A Computationally Sound Mechanized Prover for Security Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris
blanchet@di.ens.fr

Abstract

We present a new mechanized prover for secrecy properties of cryptographic protocols. In contrast to most previous provers, our tool does not rely on the Dolev-Yao model, but on the computational model. It produces proofs presented as sequences of games; these games are formalized in a probabilistic polynomial-time process calculus. Our tool provides a generic method for specifying security properties of the cryptographic primitives, which can handle shared- and public-key encryption, signatures, message authentication codes, and hash functions. Our tool produces proofs valid for a number of sessions polynomial in the security parameter, in the presence of an active adversary. We have implemented our tool and tested it on a number of examples of protocols from the literature.

1 Introduction

There exist two main frameworks for studying cryptographic protocols. In the computational model, messages are bitstrings, and the adversary is a probabilistic polynomial-time Turing machine. This model is close to the real execution of protocols, but the proofs are usually manual and informal. In contrast, in the formal, Dolev-Yao model, cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. The adversary can compute using these blackboxes. This abstract model makes it possible to build automatic verification tools, but the security proofs are in general not sound with respect to the computational model.

Since the seminal paper by Abadi and Rogaway [3], there has been much interest in relating both frameworks (see for example [1, 8, 11, 21, 25, 26, 35, 36]), to show the soundness of the Dolev-Yao model with respect to the computational model, and thus obtain automatic proofs of protocols in the computational model. However, this approach has limitations: since the computational and Dolev-Yao models do not correspond exactly, additional hypotheses are necessary in order to guarantee soundness. (For example, key cycles have to be excluded, or a specific security definition of encryption is needed [5].)

In this paper, we propose a different approach for automatically proving protocols in the computational model: we have built a mechanized prover that works directly in the computational model, without considering the Dolev-Yao model. Our tool produces proofs valid for a number of sessions polynomial in the security parameter, in the presence of an active adversary. These proofs are presented as sequences of games, as used by cryptographers [15, 42, 43]: the initial game represents the protocol to prove; the goal is to show that the probability of breaking a certain security property (secrecy in this paper) is negligible in this game; intermediate games are obtained each from the previous one by transformations such that the difference of probability between consecutive games is negligible; the final game is such that the desired probability is obviously negligible from the form of the game. The desired probability is then negligible in the initial game.

We represent games in a process calculus. This calculus is inspired by the pi-calculus, and by the calculi of [31, 32, 37] and of [30]. In this calculus, messages are bitstrings, and cryptographic primitives are functions from bitstrings to bitstrings. The calculus has a probabilistic semantics, and all processes run in polynomial time. The main tool for specifying security properties is observational equivalence: Q is observationally equivalent to Q', $Q \approx Q'$, when the adversary has a negligible probability of distinguishing Q from Q'. With respect to previous calculi mentioned above, our calculus introduces an important novelty which is key for the automatic proof of cryptographic protocols: the values of all variables during the execution of a process are stored in arrays. For instance, $x[i]$ is the value of x in the i-th copy of the process that defines x. Arrays replace lists often used by cryptographers in their manual proofs of protocols. For example, consider the definition of security of a message authentication code (mac). Informally, this definition says that the adversary has a negligible probability of forging a mac, that is, that all correct macs have been computed by calling the mac oracle. So, in cryptographic

proofs, one defines a list containing the arguments of calls to the mac oracle, and when checking a mac of a message m, one can additionally check that m is in this list, with a negligible change in probability. In our calculus, the arguments of the mac oracle are stored in arrays, and we perform a lookup in these arrays in order to find the message m. Arrays make it easier to automate proofs since they are always present in the calculus: one does not need to add explicit instructions to insert values in them, in contrast to the lists used in manual proofs. Therefore, many trivially sound but difficult to automate syntactic transformations disappear.

Our prover relies on a collection of game transformations, in order to transform the initial protocol into a game on which the desired security property is obvious. The most important kind of transformations comes from the definition of security of cryptographic primitives. As described in Section 3.2, these transformations can be specified in a generic way: we represent the definition of security of each cryptographic primitive by an observational equivalence $L \approx R$, where the processes L and R encode functions: they input the arguments of the function and send its result back. Then, the prover can automatically transform a process Q that calls the functions of L (more precisely, contains as subterms terms that perform the same computations as functions of L) into a process Q' that calls the functions of R instead. We have used this technique to specify several variants of shared- and public-key encryption, signature, message authentication codes, and hash functions, simply by giving the appropriate equivalence $L \approx R$ to the prover. Other game transformations are syntactic transformations, used in order to be able to apply the definition of cryptographic primitives, or to simplify the game obtained after applying these definitions.

In order to prove protocols, these game transformations are organized using a proof strategy based on advice: when a transformation fails, it suggests other transformations that should be applied before, in order to enable the desired transformation. Thanks to this strategy, protocols can often be proved in a fully automatic way. For delicate cases, our prover has an interactive mode, in which the user can manually specify the transformations to apply. It is usually sufficient to specify a few transformations coming from the security definitions of primitives, by indicating the concerned cryptographic primitive and the concerned secret key if any; the prover infers the intermediate syntactic transformations by the advice strategy. This mode is helpful for proving some public-key protocols, in which several security definitions of primitives can be applied, but only one leads to a proof of the protocol. Importantly, our prover is always sound: whatever indications the user gives, when the prover shows a security property of the protocol, the property indeed holds assuming the given hypotheses on the cryptographic primitives.

Our prover CryptoVerif has been implemented in Ocaml (9700 lines of code) and is available at `http://www.di.ens.fr/~blanchet/cryptoc-eng.html`.

Related Work Results that show the soundness of the Dolev-Yao model with respect to the computational model, e.g. [21, 26, 36], make it possible to use Dolev-Yao provers in order to prove protocols in the computational model. However, these results have limitations, in particular in terms of allowed cryptographic primitives (they must satisfy strong security properties so that they correspond to Dolev-Yao style primitives), and they require some restrictions on protocols (such as the absence of key cycles).

Several frameworks exist for formalizing proofs of protocols in the computational model. Backes, Pfitzmann, and Waidner [6, 8, 9] have designed an abstract cryptographic library including symmetric and public-key encryption, message authentication codes, signatures, and nonces and shown its soundness with respect to computational primitives, under arbitrary active attacks. Backes and Pfitzmann [7] relate the computational and formal notions of secrecy in the framework of this library. Recently, this framework has been used for a computationally-sound machine-checked proof of the Needham-Schroeder-Lowe protocol [44]. Canetti [19] introduced the notion of universal composability. With Herzog [20], they show how a Dolev-Yao-style symbolic analysis can be used to prove security properties of protocols within the framework of universal composability, for a restricted class of protocols using public-key encryption as only cryptographic primitive. Then, they use the automatic Dolev-Yao verification tool Proverif [16] for verifying protocols in this framework. Lincoln, Mateus, Mitchell, Mitchell, Ramanathan, Scedrov, and Teague [31, 32, 34, 37, 41] developed a probabilistic polynomial-time calculus for the analysis of cryptographic protocols. They define a notion of process equivalence for this calculus, derive compositionality properties, and define an equational proof system for this calculus. Datta, Derek, Mitchell, Shmatikov, and Turuani [22] have designed a computationally sound logic that enables them to prove computational security properties using a logical deduction system. These frameworks can be used to prove security properties of protocols in the computational sense, but except for [20] which relies on a Dolev-Yao prover, they have not been mechanized up to now, as far as we know.

Laud [28] designed an automatic analysis for proving secrecy for protocols using shared-key encryption, with passive adversaries. He extended it [29] to active adversaries, but with only one session of the protocol. This work is the closest to ours. We extend it considerably by handling more primitives, and a polynomial number of sessions.

Recently, Laud [30] designed a type system for proving security protocols in the computational model. This type

system handles shared- and public-key encryption, with an unbounded number of sessions. This system relies on the Backes-Pfitzmann-Waidner library. Type inference has not been implemented yet, and we believe that it would not be obvious to automate.

Barthe, Cerderquist, and Tarento [10, 45] have formalized the generic model and the random oracle model in the interactive theorem prover Coq, and proved signature schemes in this framework. In contrast to our specialized prover, proofs in generic interactive theorem provers require a lot of human effort, in order to build a detailed enough proof for the theorem prover to check it.

Halevi [24] explains that implementing an automatic prover based on sequences of games would be useful, and suggests ideas in this direction, but does not actually implement one.

Outline The next section presents our process calculus for representing games. Section 3 describes the game transformations that we use for proving protocols. Section 4 gives criteria for proving secrecy properties of protocols. Section 5 explains how the prover chooses which transformation to apply at each point. Section 6 presents our experimental results, and Section 7 concludes. The companion technical report [17] contains additional formal details, proof sketches, and details on the modeling of some cryptographic primitives.

Notations We recall the following standard notations. We denote by $\{M_1/x_1, \ldots, M_m/x_m\}$ the substitution that replaces x_j with M_j for each $j \leq m$. The cardinal of a set or multiset S is denoted $|S|$. If S is a finite set, $x \stackrel{R}{\leftarrow} S$ chooses a random element uniformly in S and assigns it to x. If $\mathcal{A}$ is a probabilistic algorithm, $x \leftarrow \mathcal{A}(x_1, \ldots, x_m)$ denotes the experiment of choosing random coins r and assigning to x the result of running $\mathcal{A}(x_1, \ldots, x_m)$ with coins r. Otherwise, $x \leftarrow M$ is a simple assignment statement.

