{\mathcal{F}}$, where $P@\theta'$ ranges from $P@\theta$ to i) the enclosing $\mathsf{chk}.P'$, if any, or otherwise to ii) the top level P. From this, we deduce $M \in [\texttt{@out}]_{\mathcal{F}}$ or $M \in [\texttt{@chk-out}]_{\mathcal{F}}$, depending on whether ii) or i) applies, respectively. □

5.2 Diffie-Hellman Example (Continued)

We ran the above analysis on the protocol specified in Sect. 4.1, computing the result $\mathcal{F}$. Our tool generated an $\mathcal{F}$ having 47 states and 865 transitions. Our analysis was able to establish forward secrecy, as $\mathsf{m} \notin [\texttt{@out-chk}]_{\mathcal{F}}$.

5.3 Kerberos

We now study a protocol involving timestamps. We chose a simplified version of Kerberos [20, 18].

In this protocol, a key exchange is performed by an authentication server AS, a client C and a server S. Initially, the authentication server shares long term keys with the client (kc) and with the server (ks). Upon request from the client, AS generates a fresh key kcs and sends it to the client encrypted with kc. Further, AS also provides a certificate for the freshness of kcs, made of the kcs key itself and the current time, both encrypted by ks. The server S can decrypt the certificate and ensure that kcs is indeed fresh by checking the timestamp. After that, C and S use kcs to exchange a session key $ksess$, and then proceed exchanging messages encrypted with $ksess$.

We study the rôle of timestamps in the protocol. To that purpose, we introduce a vulnerability in the server S. In our implementation, we let the server to disclose kcs, potentially mining the security of the protocol. However, to keep the game fair, disclosure may only happen after a long time since the timestamp for kcs has been generated. We model this through the occurrence of a chk. Hopefully, if timestamps are properly checked, disclosing a old kcs will not disrupt new sessions of the protocol.

In our specification, we abstract the actual timestamps values with two constants before and after. Initially, the protocol uses only before: any other timestamp value is considered not valid, being in the far past or far future. After chk, the before timestamp has expired, and the protocol has moved to newer timestamps, represented by after. Similarly, we use msg1 and msg2 for the messages exchanged by C and S before and after the chk, respectively.

We expect this faulty protocol implementation not to disclose msg1 until a chk occurs. After chk, we do expect msg1 to be disclosed, but we hope any new msg2 messages to be kept secret.

We specify the above as follows: (we omit parentheses in $P_1|\cdots|P_n$ for readability)

$$
\begin{aligned}
P =& DY|\mathsf{new}\ kc\ .\mathsf{new}\ ks\ .(AS|C|S)\\
AS =& \mathsf{repl.new}\ kcs\ .\mathsf{in}\ nonce\ .\mathsf{out}\ \mathsf{enc}(\mathsf{cons}(nonce, kcs), kc)\ .\\
& \quad \mathsf{out}\ \mathsf{enc}(\mathsf{cons}(kcs, \mathsf{before}), ks)\ .\mathsf{nil}
\end{aligned}
$$

$$
\begin{aligned}
C =& \mathsf{repl.new}\ nonce\ \mathsf{.out}\ nonce\ \mathsf{.in}\ ticket\ \mathsf{.in}\ cert\ .\\
& \mathsf{let}\ ticketCorrect = \mathsf{enc}(\mathsf{cons}(nonce, \mathsf{snd}(\mathsf{dec}(ticket, kc))), kc)\ .\\
& [ticket = ticketCorrect].\mathsf{let}\ kcs = \mathsf{snd}(\mathsf{dec}(ticket, kc))\ .\\
& \mathsf{new}\ ksess\ \mathsf{.out}\ \mathsf{enc}(ksess, kcs)\ \mathsf{.out}\ cert\ \mathsf{.out}\ \mathsf{enc}(\mathsf{msg1}, ksess)\ \mathsf{.nil}\\
S =& \mathsf{repl.in}\ tsess\ \mathsf{.in}\ cert\ \mathsf{.let}\ sess = \mathsf{dec}(cert, ks)\ .\\
& \mathsf{let}\ sessCorrect = \mathsf{cons}(\mathsf{fst}(sess), \mathsf{before})\ .[sess = sessCorrect].\\
& \mathsf{let}\ ksess = \mathsf{dec}(tsess, \mathsf{fst}(sess))\ \mathsf{.in}\ m\ \mathsf{.out}\ \mathsf{hash}(\mathsf{dec}(m, ksess))\ .Chk\\
Chk =& \mathsf{in}\ know\ \mathsf{.chk.(out}\ know\ \mathsf{.out}\ sess\ \mathsf{.nil}|AS'|C'|S'|DY)\\
DY =& \mathsf{repl.out\ before\ .out\ after\ .new}\ nonceDY\ \mathsf{.out}\ nonceDY\ \mathsf{.nil}|\\
& \mathsf{repl.in}\ x\ \mathsf{.in}\ y\ \mathsf{.out\ cons}(x, y)\ \mathsf{.out\ fst}(x)\ \mathsf{.out\ snd}(x)\ .\\
& \mathsf{out\ dec}(x, y)\ \mathsf{.out\ enc}(x, y)\ \mathsf{.out\ hash}(x)\ \mathsf{.out\ val}(x)\ \mathsf{.out\ next}(x)\ \mathsf{.nil}
\end{aligned}
$$

where AS', C', S' are the same as AS, C, S except that before is replaced with after, msg1 is replaced with msg2, and Chk is replaced with nil. As in the Diffie-Hellman example, our specification, once exchanged a message msg1 or msg2, output its hash.

