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 proper ties 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

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 that calls the functions of L (more precisely, contains as subterms terms that perform the same computations as functions of L) into a process $'$ 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 model, 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 of 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

$\ldots \times \eta(\ m)$ to $\eta(\ )$, such that $\eta(f)(x_1, \ldots, x_m)$ 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: $=^{\blacktriangle\blacktriangledown}$ for the equality test, $\neq^{\blacktriangle\blacktriangledown}$ for the inequality test (both taking two values of the same type and returning a value of type *bool*), $\vee^{\blacktriangle\blacktriangledown}$ for the boolean or, $\wedge^{\blacktriangle\blacktriangledown}$ for the boolean and, $\neg$ 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 $x[\ 1, \ldots, \ m]$ returns the content of the cell of indexes $1, \ldots, m$ of the array variable x. We use x, y, z, a as variable names. The function application $f(\ 1, \ldots, \ m)$ returns the result of applying function f to $1, \ldots, m$.

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

The output process $new\ x[i_1, \ldots, i_m] : \ ;$ chooses a new random number uniformly in $\eta(\ )$, stores it in $x[i_1, \ldots, i_m]$, and executes . (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] : \ $. Deterministic functions make automatic syntactic manipulations easier: we can duplicate a term without changing its value. The process $let\ x[i_1, \ldots, i_m] : \ = \ in$ stores the bitstring value of (which must be in $\eta(\ )$) in $x[i_1, \ldots, i_m]$, and executes . The process $if \ \ then \ \ else \ '$ executes if evaluates to 1 and $'$ if evaluates to 0. Next, we explain the process $find\ (\bigoplus_{j=1}^{m} a_{j1}[\tilde{i}\,] \leq n_{j1}, \ldots, a_{jm_j}[\tilde{i}\,] \leq n_{jm_j}\ suchthat\ defined(\ j1, \ldots, \ jl_j) \wedge \ j\ then\ \ j)\ else\ $, where $\tilde{i}$ denotes a tuple $i_1, \ldots, i_{m'}$. The order and array indexes on tuples are taken component-wise, so for instance, $a_{j1}[\tilde{i}\,] \leq n_{j1}, \ldots, a_{jm_j}[\tilde{i}\,] \leq n_{jm_j}$ can be further abbreviated $\widetilde{a_j}[\tilde{i}\,] \leq \widetilde{n_j}$. A simple example is the following: $find\ a \leq n\ suchthat\ defined(x[a]) \wedge x[a] = a\ then\ \ '\ else\ $ tries to find an index a such that $x[a]$ is defined and $x[a] = a$, and when such a a is found, it executes $'$ with that value of a; otherwise, it executes . In other words, this *find* construct looks for the value a in the array x, and when a is found, it stores in a an index such that $x[a] = a$. Therefore, the *find* construct allows us to access arrays, which is key for our purpose. More generally, $find\ a_1[\tilde{i}\,] \leq n_1, \ldots, a_m[\tilde{i}\,] \leq n_m\ suchthat\ defined(\ 1, \ldots, \ l) \wedge \ \ then\ \ '\ else\ $ tries to find values of $a_1, \ldots, a_m$ for which $1, \ldots, l$ are defined and is true. In case of success, it executes $'$. In case of failure, it executes . This is further generalized to m branches: $find\ (\bigoplus_{j=1}^{m} a_{j1}[\tilde{i}\,] \leq n_{j1}, \ldots, a_{jm_j}[\tilde{i}\,] \leq n_{jm_j}\ suchthat\ defined(\ j1, \ldots, \ jl_j) \wedge \ j\ then\ \ j)\ else\ $ tries to find a branch j in $[1, m]$ such that there are values of $a_{j1}, \ldots, a_{jm_j}$ for which $j1, \ldots, jl_j$ are defined and j is true. In case of success, it executes j. In case of failure for all branches, it executes . More formally, it evaluates the conditions $defined(\ j1, \ldots, \ jl_j) \wedge \ j$ for each j and each value of $a_{j1}[\tilde{i}\,], \ldots, a_{jm_j}[\tilde{i}\,]$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is 1, it executes . Otherwise, it chooses randomly with uniform[1] probability one j and one value of $a_{j1}[\tilde{i}\,], \ldots, a_{jm_j}[\tilde{i}\,]$ such that the corresponding condition is 1, and executes j.