2 A Calculus for Games

2.1 Syntax and Informal Semantics

The syntax of our calculus is summarized in Figure 1. We denote by η the security parameter, which determines in particular the length of keys.

This calculus assumes a countable set of channel names, denoted by c. There is a mapping maxlen_η from channels to integers, such that $\mathrm{maxlen}_\eta(c)$ is the maximum length of a message sent on channel c. Longer messages are truncated. For all c, $\mathrm{maxlen}_\eta(c)$ is polynomial in η. (This is key to guaranteeing that all processes run in probabilistic polynomial time.)

$M, N ::=$	terms
i	replication index
$x[M_1, \ldots, M_m]$	variable access
$f(M_1, \ldots, M_m)$	function application
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n} Q$	replication n times
$newChannel\ c; Q$	channel restriction
$c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$	input
$P ::=$	output process
$\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k\rangle; Q$	output
$new\ x[i_1, \ldots, i_m] : T; P$	random number
$let\ x[i_1, \ldots, i_m] : T = M\ in\ P$	assignment
$if\ M\ then\ P\ else\ P'$	conditional
$find\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ P_j)$ $else\ P$	array lookup

Figure 1. Syntax of the process calculus

Our calculus also assumes a set of parameters, denoted by n, which correspond to integer values polynomial in the security parameter, so we define $I_\eta(n) = q(\eta)$ where q is a polynomial, and $I_\eta(n)$ denotes the interpretation of n for a given value of the security parameter η.

Our calculus also assumes a set of types, denoted by T. For each value of the security parameter η, each type corresponds to a subset $I_\eta(T)$ of $Bitstring \cup \{\bot\}$ where $Bitstring$ is the set of all bitstrings and $\bot$ is a special symbol. The set $I_\eta(T)$ must be recognizable in polynomial time, that is, there exists an algorithm that decides whether $x \in I_\eta(T)$ in time polynomial in the length of x and the value of η. Let *fixed-length* types be types T such that $I_\eta(T)$ is the set of all bitstrings of a certain length, this length being a function of η bounded by a polynomial. Let *large* types be types T such that $\frac{1}{|I_\eta(T)|}$ is negligible. ($f(\eta)$ is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) Particular types are predefined: $bool$, such that $I_\eta(bool) = \{0, 1\}$, where 0 means false and 1 means true; $bitstring$, such that $I_\eta(bitstring) = Bitstring$; $bitstring_\bot$ such that $I_\eta(bitstring_\bot) = Bitstring \cup \{\bot\}$; $[1, n]$ where n is a parameter, such that $I_\eta([1, n]) = [1, I_\eta(n)]$. (We consider integers as bitstrings without leading zeroes.)

The calculus also assumes a finite set of function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \to T$. For each value of η, each function symbol f corresponds to a function $I_\eta(f)$ from $I_\eta(T_1) \times$

$\ldots \times I_\eta(T_m)$ to $I_\eta(T)$, such that $I_\eta(f)(x_1, \ldots, x_m)$ is computable in polynomial time in the lengths of $x_1, \ldots, x_m$ and the value of η. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test (both taking two values of the same type T and returning a value of type *bool*), $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation (taking and returning values of type *bool*).

In this calculus, terms represent computations on bitstrings. The replication index i is an integer which serves in distinguishing different copies of a replicated process $!^{i \leq n}$. (Replication indexes are typically used as array indexes.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indexes $M_1, \ldots, M_m$ of the array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \ldots, M_m)$ returns the result of applying function f to $M_1, \ldots, M_m$.

The calculus distinguishes two kinds of processes: input processes Q are ready to receive a message on a channel; output processes P output a message on a channel after executing some internal computations. The input process 0 does nothing; $Q \mid Q'$ is the parallel composition of Q and Q'; $!^{i \leq n} Q$ represents n copies of Q in parallel, each with a different value of $i \in [1, n]$; $newChannel\ c; Q$ creates a new private channel c and executes Q; the semantics of the input $c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ will be explained below together with the semantics of the output.

The output process $new\ x[i_1, \ldots, i_m] : T; P$ chooses a new random number uniformly in $I_\eta(T)$, stores it in $x[i_1, \ldots, i_m]$, and executes P. (T must be a fixed-length type, because probabilistic polynomial-time Turing machines can choose random numbers uniformly only in such types.) Function symbols represent deterministic functions, so all random numbers must be chosen by $new\ x[i_1, \ldots, i_m] : T$. Deterministic functions make automatic syntactic manipulations easier: we can duplicate a term without changing its value. The process $let\ x[i_1, \ldots, i_m] : T = M\ in\ P$ stores the bitstring value of M (which must be in $I_\eta(T)$) in $x[i_1, \ldots, i_m]$, and executes P. The process $if\ M\ then\ P\ else\ P'$ executes P if M evaluates to 1 and P' if M evaluates to 0. Next, we explain the process $find\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ P_j)\ else\ P$, where $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_{m'}$. The order and array indexes on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ can be further abbreviated $\widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$. A simple example is the following: $find\ u \leq n\ suchthat\ defined(x[u]) \wedge x[u] = a\ then\ P'\ else\ P$ tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this *find* construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the *find* construct allows us to access arrays, which is key for our purpose. More generally, $find\ u_1[\widetilde{i}] \leq n_1, \ldots, u_m[\widetilde{i}] \leq n_m\ suchthat\ defined(M_1, \ldots, M_l) \wedge M\ then\ P'\ else\ P$ tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ are defined and M is true. In case of success, it executes P'. In case of failure, it executes P. This is further generalized to m branches: $find\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ P_j)\ else\ P$ tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions $defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ for each j and each value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is 1, it executes P. Otherwise, it chooses randomly with uniform[1] probability one j and one value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is 1, and executes P_j.

Finally, let us explain the output $\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k\rangle; Q$. A channel $c[M_1, \ldots, M_l]$ consists of both a channel name c and a tuple of terms $M_1, \ldots, M_l$. Channel names c allow us to define private channels to which the adversary can never have access, by $newChannel\ c$. (This is useful in the proofs, although all channels of protocols are often public.) Terms $M_1, \ldots, M_l$ are intuitively analogous to IP addresses and ports which are numbers that the adversary may guess. Two channels are equal when they have the same channel name and terms that evaluate to the same bitstrings. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes $\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k\rangle; Q$, one looks for an input on the same channel and with the same arity in the available input processes. If no such input process is found, the process blocks. Otherwise, one such input process $c[M'_1, \ldots, M'_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ is chosen randomly with uniform probability. The communication is then executed: for each $j \leq k$, the output message N_j is evaluated, its result is truncated to length $\mathrm{maxlen}_\eta(c)$, the obtained bitstring is stored in $x_j[\widetilde{i}]$ if it is in $I_\eta(T_j)$ (otherwise the process blocks). Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution. Note that the syntax requires an output to be followed by an input process, as in [30]. If one needs to output several messages consecutively, one can simply

[1] A probabilistic polynomial-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, there exist approximate algorithms: for example, in order to obtain a random integer in $[0, m-1]$, we can choose a random integer r uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $r \mod m$. The distribution can be made as close as we wish to the uniform distribution by choosing k large enough.

insert fictitious inputs between the outputs. The adversary can then schedule the outputs by sending messages to these inputs.

Using different channels for each input and output allows the adversary to control the network. For instance, we may write $!^{i \leq n} c[i](x[i] : T) \ldots \overline{c'[i]}\langle M \rangle \ldots$ The adversary can then decide which copy of the replicated process receives its message, simply by sending it on $c[i]$ for the appropriate value of i.

We write *if* M *then* P as an abbreviation for *if* M *then* P *else* $\overline{yield}\langle\rangle; 0$, and similarly for a *find* without *else* clause. ("*else* 0" would not be syntactically correct.) A trailing 0 after an output may be omitted.

Variables can be defined by assignments, inputs, restrictions, and array lookups. The *current replication indexes* at a certain program point in a process are $i_1, \ldots, i_m$ where the replications above the considered program point are $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$. We often abbreviate $x[i_1, \ldots, i_m]$ by x when $i_1, \ldots, i_m$ are the current replication indexes, but it should be kept in mind that this is only an abbreviation. Variables defined under a replication must be arrays: for example $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$ *let* $x[i_1, \ldots, i_m] : T = M$ *in* $\ldots$ More formally, we require the following invariant:

Invariant 1 (Single definition) The process Q_0 satisfies Invariant 1 if and only if

1. in a definition of $x[i_1, \ldots, i_m]$ in Q_0, the indexes $i_1, \ldots, i_m$ of x are the current replication indexes at that definition, and

2. two different definitions of the same variable x in Q_0 are in different branches of a *if* or a *find*.

Invariant 1 guarantees that each variable is assigned at most once for each value of its indexes. (Indeed, item 2 shows that only one definition of each variable can be executed for given indexes in each trace.)

Invariant 2 (Defined variables) The process Q_0 satisfies Invariant 2 if and only if every occurrence of a variable access $x[M_1, \ldots, M_m]$ in Q_0 is either

- syntactically under the definition of $x[M_1, \ldots, M_m]$ (in which case $M_1, \ldots, M_m$ are in fact the current replication indexes at the definition of x);

- or in a *defined* condition in a *find* process;

- or in M'_j or P_j in a process of the form *find* $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\tilde{i}] \leq \widetilde{n_j}$ *suchthat defined*$(M'_{j1}, \ldots, M'_{jl_j}) \wedge M'_j$ *then* $P_j)$ *else* P where for some $k \leq l_j$, $x[M_1, \ldots, M_m]$ is a subterm of M'_{jk}.