Using our tool, we generated $\mathcal{F}$ (77 states, 1424 transitions) and verified that $\mathsf{msg1} \notin [\texttt{@out}]_{\mathcal{F}}$ and $\mathsf{msg2} \notin [\texttt{@out-chk}]$, thus establishing the wanted properties. On a side note, we also have $\mathsf{msg1} \in [\texttt{@out-chk}]$, as it should be, since msg1 is actually disclosed and our analysis is sound.

6 A Bit of Compositionality

Real-world systems often run many different protocols in a concurrent fashion. However, one usually studies the security properties of each protocol independently. This may not be enough to ensure the integrity of a system, since two otherwise safe protocols may have unwanted interactions, especially if the protocols share secrets. One would rather be able to derive properties about $P_1|P_2$ from the studies of P_1 and P_2.

Our analysis offers some opportunities for composing security results. Assume P_1 and P_2 were analyzed beforehand, yielding the automata $\mathcal{F}_1$ and $\mathcal{F}_2$. We can build an $\mathcal{F}$ for $P_1|P_2$ by merging the transitions of $\mathcal{F}_1$ and $\mathcal{F}_2$ and adding

$$
\begin{array}{ll}
\texttt{@in}_1 \to \texttt{@in} & \texttt{@in}_1 \to \texttt{@out}_2\\
\texttt{@in}_2 \to \texttt{@in} & \texttt{@in}_2 \to \texttt{@out}_1\\
\texttt{@out} \to \texttt{@out}_1 & \texttt{@out} \to \texttt{@out}_2
\end{array}
$$

just as it happens for the analysis of the parallel. Such an $\mathcal{F}$ is sound, provided that $[\texttt{@out}_1]_{\mathcal{F}_1} \subseteq [\texttt{@in}_2]_{\mathcal{F}_2}$ and $[\texttt{@out}_2]_{\mathcal{F}_2} \subseteq [\texttt{@in}_1]_{\mathcal{F}_1}$. This last proof obligations might be checked by static analysis. If the obligations do not hold (or cannot be proved), the completion algorithm can be restarted from the above $\mathcal{F}$ to compute a sound approximation. This could be less expensive than rebuilding the approximation from scratch, since parts of the work have been already done when computing $\mathcal{F}_1$ and $\mathcal{F}_2$.

7 Conclusion

We presented a simple model for the specification of cryptographic protocols, based on process calculi and term rewriting. We stress that we allow *any* rewriting system for defining the cryptographic primitives. Further, the model deals with some basic temporal aspects, and therefore it is suitable to express certain security properties involving time, such as forward secrecy.

We defined a static analysis for the verification of protocols so that it is closed under rewritings. The analysis focuses on foreseeing the protocol behaviour before and after a selected point in time, represented by the firing of chk. Also, we explored some opportunities for composing results of our analysis.

We implemented the analysis, and used our tool to check some significant protocols. The tool confirmed that we can handle complex rewriting rules, such that those of exponentials, and protocols involving timestamps.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01), pages 104-115.*, 2001.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.
3. AVISPA project home page. http://www.avispa-project.org.
4. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, 2005.
5. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with application to security. *Journal of Information and Computation*, 168(1):68–92, 2001.
6. Y. Boichut. Tree automata for security protocols (TA4SP) tool. http:// lifc.univ-fcomte.fr/ boichut/TA4SP/TA4SP.html.
7. I. Cervesato, N. A. Durgin, J. C. Mitchell, P. D. Lincoln, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13-th IEEE Computer Security Foundations Workshop*, pages 35–51, 2000.
8. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
9. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, 1983.
10. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 2004.
11. T. Genet, Y. T. Tang-Talpin, and V. V. T. Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proc. of Workshop on Issues in the Theory of Security*, 2003.
12. Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceeding of CADE*, pages 271–290, 2000.
13. Jean Goubault-Larrecq, Muriel Roger, and Kumar N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, August 2005.

14. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
15. J. K. Millen and V. Shmatikov. Symbolic protocol analysis with products and Diffie-Hellman exponentiation. In *Computer Security Foundations Workshop*, 2003.
16. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
17. David Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.
18. B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32:33–38, 1994.
19. F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *Electronic Notes in Theoretical Computer Science*, 62, 2002.
20. J. G. Steiner, B. C. Neuman, and J. I. Shiller. Kerberos: An authentication service for open network systems. In *Proc. of the Winter 1988 Usenix Conference*, pages 191–201, 1988.
21. Timbuk tree automata tool. http://www.irisa.fr/lande/genet/timbuk.
22. R. Zunino. Control flow analysis for the applied π–calculus. In *Proceedings of the MEFISTO Project 2003*, volume ENTCS 99, pages 87–110, 2004.
23. R. Zunino and P. Degano. Finite approximations of terms up to rewriting. http://www.di.unipi.it/ zunino/papers/completion.html.