Finally, let us explain the output $\overline{c[\ 1, \ldots, \ l]}\langle \blacktriangle\blacktriangledown_1, \ldots, \blacktriangle\blacktriangledown_k \rangle; \ $. A channel $c[\ 1, \ldots, \ l]$ consists of both a channel name c and a tuple of terms $1, \ldots, l$. 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 $1, \ldots, l$ 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[\ 1, \ldots, \ l]}\langle \blacktriangle\blacktriangledown_1, \ldots, \blacktriangle\blacktriangledown_k \rangle; \ $, 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[\ '_1, \ldots, \ '_l](x_1[\tilde{i}\,] : \ 1, \ldots, x_k[\tilde{i}\,] : \ k); \ $ is chosen randomly with uniform probability. The communication is then executed: for each $j \leq$, the output message $\blacktriangle\blacktriangledown_j$ is evaluated, its result is truncated to length $\mathrm{maxlen}_\eta(c)$, the obtained bitstring is stored in $x_j[\tilde{i}\,]$ if it is in $\eta(\ j)$ (otherwise the process blocks). Finally, the output process that follows the input is executed. The input process 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 $!^{i \leq n} c[i](x[i] : \ ) \ldots \overline{c'[i]}\langle \ \rangle \ldots$ 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* $\ $ *then* $\ $ as an abbreviation for *if* $\ $ *then* $\ $ *else* $\overline{\text{yield}}\langle\rangle; 0$, 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] = \ $ *in* $\ldots$ More formally, we require the following invariant:

Invariant 1 (Single definition) The process $\ _0$ satisfies Invariant 1 if and only if

1. in a definition of $x[i_1, \ldots, i_m]$ in $\ _0$, 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 $\ _0$ 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 $\ _0$ satisfies Invariant 2 if and only if every occurrence of a variable access $x[\ _1, \ldots, \ _m]$ in $\ _0$ is either

- syntactically under the definition of $x[\ _1, \ldots, \ _m]$ (in which case $\ _1, \ldots, \ _m$ are in fact the current replication indexes at the definition of x);
- or in a *defined* condition in a *find* process;
- or in $\ _j'$ or $\ _j$ in a process of the form *find* $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[i] \leq \widetilde{n_j}$ *suchthat defined*$(\ _{j1}', \ldots, \ _{jl_j}') \wedge \ _j'$ *then* $\ _j)$ *else* where for some $\ \leq l_j$, $x[\ _1, \ldots, \ _m]$ is a subterm of $\ _{jk}'$.

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 : \ _1 \times \ldots \times \ _m \to \ $ is *poly injective* when it is injective and its inverses can be computed in polynomial time, that is, there exist functions $f_j^{-1} : \ \to \ _j$ $(1 \leq j \leq m)$ 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) = \ $ *in* $\ $ *else* $\ $ as an abbreviation for *let* $y : \ = \ $ *in* *let* $x_1 : \ _1 = f_1^{-1}(y)$ *in* $\ldots$ *let* $x_m : \ _m = f_m^{-1}(y)$ *in* *if* $f(x_1, \ldots, x_m) = y$ *then* $\ $ *else* $\ $. We naturally generalize this construct to *let* $\blacktriangle = \ $ *in* $\ $ *else* $\ $ where $\blacktriangle$ is built from poly-injective functions and variables.