Invariant 2 guarantees that variables can be accessed only when they have been initialized. It checks that the definition of the variable access is either in scope (first item) or checked by a *find* (last item). Both invariants are checked by the prover for the initial game, and preserved by all game transformations.

We say that a function $f : T_1 \times \ldots \times T_m \to T$ is *poly-injective* when it is injective and its inverses can be computed in polynomial time, that is, there exist functions $f_j^{-1} : T \to T_j$ $(1 \leq j \leq m)$ such that $f_j^{-1}(f(x_1, \ldots, x_m)) = x_j$ and f_j^{-1} can be computed in polynomial time in the length of $f(x_1, \ldots, x_m)$ and in the security parameter. When f is poly-injective, we define a pattern matching construct *let* $f(x_1, \ldots, x_m) = M$ *in* P *else* Q as an abbreviation for *let* $y : T = M$ *in let* $x_1 : T_1 = f_1^{-1}(y)$ *in* $\ldots$ *let* $x_m : T_m = f_m^{-1}(y)$ *in if* $f(x_1, \ldots, x_m) = y$ *then* P *else* Q. We naturally generalize this construct to *let* $N = M$ *in* P *else* Q where N is built from poly-injective functions and variables.

Let us introduce two cryptographic primitives that we use in the following.

Definition 1 Let T_{mr}, T_{mk}, and T_{ms} be types that correspond intuitively to random seeds, keys, and message authentication codes, respectively; T_{mr} is a fixed-length type. A message authentication code [14] consists of three function symbols:

- $mkgen : T_{mr} \to T_{mk}$ where $I_\eta(mkgen) = mkgen_\eta$ is the key generation algorithm taking as argument a random bitstring, and returning a key. (Usually, $mkgen$ is a randomized algorithm; here, since we separate the choice of random numbers from computation, $mkgen$ takes an additional argument representing the random coins.)

- $mac : bitstring \times T_{mk} \to T_{ms}$ where $I_\eta(mac) = mac_\eta$ is the mac algorithm taking as argument a message and a key, and returning the corresponding tag. (We assume here that mac is deterministic; we could easily encode a randomized mac by adding an additional argument as for $mkgen$.)

- $check : bitstring \times T_{mk} \times T_{ms} \to bool$ where $I_\eta(check) = check_\eta$ is a checking algorithm such that $check_\eta(m, k, t) = 1$ if and only if t is a valid mac of message m under key k. (Since mac is deterministic, $check_\eta(m, k, t)$ is typically $mac_\eta(m, k) = t$.)

We have $\forall m \in Bitstring, \forall r \in I_\eta(T_{mr}), check_\eta(m, mkgen_\eta(r), mac_\eta(m, mkgen_\eta(r))) = 1$.

A mac is secure against existential forgery under chosen

message attack if and only if for all polynomials q,

$$\max_{\mathcal{A}} \Pr[r \stackrel{R}{\leftarrow} I_\eta(T_{mr}); k \leftarrow mkgen_\eta(r); \\ (m,t) \leftarrow \mathcal{A}^{mac_\eta(.,k),check_\eta(.,k,.)} : check_\eta(m,k,t)]$$

is negligible, where the adversary $\mathcal{A}$ is any probabilistic Turing machine, running in time $q(\eta)$, with oracle access to $mac_\eta(.,k)$ and $check_\eta(.,k,.)$, and $\mathcal{A}$ has not called $mac_\eta(.,k)$ on message m.

Definition 2 Let T_r and T'_r be fixed-length types; let T_k and T_e be types. A symmetric encryption scheme [12] (stream cipher) consists of three function symbols $kgen : T_r \rightarrow T_k$, $enc : bitstring \times T_k \times T'_r \rightarrow T_e$, and $dec : T_e \times T_k \rightarrow bitstring_\perp$, with $I_\eta(kgen) = kgen_\eta$, $I_\eta(enc) = enc_\eta$, $I_\eta(dec) = dec_\eta$, such that for all $m \in Bitstring$, $r \in I_\eta(T_r)$, and $r' \in I_\eta(T'_r)$, $dec_\eta(enc_\eta(m, kgen_\eta(r), r'), kgen_\eta(r)) = m$.

Let $LR(x,y,b) = x$ if $b = 0$ and $LR(x,y,b) = y$ if $b = 1$, defined only when x and y are bitstrings of the same length. A stream cipher is IND-CPA (satisfies indistinguishability under chosen plaintext attacks) if and only if for all polynomials q,

$$\max_{\mathcal{A}} 2\Pr[b \stackrel{R}{\leftarrow} \{0,1\}; r \stackrel{R}{\leftarrow} I_\eta(T_r); k \leftarrow kgen_\eta(r); \\ b' \leftarrow \mathcal{A}^{r' \stackrel{R}{\leftarrow} I_\eta(T'_r); enc_\eta(LR(.,.,b),k,r')} : b' = b] - 1$$

is negligible, where the adversary $\mathcal{A}$ is any probabilistic Turing machine, running in time $q(\eta)$, with oracle access to the left-right encryption algorithm which given two bitstrings a_0 and a_1 of the same length, returns $r' \stackrel{R}{\leftarrow} I_\eta(T'_r)$; $enc_\eta(LR(a_0,a_1,b),k,r')$, that is, encrypts a_0 when $b = 0$ and a_1 when $b = 1$.

Example 1 Let us consider the following trivial protocol:

$$A \rightarrow B : e, mac(e, x_{mk}) \quad \text{where } e = enc(x'_k, x_k, x''_r) \\ \text{and } x''_r, x'_k \text{ are fresh random numbers}$$

A and B are assumed to share a key x_k for a stream cipher and a key x_{mk} for a message authentication code. A creates a fresh key x'_k, and sends it encrypted under x_k to B. A mac is appended to the message, in order to guarantee integrity. The goal of the protocol is that x'_k should be a secret key shared between A and B. This protocol can be modeled in our calculus by the following process Q_0:

$$Q_0 = start(); new\ x_r : T_r; let\ x_k : T_k = kgen(x_r)\ in \\ new\ x'_r : T_{mr}; let\ x_{mk} : T_{mk} = mkgen(x'_r)\ in \\ \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A[i](); new\ x'_k : T_k; new\ x''_r : T'_r; \\ let\ x_m : bitstring = enc(k2b(x'_k), x_k, x''_r)\ in \\ \overline{c_A[i]}\langle x_m, mac(x_m, x_{mk})\rangle$$

$$Q_B = !^{i' \leq n} c_B[i'](x'_m, x_{ma}); if\ check(x'_m, x_{mk}, x_{ma}) \\ then\ let\ i_\perp(k2b(x''_k)) = dec(x'_m, x_k)\ in\ \overline{c_B[i']}\langle\rangle$$

When Q_0 receives a message on channel $start$, it begins execution: it generates the keys x_k and x_{mk} by choosing random coins x_r and $x_{r'}$ and applying the appropriate key generation algorithms. Then it yields control to the context (the adversary), by outputting on channel c. After this output, n copies of processes for A and B are ready to be executed, when the context outputs on channels $c_A[i]$ or $c_B[i]$ respectively. In a session that runs as expected, the context first sends a message on $c_A[i]$. Then Q_A creates a fresh key x'_k (T_k is assumed to be a fixed-length type), encrypts it under x_k with random coins x''_r, computes the mac of the encryption under x_{mk}, and sends the ciphertext and the mac on $c_A[i]$. The function $k2b : T_k \rightarrow bitstring$ is the natural injection $I_\eta(k2b)(x) = x$; it is needed only for type conversion. The context is then expected to forward this message on $c_B[i]$. When Q_B receives this message, it checks the mac, decrypts, and stores the obtained key in x''_k. (The function $i_\perp : bitstring \rightarrow bitstring_\perp$ is the natural injection; it is useful to check that decryption succeeded.) This key x''_k should be secret.

The context is responsible for forwarding messages from A to B. It can send messages in unexpected ways in order to mount an attack.

Although we use a trivial running example due to length constraints, this example is sufficient to illustrate the main features of our prover. Section 6 presents results obtained on more realistic protocols.

We denote by $\mathrm{var}(P)$ the set of variables that occur in P, and by $\mathrm{fc}(P)$ the set of free channels of P. (We use similar notations for input processes.)

2.2 Type System

We use a type system to check that bitstrings of the proper type are passed to each function, and that array indexes are used correctly.

To be able to type variable accesses used not under their definition (such accesses are guarded by a *find* construct), the type-checking algorithm proceeds in two passes. In the first pass, we build a type environment $\mathcal{E}$, which maps variable names x to types $[1,n_1] \times \ldots \times [1,n_m] \rightarrow T$, where the definition of $x[i_1,\ldots,i_m]$ of type T occurs under replications $!^{i_1 \leq n_1}, \ldots, !^{i_m \leq n_m}$. We require that all definitions of the same variable x yield the same value of $\mathcal{E}(x)$, so that $\mathcal{E}$ is properly defined.

In the second pass, the process is typechecked in the type environment $\mathcal{E}$ by a simple type system. This type system is detailed in [17]. It defines the judgment $\mathcal{E} \vdash Q$ which means that the process Q is well-typed in environment $\mathcal{E}$.

Invariant 3 (Typing) The process Q_0 satisfies Invariant 3 if and only the type environment $\mathcal{E}$ for Q_0 is well-defined, and $\mathcal{E} \vdash Q_0$.

We require the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : T \to T'$ to bypass the type system. Similarly, the type system does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions. The type system just makes explicit which set of bitstrings may appear at each point of the protocol.

2.3 Formal Semantics

The semantics is defined by a probabilistic reduction relation formally detailed in [17]. The notation $E, M \Downarrow a$ means that the term M evaluates to the bitstring a in environment E. We denote by $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle]$ the probability that Q outputs the bitstring a on channel c after some reductions.

Our semantics is such that, for each process Q, there exists a probabilistic polynomial time Turing machine that simulates Q. (Processes run in polynomial time since the number of processes created by a replication and the length of messages sent on channels are bounded by polynomials.) Conversely, our calculus can simulate a probabilistic polynomial-time Turing machine, simply by choosing coins by new and by applying a function symbol defined to perform the same computations as the Turing machine.

2.4 Observational Equivalence

A context is a process containing a hole $[\,]$. An evaluation context C is a context built from $[\,]$, $newChannel\ c; C$, $Q \mid C$, and $C \mid Q$. We use an evaluation context to represent the adversary. We denote by $C[Q]$ the process obtained by replacing the hole $[\,]$ in the context C with the process Q.

Definition 3 (Observational equivalence) Let Q and Q' be two processes, and V a set of variables. Assume that Q and Q' satisfy Invariants 1, 2, and 3 and the variables of V are defined in Q and Q', with the same types.

An evaluation context is said to be *acceptable* for Q, Q', V if and only if $\mathrm{var}(C) \cap (\mathrm{var}(Q) \cup \mathrm{var}(Q')) \subseteq V$ and $C[Q]$ satisfies Invariants 1, 2, and 3. (Then $C[Q']$ also satisfies these invariants.)

We say that Q and Q' are *observationally equivalent* with public variables V, written $Q \approx^V Q'$, when for all evaluation contexts C acceptable for Q, Q', V, for all channels c and bitstrings a, $|\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] - \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]|$ is negligible.

Intuitively, the goal of the adversary represented by context C is to distinguish Q from Q'. When it succeeds, it performs a different output, for example $\overline{c}\langle 0\rangle$ when it has recognized Q and $\overline{c}\langle 1\rangle$ when it has recognized Q'. When $Q \approx^V Q'$, the context has negligible probability of distinguishing Q from Q'.

The unusual requirement on variables of C comes from the presence of arrays and of the associated $find$ construct which gives C direct access to variables of Q and Q': the context C is allowed to access variables of Q and Q' only when they are in V. (In more standard settings, the calculus does not have constructs that allow the context to access variables of Q and Q'.) The following result is not difficult to prove:

Lemma 1 *$\approx^V$ is an equivalence relation, and $Q \approx^V Q'$ implies that $C[Q] \approx^{V'} C[Q']$ for all evaluation contexts C acceptable for Q, Q', V and all $V' \subseteq V \cup (\mathrm{var}(C) \setminus (\mathrm{var}(Q) \cup \mathrm{var}(Q')))$.*

We denote by $Q \approx_0^V Q'$ the particular case in which for all evaluation contexts C acceptable for Q, Q', V, for all channels c and bitstrings a, $\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. When V is empty, we write $Q \approx Q'$ instead of $Q \approx^V Q'$ and $Q \approx_0 Q'$ instead of $Q \approx_0^V Q'$.

3 Game Transformations

In this section, we describe the game transformations that allow us to transform the process that represents the initial protocol into a process on which the desired security property can be proved directly, by criteria given in Section 4. These transformations are parametrized by the set V of variables that the context can access. As we shall see in Section 4, V contains variables that we would like to prove secret. These transformations transform a process Q_0 into a process Q_0' such that $Q_0 \approx^V Q_0'$.

3.1 Syntactic Transformations

RemoveAssign(x): When x is defined by an assignment $let\ x[i_1, \ldots, i_l] : T = M\ in\ P$, we replace x with its value. Precisely, the transformation is performed only when x does not occur in M (non-cyclic assignment). When x has several definitions, we simply replace $x[i_1, \ldots, i_l]$ with M in P. (For accesses to x guarded by $find$, we do not know which definition of x is actually used.) When x has a single definition, we replace everywhere in the game $x[M_1, \ldots, M_l]$ with $M\{M_1/i_1, \ldots, M_l/i_l\}$. We additionally update the $defined$ conditions of $find$ to preserve Invariant 2, and to maintain the requirement that $x[M_1, \ldots, M_l]$ is defined when it was required in the initial game. When $x \in V$, its definition is kept unchanged.

Otherwise, when x is not referred to at all after the transformation, we remove the definition of x. When x is referred to only at the root of *defined* tests, we replace its definition with a constant. (The definition point of x is important, but not its value.)

Example 2 In the process of Example 1, the transformation **RemoveAssign**(x_{mk}) substitutes $mkgen(x'_r)$ for x_{mk} in the whole process and removes the assignment $let\ x_{mk} : T_{mk} = mkgen(x'_r)$. After this substitution, $mac(x_m, x_{mk})$ becomes $mac(x_m, mkgen(x'_r))$ and $check(x'_m, x_{mk}, x_{ma})$ becomes $check(x'_m, mkgen(x'_r), x_{ma})$, thus exhibiting terms required in Section 3.2. The situation is similar for **RemoveAssign**(x_k).

SArename(x): The transformation **SArename** (single assignment rename) aims at renaming variables so that each variable has a single definition in the game; this is useful for distinguishing cases depending on which definition of x has set $x[\tilde{i}]$. This transformation can be applied only when $x \notin V$. When x has $m > 1$ definitions, we rename each definition of x to a different variable $x_1, \ldots, x_m$. Terms $x[\tilde{i}]$ under a definition of $x_j[\tilde{i}]$ are then replaced with $x_j[\tilde{i}]$. Each branch of find $FB = \tilde{u}[\tilde{i}] \leq \tilde{n}\ suchthat\ defined(M'_1, \ldots, M'_{l'}) \wedge M\ then\ P$ where $x[M_1, \ldots, M_l]$ is a subterm of some M'_k for $k \leq l'$ is replaced with m branches $FB\{x_j[M_1, \ldots, M_l]/x[M_1, \ldots, M_l]\}$ for $1 \leq j \leq m$.

Simplify: The prover uses a simplification algorithm, based on an equational prover, using an algorithm similar to the Knuth-Bendix completion [27]. This equational prover uses:

- User-defined equations, of the form $\forall x_1 : T_1, \ldots, \forall x_m : T_m, M$ which mean that for all environments E, if for all $j \leq m$, $E(x_j) \in I_\eta(T_j)$, then $E, M \Downarrow 1$. For example, considering mac and stream ciphers as in Definitions 1 and 2 respectively, we have:

$$\forall r : T_{mr}, \forall m : bitstring, \\ check(m, mkgen(r), mac(m, mkgen(r))) = 1 \qquad (mac)$$

$$\forall m : bitstring; \forall r : T_r, \forall r' : T'_r, \\ dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m) \qquad (enc)$$

  We express the poly-injectivity of the function $k2b$ of Example 1 by

$$\forall x : T_k, \forall y : T_k, (k2b(x) = k2b(y)) = (x = y) \\ \forall x : T_k, k2b^{-1}(k2b(x)) = x \qquad (k2b)$$

  where $k2b^{-1}$ is a function symbol that denotes the inverse of $k2b$. We have similar formulas for $i_\perp$.

- Equations that come from the process. For example, in the process $if\ M\ then\ P\ else\ P'$, we have $M = 1$ in P and $M = 0$ in P'.

- The low probability of collision between random values. For example, when x is defined by $new\ x : T$ and T is a large type, $x[M_1, \ldots, M_m] = x[M'_1, \ldots, M'_m]$ implies $M_1 = M'_1, \ldots, M_m = M'_m$ up to negligible probability.

The prover combines these properties to simplify terms, and uses simplified forms of terms to simplify processes. For example, if M simplifies to 1, then $if\ M\ then\ P\ else\ P'$ simplifies to P.

Proposition 1 *Let Q_0 be a process that satisfies Invariants 1, 2, and 3, and Q'_0 the process obtained from Q_0 by one of the transformations above. Then Q'_0 satisfies Invariants 1, 2, and 3, and $Q_0 \approx^V Q'_0$.*

3.2 Applying the Definition of Security of Primitives

The security of cryptographic primitives is defined using observational equivalences given as axioms. Importantly, this formalism allows us to specify many different primitives in a generic way. Such equivalences are then used by the prover in order to transform a game into another, observationally equivalent game, as explained in the following of this section.