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

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

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

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

A mac is secure against existential forgery under chosen

message attack if and only if for all polynomials q,

$$\max_{\mathcal{A}} \Pr[r \xleftarrow{R} \ _\eta(\ _{mr}); \quad \leftarrow mkgen_\eta(r); \\ (m, \ ) \leftarrow \mathcal{A}^{mac_\eta(.,k), check_\eta(.,k,.)} : check_\eta(m, \ , \ )]$$

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

Definition 2 Let $\ _r$ and $\ _r'$ be fixed-length types; let $\ _k$ and $\ _e$ be types. A symmetric encryption scheme [12] (stream cipher) consists of three function symbols $kgen : \ _r \to \ _k$, $enc : bitstring \times \ _k \times \ _r' \to \ _e$, and $dec : \ _e \times \ _k \to bitstring_\perp$, with $\ _\eta(kgen) = kgen_\eta$, $\ _\eta(enc) = enc_\eta$, $\ _\eta(dec) = dec_\eta$, such that for all $m \in Bitstring$, $r \in \ _\eta(\ _r)$, and $r' \in \ _\eta(\ _r')$, $dec_\eta(enc_\eta(m, kgen_\eta(r), r'), kgen_\eta(r)) = m$.

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,

$$\max_{\mathcal{A}} 2 \Pr[b \xleftarrow{R} \{0,1\}; r \xleftarrow{R} \ _\eta(\ _r); \quad \leftarrow kgen_\eta(r); \\ b' \leftarrow \mathcal{A}^{r' \xleftarrow{R} I_\eta(T_r'); enc_\eta(LR(.,.,b),k,r')} : b' = b] - 1$$

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 $r' \xleftarrow{R} \ _\eta(\ _r'); enc_\eta(LR(a_0, a_1, b), \ , r')$, that is, encrypts a_0 when $b = 0$ and a_1 when $b = 1$.

Example 1 Let us consider the following trivial protocol:

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

A and $\ $ 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 $\ $. 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 $\ $. This protocol can be modeled in our calculus by the following process $\ _0$:

$$\ _0 = s \ ar \ (); new \ x_r : \ _r; let \ x_k : \ _k = kgen(x_r) \ in \\ new \ x'_r : \ _{mr}; let \ x_{mk} : \ _{mk} = mkgen(x'_r) \ in \\ \overline{c}\langle\rangle; (\ _A \mid \ _B)$$

$$\ _A = !^{i \le n} c_A[i](); new \ x'_k : \ _k; new \ x''_r : \ _r'; \\ let \ x_m : bitstring = enc(\ _{2b}(x'_k), x_k, x''_r) \ in \\ \overline{c_A[i]}\langle x_m, mac(x_m, x_{mk})\rangle$$

$$\ _B = !^{i' \le n} c_B[i'](x'_m, x_{ma}); if \ check(x'_m, x_{mk}, x_{ma}) \\ then \ let \ i_\perp(\ _{2b}(x''_k)) = dec(x'_m, x_k) \ in \ \overline{c_B[i']}\langle\rangle$$

When $\ _0$ receives a message on channel $s \ ar$, 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 $\ $ 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 $\ _A$ creates a fresh key x'_k ($\ _k$ 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 $\ _{2b} : \ _k \to bitstring$ is the natural injection $\ _\eta(\ _{2b})(x) = x$; it is needed only for type conversion. The context is then expected to forward this message on $c_B[i]$. When $\ _B$ receives this message, it checks the mac, decrypts, and stores the obtained key in x''_k. (The function $i_\perp : bitstring \to bitstring_\perp$ 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 $\ $. 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 $\mathrm{var}(\ )$ the set of variables that occur in $\ $, and by $\mathrm{fc}(\ )$ the set of free channels of $\ $. (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] \to \ $, where the definition of $x[i_1, \ldots, i_m]$ of type $\ $ occurs under replications $!^{i_1 \le n_1}, \ldots, !^{i_m \le n_m}$. 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 \ $ which means that the process $\ $ is well-typed in environment $\mathcal{E}$.

Invariant 3 (Typing) The process $_0$ satisfies Invariant 3 if and only the type environment $\mathcal{E}$ for $_0$ is well-defined, and $\mathcal{E} \vdash \ _0$.

We require the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : \ \rightarrow \ '$ 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 $, \ \Downarrow a$ means that the term evaluates to the bitstring a in environment . We denote by $\Pr[\ \rightsquigarrow_\eta \overline{c}\langle a\rangle]$ the probability that outputs the bitstring a on channel c after some reductions.