The primitives are specified using equivalences of the form $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ where G is defined by the following grammar, with $l \geq 0$ and $m \geq 1$:

$G ::=$	group of functions
$!^{i \leq n} new\ y_1 : T_1; \ldots; new\ y_l : T_l; (G_1, \ldots, G_m)$	replication, restrictions
$(x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP$	function

$FP ::=$	functional processes
M	term
$new\ x[\tilde{i}] : T; FP$	random number
$let\ x[\tilde{i}] : T = M\ in\ FP$	assignment
$if\ M\ then\ FP_1\ else\ FP_2$	test
$find\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\tilde{i}] \leq \widetilde{n_j}\ suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ FP_j)\ else\ FP$	array lookup

Intuitively, $(x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP$ represents a function that takes as argument values $x_1, \ldots, x_l$ of types $T_1, \ldots, T_l$ respectively, and returns a result computed by FP. The observational equivalence $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ expresses that the adversary has a negligible probability of distinguishing functions in the left-hand

$$[\![(G_1, \ldots, G_m)]\!] = [\![G_1]\!]^1 \mid \ldots \mid [\![G_m]\!]^m$$

$$[\![!^{i \leq n} new\ y_1 : T_1; \ldots; new\ y_l : T_l; (G_1, \ldots, G_m)]\!]^{\widetilde{j}}_{\widetilde{i}} =$$
$$!^{i \leq n} c_{\widetilde{j}}[\widetilde{i}, i](); new\ y_1 : T_1; \ldots; new\ y_l : T_l; \overline{c_{\widetilde{j}}[\widetilde{i}, i]}\langle\rangle; ([\![G_1]\!]^{\widetilde{j},1}_{\widetilde{i},i} \mid \ldots \mid [\![G_m]\!]^{\widetilde{j},m}_{\widetilde{i},i})$$

$$[\![(x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP]\!]^{\widetilde{j}}_{\widetilde{i}} = c_{\widetilde{j}}[\widetilde{i}](x_1 : T_1, \ldots, x_l : T_l); [\![FP]\!]^{\widetilde{j}}_{\widetilde{i}}$$

$$[\![M]\!]^{\widetilde{j}}_{\widetilde{i}} = \overline{c_{\widetilde{j}}[\widetilde{i}]}\langle M \rangle$$

$$[\![new\ x[\widetilde{i}] : T; FP]\!]^{\widetilde{j}}_{\widetilde{i}} = new\ x[\widetilde{i}] : T; [\![FP]\!]^{\widetilde{j}}_{\widetilde{i}}$$

$$[\![let\ x[\widetilde{i}] : T = M\ in\ FP]\!]^{\widetilde{j}}_{\widetilde{i}} = let\ x[\widetilde{i}] : T = M\ in\ [\![FP]\!]^{\widetilde{j}}_{\widetilde{i}}$$

$$[\![if\ M\ then\ FP_1\ else\ FP_2]\!]^{\widetilde{j}}_{\widetilde{i}} = if\ M\ then\ [\![FP_1]\!]^{\widetilde{j}}_{\widetilde{i}}\ else\ [\![FP_2]\!]^{\widetilde{j}}_{\widetilde{i}}$$

$$[\![find\ (\textstyle\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}\ suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ FP_j)\ else\ FP]\!]^{\widetilde{j}}_{\widetilde{i}} =$$
$$find\ (\textstyle\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}\ suchthat\ defined(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ then\ [\![FP_j]\!]^{\widetilde{j}}_{\widetilde{i}})\ else\ [\![FP]\!]^{\widetilde{j}}_{\widetilde{i}}$$

where $c_{\widetilde{j}}$ are pairwise distinct channels, $\widetilde{i} = i_1, \ldots, i_{l'}$, and $\widetilde{j} = j_0, \ldots, j_{l'}$.

Figure 2. Translation from functional processes to processes

side from corresponding functions in the right-hand side. Formally, functions can be encoded as processes that input their arguments and output their result on a channel, as shown in Figure 2. The translation of $!^{i \leq n} new\ y_1 : T_1; \ldots; new\ y_l : T_l; (G_1, \ldots, G_m)$ inputs and outputs on channel $c_{\widetilde{j}}$ so that the context can trigger the generation of random numbers $y_1, \ldots, y_l$. The translation of $(x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP$ inputs the arguments of the function on channel $c_{\widetilde{j}}$ and translates FP, which outputs the result of FP on $c_{\widetilde{j}}$. (In the left-hand side, the result FP of functions must simply be a term M.) The observational equivalence $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ is then an abbreviation for $[\![(G_1, \ldots, G_m)]\!] \approx [\![(G'_1, \ldots, G'_m)]\!]$.

For example, the security of a mac (Definition 1) is represented by the equivalence $L \approx R$ where:

$$\begin{aligned} L = {} & !^{i'' \leq n''} new\ r : T_{mr}; (\\ & !^{i \leq n}(x : bitstring) \rightarrow mac(x, mkgen(r)), \\ & !^{i' \leq n'}(m : bitstring, ma : T_{ms}) \rightarrow \\ & \quad check(m, mkgen(r), ma)) \\ R = {} & !^{i'' \leq n''} new\ r : T_{mr}; (\\ & !^{i \leq n}(x : bitstring) \rightarrow mac'(x, mkgen'(r)), \\ & !^{i' \leq n'}(m : bitstring, ma : T_{ms}) \rightarrow \\ & \quad find\ u \leq n\ suchthat\ defined(x[u]) \wedge (m = x[u]) \\ & \quad \wedge check'(m, mkgen'(r), ma)\ then\ 1\ else\ 0) \end{aligned} \qquad (mac_{\text{eq}})$$

where mac', $check'$, and $mkgen'$ are function symbols with the same types as mac, $check$, and $mkgen$ respectively. (We use different function symbols on the left- and right-hand sides, just to prevent a repeated application of the transformation induced by this equivalence. Since we add these function symbols, we also add the equation

$$\forall r : T_{mr}, \forall m : bitstring,$$
$$check'(m, mkgen'(r), mac'(m, mkgen'(r))) = 1 \qquad (mac')$$

which restates Equation (mac) for mac', $check'$, and $mkgen'$.) Intuitively, the equivalence $L \approx R$ leaves mac computations unchanged (except for the use of primed function symbols in R), and allows one to replace a mac checking $check(m, mkgen(r), ma)$ with a lookup in the array x of messages whose mac has been computed with key $mkgen(r)$: if m is found in the array x and $check(m, mkgen(r), ma)$, we return 1; otherwise, the check fails (up to negligible probability), so we return 0. (If the check succeeded with m not in the array x, the adversary would have forged a mac.) Obviously, the form of L requires that r is used only to compute or check macs, for the equivalence to be correct. Formally, the following result shows the correctness of our modeling. It is a fairly easy consequence of Definition 1.

Proposition 2 *Assuming $(mkgen, mac, c)$ is a message authentication code secure against existential forgery under chosen message attack, $I_\eta(mkgen') = I_\eta(mkgen)$, $I_\eta(mac') = I_\eta(mac)$, and $I_\eta(check') = I_\eta(check)$, then $[\![L]\!] \approx [\![R]\!]$.*

Similarly, we represent the security of a IND-CPA

stream cipher (Definition 2) by the equivalence:

$$\begin{aligned} &!^{i' \leq n'} \mathit{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \\ &\qquad \mathit{new}\ r' : T'_r; \mathit{enc}(x, \mathit{kgen}(r), r') \\ \approx &!^{i' \leq n'} \mathit{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \\ &\qquad \mathit{new}\ r' : T'_r; \mathit{enc}'(Z(x), \mathit{kgen}'(r), r') \end{aligned} \qquad (\mathit{enc}_{\mathrm{eq}})$$

where enc' and kgen' are function symbols with the same types as enc and kgen respectively, and $Z : \mathit{bitstring} \rightarrow \mathit{bitstring}$ is the function that returns a bitstring of the same length as its argument, consisting only of zeroes. Using equations such as $\forall x : T, Z(\mathit{T2b}(x)) = Z_T$, we can prove that $Z(\mathit{T2b}(x))$ does not depend on x when x is of a fixed-length type and $\mathit{T2b} : T \rightarrow \mathit{bitstring}$ is the natural injection. The representation of other primitives can be found in [17].

We use such equivalences $L \approx R$ in order to transform a process Q_0 observationally equivalent to $C[[\![L]\!]]$ into a process Q'_0 observationally equivalent to $C[[\![R]\!]]$, for some evaluation context C. In order to check that $Q_0 \approx C[[\![L]\!]]$, the prover uses sufficient conditions, which essentially guarantee that all uses of certain secret variables of Q_0, in a set S, can be implemented by calling functions of L. Let $\mathcal{M}$ be a set of occurrences of terms, corresponding to uses of variables of S. Informally, the prover shows the following properties. The variables of S occur only in terms of $\mathcal{M}$. For each $M \in \mathcal{M}$, there exist a term N_M, result of a function of L, and a substitution σ_M such that $M = \sigma_M N_M$. (Precisely, σ_M applies to the abbreviated form of N_M in which we write x instead of $x[\widetilde{i}]$.) Let $\widetilde{i}$ and $\widetilde{i'}$ be the sequences of current replication indexes at N_M in L and at M in Q_0, respectively. There exists a function mapIdx_M that maps the array indexes at M in Q_0 to the array indexes at N_M in L: the evaluation of M when $\widetilde{i'} = \widetilde{a}$ will correspond in $C[[\![L]\!]]$ to the evaluation of N_M when $\widetilde{i} = \mathit{mapIdx}_M(\widetilde{a})$. Thus, σ_M and mapIdx_M induce a correspondence between Q_0 and L: for all $M \in \mathcal{M}$, for all $x[\widetilde{i''}]$ that occur in N_M, $(\sigma_M x)\{\widetilde{a}/\widetilde{i'}\}$ corresponds to $x[\widetilde{i''}]\{\mathit{mapIdx}_M(\widetilde{a})/\widetilde{i}\}$, that is, $(\sigma_M x)\{\widetilde{a}/\widetilde{i'}\}$ in a trace of Q_0 has the same value as $x[\widetilde{i''}]\{\mathit{mapIdx}_M(\widetilde{a})/\widetilde{i}\}$ in the corresponding trace of $C[[\![L]\!]]$ ($\widetilde{i''}$ is a prefix of $\widetilde{i}$).

For example, consider a process Q_0 that contains $M_1 = \mathit{enc}(M'_1, \mathit{kgen}(x_r), x'_r[i_1])$ and $M_2 = \mathit{enc}(M'_2, \mathit{kgen}(x_r), x''_r[i_2])$ with $i_1 \leq n_1$, $i_2 \leq n_2$, and x_r, x'_r, x''_r bound by restrictions. Let $S = \{x_r, x'_r, x''_r\}$, $\mathcal{M} = \{M_1, M_2\}$, and $N_{M_1} = N_{M_2} = \mathit{enc}(x[i', i], \mathit{kgen}(r[i']), r'[i', i])$. The functions mapIdx_{M_1} and mapIdx_{M_2} are defined by $\mathit{mapIdx}_{M_1}(a_1) = (1, a_1)$ for $a_1 \in [1, I_\eta(n_1)]$ and $\mathit{mapIdx}_{M_2}(a_2) = (1, a_2 + I_\eta(n_1))$ for $a_2 \in [1, I_\eta(n_2)]$. Then $M'_1\{a_1/i_1\}$ corresponds to $x[1, a_1]$, x_r to $r[1]$, $x'_r[a_1]$ to $r'[1, a_1]$, $M'_2\{a_2/i_2\}$ to $x[1, a_2 + I_\eta(n_1)]$, and $x''_r[a_2]$ to $r'[1, a_2 + I_\eta(n_1)]$. The functions mapIdx_{M_1} and mapIdx_{M_2} are such that $x_{r'}[a_1]$ and $x_{r''}[a_2]$ never correspond to the same cell of r'; indeed, $x_{r'}[a_1]$ and $x_{r''}[a_2]$ are independent random numbers in Q_0, so their images in $C[[\![L]\!]]$ must also be independent random numbers.

The above correspondence must satisfy the following soundness conditions: when x is a function argument in L, the term that corresponds to $x[\widetilde{a'}]$ must have the same type as $x[\widetilde{a'}]$, and when two terms correspond to the same $x[\widetilde{a'}]$, they must evaluate to the same value; when x is bound by $\mathit{new}\ x : T$ in L, the term that corresponds to $x[\widetilde{a'}]$ must be evaluate to $z[\widetilde{a''}]$ where $z \in S$ and z is bound by $\mathit{new}\ z : T$ in Q_0, and the relation that associates $z[\widetilde{a''}]$ to $x[\widetilde{a'}]$ is an injective function. (It is easy to check that, in the previous example, these conditions are satisfied.)

The transformation of Q_0 into Q'_0 consists in two steps. First, we replace the restrictions that define variables of S with restrictions that define fresh variables corresponding to variables bound by new in R. The correspondence between variables of Q_0 and variables $C[[\![L]\!]]$ is extended to include these fresh variables. Second, we reorganize Q_0 so that each evaluation of a term $M \in \mathcal{M}$ first stores the values of the arguments $x_1, \ldots, x_m$ of the function $(x_1 : T_1, \ldots, x_m : T_m) \rightarrow N_M$ in fresh variables, then computes N_M and stores its result in a fresh variable, and uses this variable instead of M; then we simply replace the computation of N_M with the corresponding functional process of R, taking into account the correspondence of variables.

The full formal description of this transformation is given in [17]. The following proposition shows the soundness of the transformation.

Proposition 3 *Let Q_0 be a process that satisfies Invariants 1, 2, and 3, and Q'_0 the process obtained from Q_0 by the above transformation. Then Q'_0 satisfies Invariants 1, 2, and 3, and if $[\![L]\!] \approx [\![R]\!]$ for all polynomials $\mathrm{maxlen}_\eta(c_{j_0,\ldots,j_l})$ and $I_\eta(n)$ where n is any replication bound of L or R, then $Q_0 \approx^V Q'_0$.*

Example 3 In order to treat Example 1, the prover is given as input the indication that T_{mr}, T_r, T'_r, and T_k are fixed-length types; the type declarations for the functions $\mathit{mkgen}, \mathit{mkgen}' : T_{mr} \rightarrow T_{mk}$, $\mathit{mac}, \mathit{mac}' : \mathit{bitstring} \times T_{mk} \rightarrow T_{ms}$, $\mathit{check}, \mathit{check}' : \mathit{bitstring} \times T_{mk} \times T_{ms} \rightarrow \mathit{bool}$, $\mathit{kgen}, \mathit{kgen}' : T_r \rightarrow T_k$, $\mathit{enc}, \mathit{enc}' : \mathit{bitstring} \times T_k \times T'_r \rightarrow T_e$, $\mathit{dec} : T_e \times T_k \rightarrow \mathit{bitstring}_\perp$, $k2b : T_k \rightarrow \mathit{bitstring}$, $i_\perp : \mathit{bitstring} \rightarrow \mathit{bitstring}_\perp$, $Z : \mathit{bitstring} \rightarrow \mathit{bitstring}$, and the constant $Z_k : \mathit{bitstring}$; the equations (mac), (mac'), (enc), and $\forall x : T_k, Z(k2b(x)) = Z_k$ (which expresses that all keys have the same length); the indication that $k2b$ and $i_\perp$ are poly-injective (which generates the equations $(k2b)$ and similar equations for $i_\perp$); equiva-

lences $L \approx R$ for mac (mac_{eq}) and encryption (enc_{eq}); and the process Q_0 of Example 1.

The prover first applies **RemoveAssign**(x_{mk}) to the process Q_0 of Example 1, as described in Example 2. The process can then be transformed using the security of the mac. We take $S = \{x'_r\}$, $M_1 = mac(x_m[i], mkgen(x'_r))$, $M_2 = check(x'_m[i'], mkgen(x'_r), x_{ma}[i'])$, and $\mathcal{M} = \{M_1, M_2\}$. We have $N_{M_1} = mac(x[i'', i], mkgen(r[i'']))$, $N_{M_2} = check(m[i'', i'], mkgen(r[i'']), ma[i'', i'])$, $mapIdx_{M_1}(a_1) = (1, a_1)$, and $mapIdx_{M_2}(a_2) = (1, a_2)$, so $x_m[a_1]$ corresponds to $x[1, a_1]$, x'_r to $r[1]$, $x'_m[a_2]$ to $m[1, a_2]$, and $x_{ma}[a_2]$ to $ma[1, a_2]$.

After transformation, we get the following process Q'_0:

$$
\begin{aligned}
&Q'_0 = start(); new\ x_r : T_r; let\ x_k : T_k = kgen(x_r)\ in\\
&\quad new\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q'_A \mid Q'_B)\\
&Q'_A = !^{i \le n} c_A[i](); new\ x'_k : T_k; new\ x''_r : T'_r;\\
&\quad let\ x_m : bitstring = enc(k2b(x'_k), x_k, x''_r)\ in\\
&\quad \overline{c_A[i]}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle\\
&Q'_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad find\ u \le n\ suchthat\ defined(x_m[u]) \wedge x'_m = x_m[u] \wedge\\
&\qquad\qquad check'(x'_m, mkgen'(x'_r), x_{ma})\ then\\
&\qquad (if\ 1\ then\ let\ i_\perp(k2b(x''_k)) = dec(x'_m, x_k)\ in\ \overline{c_B[i']}\langle\rangle)\\
&\quad else\\
&\qquad (if\ 0\ then\ let\ i_\perp(k2b(x''_k)) = dec(x'_m, x_k)\ in\ \overline{c_B[i']}\langle\rangle)
\end{aligned}
$$

The initial definition of x'_r is removed and replaced with a new definition, which we still call x'_r. The term $mac(x_m, mkgen(x'_r))$ is replaced with $mac'(x_m, mkgen'(x'_r))$. The term $check(x'_m, mkgen(x'_r), x_{ma})$ becomes $find\ u \le n\ suchthat\ defined(x_m[u]) \wedge x'_m = x_m[u] \wedge check'(x'_m, mkgen'(x'_r), x_{ma})\ then\ 1\ else\ 0$ which yields Q'_B after transformation of functional processes into processes. The process looks up the message x'_m in the array x_m, which contains the messages whose mac has been computed with key $mkgen(x'_r)$. If the mac of x'_m has never been computed, the check always fails (it returns 0) by the definition of security of the mac. Otherwise, it returns 1 when $check'(x'_m, mkgen'(x'_r), x_{ma})$.

After applying **Simplify**, Q'_A is unchanged and Q'_B becomes

$$
\begin{aligned}
&Q'_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad find\ u \le n\ suchthat\ defined(x_m[u], x'_k[u]) \wedge\\
&\qquad x'_m = x_m[u] \wedge check'(x'_m, mkgen'(x'_r), x_{ma})\ then\\
&\quad let\ x''_k : T_k = x'_k[u]\ in\ \overline{c_B[i']}\langle\rangle
\end{aligned}
$$

First, the tests $if\ 1\ then\ \ldots$ and $if\ 0\ then\ \ldots$ are simplified. The term $dec(x'_m, x_k)$ is simplified knowing $x'_m = x_m[u]$ by the *find* condition, $x_m[u] = enc(k2b(x'_k[u]), x_k, x''_r[u])$ by the assignment that defines x_m, $x_k = kgen(x_r)$ by the assignment that defines x_k, and $dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$ by (*enc*). So we have $dec(x'_m, x_k) = i_\perp(k2b(x'_k[u]))$. By injectivity of $i_\perp$ and $k2b$, the assignment to x''_k simply becomes $x''_k = x'_k[u]$, using the equations $\forall x : bitstring, i_\perp^{-1}(i_\perp(x)) = x$. and $\forall x : T_k$, $k2b^{-1}(k2b(x)) = x$.