Our semantics is such that, for each process , there exists a probabilistic polynomial time Turing machine that simulates . (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 $[\,]$, $newChannel\ c; C$, $\mid C$, and $C \mid$. We use an evaluation context to represent the adversary. We denote by $C[\ ]$ the process obtained by replacing the hole $[\,]$ in the context C with the process .

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

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

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

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

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 and $'$: the context C is allowed to access variables of and $'$ only when they are in V. (In more standard settings, the calculus does not have constructs that allow the context to access variables of and $'$.) The following result is not difficult to prove:

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

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

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 $_0$ into a process $'_0$ such that $_0 \approx^V \ '_0$.

3.1 Syntactic Transformations

RemoveAssign(x): When x is defined by an assignment $let\ x[i_1, \ldots, i_l] : \ = \ in$, we replace x with its value. Precisely, the transformation is performed only when x does not occur in (non-cyclic assignment). When x has several definitions, we simply replace $x[i_1, \ldots, i_l]$ with in . (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 $x[\ _1, \ldots, \ _l]$ with $\{ \ _1/i_1, \ldots, \ _l/i_l\}$. We additionally update the *defined* conditions of *find* to preserve Invariant 2, and to maintain the requirement that $x[\ _1, \ldots, \ _l]$ 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 $let\ x_{mk} : \ _{mk} = mkgen(x'_r)$. 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 $= \tilde{u}[\tilde{i}] \leq \tilde{n}\ suchthat\ defined(\ _1', \ldots, \ _{l'}') \wedge$ $then$ where $x[\ _1, \ldots, \ _l]$ is a subterm of some $\ _k'$ for $\leq l'$ is replaced with m branches $\{x_j[\ _1, \ldots, \ _l]/x[\ _1, \ldots, \ _l]\}$ 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 $\forall x_1 : \ _1, \ldots, \forall x_m : \ _m,$ which mean that for all environments , if for all $j \leq m$, $(x_j) \in \ _\eta(\ _j)$, then , $\Downarrow 1$. For example, considering mac and stream ciphers as in Definitions 1 and 2 respectively, we have:

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

$$\forall m : bitstring; \forall r : \ _r, \forall r' : \ _r', \quad dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m) \qquad (enc)$$

  We express the poly-injectivity of the function $\ 2b$ of Example 1 by

$$\forall x : \ _k, \forall y : \ _k, (\ 2b(x) = \ 2b(y)) = (x = y)$$
$$\forall x : \ _k, \ 2b^{-1}(\ 2b(x)) = x \qquad (\ 2b)$$

  where $\ 2b^{-1}$ is a function symbol that denotes the inverse of $\ 2b$. We have similar formulas for $i_\perp$.

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

- The low probability of collision between random values. For example, when x is defined by $new\ x : \ $ and is a large type, $x[\ _1, \ldots, \ _m] = x[\ _1', \ldots, \ _m']$ implies $\ _1 = \ _1', \ldots, \ _m = \ _m'$ up to negligible probability.

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

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

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
$!^{i \leq n} new\ y_1 : \ _1; \ldots; new\ y_l : \ _l; (G_1, \ldots, G_m)$	replication, restrictions
$(x_1 : \ _1, \ldots, x_l : \ _l) \rightarrow$	function
$::=$	functional processes
	term
$new\ x[\tilde{i}] : \ ;$	random number
$let\ x[\tilde{i}] : \ = \ in$	assignment
$if \quad then \ _1 else \ _2$	test
$find\ (\bigoplus_{j=1}^{m} \tilde{u}_j[\tilde{i}] \leq \tilde{n}_j\ suchthat$ $defined(\ _{j1}, \ldots, \ _{jl_j}) \wedge \ _j\ then \ _j)\ else$	array lookup

Intuitively, $(x_1 : \ _1, \ldots, x_l : \ _l) \rightarrow$ represents a function that takes as argument values $x_1, \ldots, x_l$ of types $\ _1, \ldots, \ _l$ respectively, and returns a result computed by . 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