After applying **RemoveAssign**(x_k), one can apply the security of encryption: $enc(k2b(x'_k), kgen(x_r), x''_r)$ becomes $enc'(Z(k2b(x'_k)), kgen(x_r), x''_r)$. After **Simplify**, it becomes $enc'(Z_k, kgen(x_r), x''_r)$, using $\forall x : T_k$, $Z(k2b(x)) = Z_k$ (which expresses that all keys have the same length).

Using lists instead of arrays simplifies this transformation: we do not need to add instructions that insert values in the list, since all variables are always implicitly arrays. Moreover, if there are several occurrences of $mac(x_i, k)$ with the same key in the initial process, each $check(m_j, k, ma_j)$ is replaced with a *find* with one branch for each occurrence of mac. Therefore, the prover distinguishes automatically the cases in which the checked mac ma_j comes from each occurrence of mac, that is, it distinguishes cases depending on the value of i such that $m_j = x_i$. Typically, distinguishing these cases is useful in the following of the proof of the protocol. (A similar situation arises for other cryptographic primitives specified using *find*.)

4 Criteria for Proving Secrecy Properties

Let us now define syntactic criteria that allow us to prove secrecy properties of protocols.

We use $if\ defined(M)\ then\ P$ as syntactic sugar for $find\ \ suchthat\ defined(M) \wedge 1\ then\ P\ else\ \overline{yield}\langle\rangle$.

Definition 4 (One-session secrecy) The process Q *preserves the one-session secrecy of* x when $Q \mid Q_x \approx Q \mid Q'_x$, where

$$
\begin{aligned}
Q_x &= c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]);\\
&\qquad if\ defined(x[u_1, \ldots, u_m])\ then\ \overline{c}\langle x[u_1, \ldots, u_m]\rangle\\
Q'_x &= c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]);\\
&\qquad if\ defined(x[u_1, \ldots, u_m])\ then\ new\ y : T; \overline{c}\langle y\rangle
\end{aligned}
$$

$c \notin \text{fc}(Q)$, $u_1, \ldots, u_m \notin \text{var}(Q)$, and $\mathcal{E}(x) = [1, n_1] \times \ldots \times [1, n_m] \to T$.

Intuitively, the adversary cannot distinguish a process that outputs the value of the secret from one that outputs a random number. The adversary performs a single test query, modeled by Q_x and Q'_x.

Proposition 4 (One-session secrecy) *Consider a process Q such that there exists a set of variables S such that 1) the definitions of x are either restrictions new $x[\widetilde{i}] : T$ and $x \in S$, or assignments let $x[\widetilde{i}] : T = z[M_1, \ldots, M_l]$ where z is defined by restrictions new $z[i'_1, \ldots, i'_l] : T$, and $z \in S$, and 2) all accesses to variables $y \in S$ in Q are of the form "let $y'[\widetilde{i}] : T' = y[M_1, \ldots, M_l]$" with $y' \in S$. Then $Q \mid Q_x \approx_0 Q \mid Q'_x$, hence Q preserves the one-session secrecy of x.*

Intuitively, only the variables in S depend on the restriction that defines x; the sent messages and the control flow of the process are independent of x, so the adversary obtains no information on x. In the implementation, the set S is computed by fixpoint iteration, starting from x or z and adding variables y' defined by "*let* $y'[\widetilde{i}] : T' = y[M_1, \ldots, M_l]$" when $y \in S$.

Definition 5 (Secrecy) The process Q *preserves the secrecy of* x when $Q \mid R_x \approx Q \mid R'_x$, where

$R_x = !^{i \leq n} c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]);$
if defined$(x[u_1, \ldots, u_m])$ *then* $\overline{c}\langle x[u_1, \ldots, u_m]\rangle$
$R'_x = !^{i \leq n} c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]);$
if defined$(x[u_1, \ldots, u_m])$ *then*
find $u' \leq n$ *suchthat defined*$(y[u'], u_1[u'], \ldots, u_m[u'])$
$\quad \wedge u_1[u'] = u_1 \wedge \ldots \wedge u_m[u'] = u_m$
then $\overline{c}\langle y[u']\rangle$ *else new* $y : T; \overline{c}\langle y\rangle$

$c \notin \mathrm{fc}(Q), u_1, \ldots, u_m, u' \notin \mathrm{var}(Q), \mathcal{E}(x) = [1, n_1] \times \ldots \times [1, n_m] \to T$, and $I_\eta(n) \geq I_\eta(n_1) \times \ldots \times I_\eta(n_m)$.

Intuitively, the adversary cannot distinguish a process that outputs the value of the secret for several indexes from one that outputs independent random numbers. In this definition, the adversary can perform several test queries, modeled by R_x and R'_x. This corresponds to the "real-or-random" definition of security [4]. (As shown in [4], this notion is stronger than the more standard approach in which the adversary can perform a single test query and some reveal queries, which always reveal $x[u_1, \ldots, u_m]$.)

Proposition 5 (Secrecy) *Assume that Q satisfies the hypothesis of Proposition 4.*

When $\mathcal{T}$ is a trace of $C[Q]$ for some evaluation context C, we define $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}])$*, the defining restriction of $x[\widetilde{a}]$ in trace $\mathcal{T}$, as follows: if $x[\widetilde{a}]$ is defined by* new $x[\widetilde{a}] : T$ *in $\mathcal{T}$,* $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = x[\widetilde{a}]$*; if $x[\widetilde{a}]$ is defined by* let $x[\widetilde{a}] : T = z[M_1, \ldots, M_l]$, $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = z[a'_1, \ldots, a'_l]$ *where $E, M_k \Downarrow a'_k$ for all $k \leq l$ and E is the environment in $\mathcal{T}$ at the definition of $x[\widetilde{a}]$.*

Assume that for all evaluation contexts C acceptable for Q, 0, $\{x\}$, the probability $\Pr[\mathcal{T} \wedge \widetilde{a} \neq \widetilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$ *is negligible. Then Q preserves the secrecy of x.*

The hypothesis can be verified using simplification (see **Simplify** in Section 3.1). Intuitively, the required condition guarantees that when $\widetilde{a} \neq \widetilde{a'}$, we have $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) \neq \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])$ except in cases of negligible probability, so $x[\widetilde{a}]$ and $x[\widetilde{a'}]$ are defined by different restrictions so they are independent random numbers. This notion of secrecy composed with correspondence assertions [46] can be used to prove security of a key exchange. (Correspondence assertions are properties of the form "if some event $e(\widetilde{M})$ has been executed then some events $e_i(\widetilde{M_i})$ for $i \leq m$ have been executed".) We postpone this point to a future paper, since we do not present the verification of correspondence assertions in this paper. (This verification is currently being implemented.)

Lemma 2 *If $Q \approx^{\{x\}} Q'$ and Q preserves the one-session secrecy of x then Q' preserves the one-session secrecy of x. The same result holds for secrecy.*

We can then apply the following technique. When we want to prove that Q_0 preserves the (one-session) secrecy of x, we transform Q_0 by the transformations described in Section 3 with $V = \{x\}$. By Propositions 1 and 3, we obtain a process Q'_0 such that $Q_0 \approx^V Q'_0$. We use Propositions 4 or 5 to show that Q'_0 preserves the (one-session) secrecy of x, and finally conclude that Q_0 also preserves the (one-session) secrecy of x by Lemma 2.

Example 4 After the transformations of Example 3, the only variable access to x'_k in the considered process is *let* $x''_k : T_k = x'_k[u]$ and x''_k is not used in the considered process. So by Proposition 4, the considered process preserves the one-session secrecy of x''_k (with $S = \{x'_k, x''_k\}$). By Lemma 2, the process of Example 1 also preserves the one-session secrecy of x''_k. However, this process does not preserve the secrecy of x''_k, because the adversary can force several sessions of B to use the same key x''_k, by replaying the message sent by A. (Accordingly, the hypothesis of Proposition 5 is not satisfied.)

The criteria given in this section might seem restrictive, but in fact, they should be sufficient for all protocols, provided the previous transformation steps are powerful enough to transform the protocol into a simpler protocol, on which these criteria can then be applied.

5 Proof Strategy

Up to now, we have described the available game transformations. Next, we explain how we organize these transformations in order to prove protocols.

At the beginning of the proof, and after each successful cryptographic transformation (that is, a transformation of Section 3.2), the prover executes **Simplify**, and tests whether the desired security properties are proved, as described in Section 4. If so, it stops.

In order to perform the cryptographic transformations and the other syntactic transformations, our proof strategy relies of the idea of advice. Precisely, the prover tries to execute each available cryptographic transformation in turn. When such a cryptographic transformation fails, it returns some syntactic transformations that could make the desired transformation work. (These are the advised transformations.) Then the prover tries to perform these syntactic transformations. If they fail, they may also suggest other advised transformations, which are then executed. When the syntactic transformations finally succeed, we retry the desired cryptographic transformation, which may succeed or fail, perhaps with new advised transformations, and so on.

The prover determines the advised transformations using the following main conditions:

- Assume that we try to execute a cryptographic transformation, and need to recognize a certain term M of L, but we find in Q_0 only part of M, the other parts being variable accesses $x[\ldots]$ while we expect function applications. In this case, we advise **RemoveAssign**(x). For example, if Q_0 contains $enc(M', x_k, x'_r)$ and we look for $enc(x_m, kgen(x_r), x_{r'})$, we advise **RemoveAssign**(x_k). If Q_0 contains $let\ x_k = mkgen(x_r)$ and we look for $mac(x_m, mkgen(x_r))$, we also advise **RemoveAssign**(x_k). (The transformation of Example 2 is advised for this reason.)