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

$[\![!^{i\ \ n} new$

$\overline{c_{\widetilde{j}}}$

$\overline{c_{\widetilde{j}}[\widetilde{\ }, \ ]}\langle\rangle; ([\![G_1$

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

$[\![let\ x[\widetilde{\ }] :$

stream cipher (Definition 2) by the equivalence:

$$\begin{aligned} !^{i' \leq n'}\, new\ r : \ & {}_r; !^{i \leq n}(x : bitstring) \rightarrow \\ & new\ r' : \ {}'_r; enc(x, kgen(r), r') \\ \approx !^{i' \leq n'}\, new\ r : \ & {}_r; !^{i \leq n}(x : bitstring) \rightarrow \\ & new\ r' : \ {}'_r; enc'(\ (x), kgen'(r), r') \end{aligned} \quad (enc_{\text{eq}})$$

where enc' and $kgen'$ are function symbols with the same types as enc and $kgen$ respectively, and $: bitstring \rightarrow bitstring$ is the function that returns a bitstring of the same length as its argument, consisting only of zeroes. Using equations such as $\forall x : \ , \ (T2b(x)) = \ _T$, we can prove that $(T2b(x))$ does not depend on x when x is of a fixed-length type and $T2b : \ \rightarrow bitstring$ 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 $_0$ observationally equivalent to $C[[\![L]\!]]$ into a process $'_0$ observationally equivalent to $C[[\![R]\!]]$, for some evaluation context C. In order to check that $_0 \approx C[[\![L]\!]]$, the prover uses sufficient conditions, which essentially guarantee that all uses of certain secret variables of $_0$, 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 $\in \mathcal{M}$, there exist a term $_M$, result of a function of L, and a substitution σ_M such that $= \sigma_M{}_M$. (Precisely, σ_M applies to the abbreviated form of $_M$ in which we write x instead of $x[\tilde{i}]$.) Let $\tilde{i}$ and $\tilde{i}'$ be the sequences of current replication indexes at $_M$ in L and at in $_0$, respectively. There exists a function $mapIdx_M$ that maps the array indexes at in $_0$ to the array indexes at $_M$ in L: the evaluation of when $\tilde{i}' = \tilde{a}$ will correspond in $C[[\![L]\!]]$ to the evaluation of $_M$ when $\tilde{i} = mapIdx_M(\tilde{a})$. Thus, σ_M and $mapIdx_M$ induce a correspondence between $_0$ and L: for all $\in \mathcal{M}$, for all $x[\tilde{i}']$ that occur in $_M$, $(\sigma_M x)\{\tilde{a}/\tilde{i}'\}$ corresponds to $x[\tilde{i}']\{mapIdx_M(\tilde{a})/\tilde{i}\}$, that is, $(\sigma_M x)\{\tilde{a}/\tilde{i}'\}$ in a trace of $_0$ has the same value as $x[\tilde{i}']\{mapIdx_M(\tilde{a})/\tilde{i}\}$ in the corresponding trace of $C[[\![L]\!]]$ ($\tilde{i}'$ is a prefix of $\tilde{i}$).

For example, consider a process $_0$ that contains $_1 = enc(\ {}'_1, kgen(x_r), x'_r[i_1])$ and $_2 = enc(\ {}'_2, kgen(x_r), x''_r[i_2])$ with $i_1 \leq n_1$, $i_2 \leq n_2$, and x_r, x'_r, x''_r bound by restrictions. Let $S = \{x_r, x'_r, x''_r\}$, $\mathcal{M} = \{ \ _1, \ _2\}$, and $_{M_1} = \ _{M_2} = enc(x[i', j], kgen(r[i']), r'[i', j])$. The functions $mapIdx_{M_1}$ and $mapIdx_{M_2}$ are defined by $mapIdx_{M_1}(a_1) = (1, a_1)$ for $a_1 \in [1, \ _\eta(n_1)]$ and $mapIdx_{M_2}(a_2) = (1, a_2 + \ _\eta(n_1))$ for $a_2 \in [1, \ _\eta(n_2)]$. Then $\ {}'_1\{a_1/i_1\}$ corresponds to $x[1, a_1]$, x_r to $r[1]$, $x'_r[a_1]$ to $r'[1, a_1]$, $\ {}'_2\{a_2/i_2\}$ to $x[1, a_2 + \ _\eta(n_1)]$, and $x''_r[a_2]$ to $r'[1, a_2 + \ _\eta(n_1)]$. The $_2$ are $_M$

lences $L \approx R$ for mac (mac_{eq}) and encryption (enc_{eq}); and the process P_0 of Example 1.