- When we try to execute **RemoveAssign**(x), x has several definitions, and there are accesses to variable x guarded by *find* in Q_0, we advise **SArename**(x).

- When we check whether x is secret or one-session secret, we have an assignment $let\ x[\widetilde{i}] : T = y[\widetilde{M}]\ in\ P$, and there is at least one assignment defining y, we advise **RemoveAssign**(y).

6 Experimental Results

We have successfully tested our prover on a number of protocols of literature. All these protocols have been tested in a configuration in which the honest participants are willing to run sessions with the adversary, and we prove secrecy of keys for sessions between honest participants. In these examples, shared-key encryption is encoded using a stream cipher and a mac as in Example 1, public-key encryption is assumed to be IND-CCA2 (indistinguishability under adaptive chosen-ciphertext attacks) [13], public-key signature is assumed to be secure against existential forgery.

Otway-Rees [40]: We automatically prove the secrecy of the exchanged key.

Yahalom [18]: For the original version of the protocol, our prover cannot show one-session secrecy of the exchanged key, because the protocol is not secure, at least using encrypt-then-mac as definition of encryption. Indeed, there is a confirmation round $\{N_B\}_K$ where K is the exchanged key. This message may reveal some information on K. After removing this confirmation round, our prover shows the one-session secrecy of K. However, it cannot show the secrecy of K, since in the absence of a confirmation round, the adversary may force several sessions of Yahalom to use the same key.

Needham-Schroeder shared-key [38]: Our prover shows one-session secrecy of the exchanged key. It does not prove the secrecy of the exchanged key, since there is a well known attack [23] in which the adversary forces several sessions of the protocol to use the same key. Our prover shows the secrecy for the corrected version [39].

Denning-Sacco public-key [23]: Our prover cannot show the one-session secrecy of the exchanged key, since there is an attack against this protocol [2]. One-session secrecy of the exchanged key is proved for the corrected version [2]. Secrecy is not proved since the adversary can force several sessions of the protocol to use the same key. (We do not model timestamps in this protocol.) In contrast to the previous examples, we give the main proof steps to the prover manually, as follows:

```
SArename Rkey
crypto enc rkB
crypto sign rkS
crypto sign rkA
success
```

The variable `Rkey` defines a table of public keys, and is assigned at three places, corresponding to principals A and B, and to other principals defined by the adversary. The instruction `SArename Rkey` allows us to distinguish these three cases. The instruction `crypto enc rkB` means that the prover should apply the definition of security of encryption (primitive `enc`), for the key generated from random number `rkB`. The instruction `success` means that prover should check whether the desired security properties are proved.

Needham-Schroeder public-key [38]: This protocol is an authentication protocol. Since our prover cannot check authentication yet, we transform it into a key exchange protocol in several ways, by choosing for the key either one of the nonces N_A and N_B shared between A and B, or $H(N_A, N_B)$ where H is a hash function (in the random oracle model). When the key is $H(N_A, N_B)$, one-session secrecy of the key cannot be proved for the original protocol, due to the well-known attack [33]. For the corrected version [33], our prover shows secrecy of the key. For both versions, the prover cannot prove one-session secrecy of N_A or N_B. For N_B, the failure of the proof corresponds to an attack: the adversary can check whether it is given N_B or a random number by sending $\{N'_B\}_{pk_B}$ to B as the last message of the protocol: B accepts if and only if $N'_B = N_B$. For N_A, the failure of the proof comes from limitations of our prover: The prover cannot take into account that N_A is accepted only after all messages that contain N_A have been sent, which prevents the previous attack. (This is the only case in our examples where the failure of the proof comes from limitations of the prover. This problem could probably be solved by improving the transformation **Simplify**.) Like for the Denning-Sacco protocol, we provided the main proof steps to the prover manually, as follows when the distributed key is N_A or N_B:

```
SArename Rkey
crypto sign rkS
crypto enc rkA
crypto enc rkB
success
```

When the distributed key is $H(N_A, N_B)$, the proof is as follows:

```
SArename Rkey
crypto sign rkS
crypto enc rkA
crypto enc rkB
crypto hash
SArename Na_39
simplify
success
```

The total runtime for all these tests is 60 s on a Pentium M 1.8 GHz, for version 1.01 of our prover CryptoVerif.

7 Conclusion

This paper presents a prover for cryptographic protocols sound in the computational model. This prover works with no or very little help from the user, can handle a wide variety of cryptographic primitives in a generic way, and produces proofs valid for a polynomial number of sessions in the presence of an active adversary. Thus, it represents important progress with respect to previous work in this area.

We have recently extended our prover to provide exact security proofs (that is, proofs with an explicit probability of an attack, instead of the asymptotic result that this probability is negligible) and to prove correspondence assertions. We leave these extensions for a future paper. In the future, it would also be interesting to handle even more cryptographic primitives, such as Diffie-Hellman key agreements. (In order to handle them, the language of equivalences that we use to specify the security properties of primitives will need to be extended.)

Acknowledgments I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him. I also thank Jacques Stern for initiating this work. This work was partly supported by ARA SSIA Formacrypt.

References

[1] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *TACS'01*, volume 2215 of *LNCS*, pages 82–94. Springer, Oct. 2001.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. In *PKC'05*, volume 3386 of *LNCS*, pages 65–84. Springer, Jan. 2005.

[5] P. Adão, G. Bana, J. Herzog, and A. Scedrov. Soundness of formal encryption in the presence of key-cycles. In *ESORICS'05*, volume 3679 of *LNCS*, pages 374–396. Springer, Sept. 2005.

[6] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *17th IEEE Computer Security Foundations Workshop*. IEEE, June 2004.

[7] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *26th IEEE Symposium on Security and Privacy*, pages 171–182. IEEE, May 2005.

[8] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *CCS'03*, pages 220–230. ACM, Oct. 2003.

[9] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *ESORICS 2003*, volume 2808 of *LNCS*, pages 271–290. Springer, Oct. 2003.

[10] G. Barthe, J. Cederquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In *IJCAR'04*, volume 3097 of *LNCS*, pages 385–399. Springer, July 2004.

[11] M. Baudet, V. Cortier, and S. Kremer. Computationally sound implementations of equational theories against passive adversaries. In *ICALP'05*, volume 3580 of *LNCS*, pages 652–663. Springer, July 2005.

[12] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *FOCS'97*, pages 394–403. IEEE, Oct. 1997. Full paper available at `http://www-cse.ucsd.edu/users/mihir/papers/sym-enc.html`.

[13] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology - CRYPTO '98*, volume 1462 of *LNCS*, pages 26–45. Springer, Aug. 1998.

[14] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, Dec. 2000.

[15] M. Bellare and P. Rogaway. The game-playing technique. Cryptology ePrint Archive, Report 2004/331, Dec. 2004. Available at `http://eprint.iacr.org/2004/331`.

[16] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, May 2004.

[17] B. Blanchet. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, Nov. 2005. Available at `http://eprint.iacr.org/2005/401`.

[18] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[19] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS'01*, pages 136–145. IEEE, Oct. 2001. An updated version is available at Cryptology ePrint Archive, `http://eprint.iacr.org/2000/067`.

[20] R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). Cryptology ePrint Archive, Report 2004/334, 2004. Available at `http://eprint.iacr.org/2004/334`.

[21] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP'05*, volume 3444 of *LNCS*, pages 157–171. Springer, Apr. 2005.

[22] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *ICALP'05*, volume 3580 of *LNCS*, pages 16–29. Springer, July 2005.

[23] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.

[24] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, June 2005. Available at `http://eprint.iacr.org/2005/181`.

[25] J. Herzog. A computational interpretation of Dolev-Yao adversaries. In *WITS'03*, pages 146–155, Apr. 2003.

[26] R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In *ESOP'05*, volume 3444 of *LNCS*, pages 172–185. Springer, Apr. 2005.

[27] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[28] P. Laud. Handling encryption in an analysis for secure information flow. In *ESOP'03*, volume 2618 of *LNCS*, pages 159–173. Springer, Apr. 2003.

[29] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy*, pages 71–85, May 2004.

[30] P. Laud. Secrecy types for a simulatable cryptographic library. In *CCS'05*, pages 26–35. ACM, Nov. 2005.

[31] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS'98*, pages 112–121, Nov. 1998.

[32] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *FM'99*, volume 1708 of *LNCS*, pages 776–793. Springer, Sept. 1999.

[33] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[34] P. Mateus, J. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In *CONCUR'03*, volume 2761 of *LNCS*, pages 327–349. Springer, Sept. 2003.

[35] D. Micciancio and B. Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. *Journal of Computer Security*, 12(1):99–129, 2004.

[36] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *TCC'04*, volume 2951 of *LNCS*, pages 133–151. Springer, Feb. 2004.

[37] J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 2006. To appear.

[38] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[39] R. M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[40] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[41] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *FOSSACS'04*, volume 2987 of *LNCS*, pages 468–483. Springer, Mar. 2004.

[42] V. Shoup. A proposal for an ISO standard for public-key encryption, Dec. 2001. ISO/IEC JTC 1/SC27.

[43] V. Shoup. OAEP reconsidered. *Journal of Cryptology*, 15(4):223–249, Sept. 2002.

[44] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. Unpublished manuscript, Feb. 2006.

[45] S. Tarento. Machine-checked security proofs of cryptographic signature schemes. In *ESORICS 2005*, volume 3679 of *LNCS*, pages 140–158. Springer, Sept. 2005.

[46] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 178–194, May 1993.