The prover first applies **RemoveAssign**(x_{mk}) to the process P_0 of Example 1, as described in Example 2. The process can then be transformed using the security of the mac. We take $S = \{x'_r\}$, $P_1 = mac(x_m[i], mkgen(x'_r))$, $P_2 = check(x'_m[i'], mkgen(x'_r), x_{ma}[i'])$, and $\mathcal{M} = \{P_1, P_2\}$. We have $\blacktriangleright_{M_1} = mac(x[i'', i], mkgen(r[i'']))$, $\blacktriangleright_{M_2} = check(m[i'', i'], mkgen(r[i'']), ma[i'', i'])$, $mapIdx_{M_1}(a_1) = (1, a_1)$, 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 P'_0:

$$
\begin{array}{l}
P'_0 = s\ ar\ (); new\ x_r : T_r; let\ x_k : T_k = kgen(x_r)\ in \\
\quad new\ x'_r : T_{mr}; \overline{c}\langle\rangle; (P'_A \mid P'_B) \\
P'_A = !^{i \leq n} c_A[i](); new\ x'_k : T_k; new\ x''_r : T_r; \\
\quad let\ x_m : bitstring = enc(P_{2b}(x'_k), x_k, x''_r)\ in \\
\quad \overline{c_A[i]}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle \\
P'_B = !^{i' \leq n} c_B[i'](x'_m, x_{ma}); \\
\quad find\ u \leq n\ suchthat\ defined(x_m[u]) \wedge x'_m = x_m[u] \wedge \\
\qquad check'(x'_m, mkgen'(x'_r), x_{ma})\ then \\
\qquad (if\ 1\ then\ let\ i_\perp(P_{2b}(x''_k)) = dec(x'_m, x_k)\ in\ \overline{c_B[i']}\langle\rangle) \\
\quad else \\
\qquad (if\ 0\ then\ let\ i_\perp(P_{2b}(x''_k)) = dec(x'_m, x_k)\ in\ \overline{c_B[i']}\langle\rangle)
\end{array}
$$

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 $find\ u \leq n\ suchthat\ defined(x_m[u]) \wedge x'_m = x_m[u] \wedge check'(x'_m, mkgen'(x'_r), x_{ma})\ then\ 1\ else\ 0$ which yields P'_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**, P'_A is unchanged and P'_B becomes

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

First, the tests *if* 1 *then* ... and *if* 0 *then* ... are simplified. The term $dec(x'_m, x_k)$ is simplified knowing $x'_m = x_m[u]$ by the *find* condition, $x_m[u] = enc(P_{2b}(x'_k[u]), x_k, x''_r[u])$ 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 $dec(x'_m, x_k) = i_\perp(P_{2b}(x'_k[u]))$. By injectivity of $i_\perp$ and P_{2b}, 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 $\forall x : T_k$, $P_{2b}^{-1}(P_{2b}(x)) = x$.

After applying **RemoveAssign**(x_k), one can apply the security of encryption: $enc(P_{2b}(x'_k), kgen(x_r), x''_r)$ becomes $enc'(P_{2b}(x'_k)), kgen(x_r), x''_r)$. After **Simplify**, it becomes $enc'(T_k, kgen(x_r), x''_r)$, using $\forall x : T_k$, $P_{2b}(x)) = T_k$ (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 $mac(x_i, P_1)$ with the same key in the initial process, each $check(m_j, P_1, ma_j)$ 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 P *preserves the one session secrecy of* x when $P \mid C_x \approx P \mid C'_x$, where

$$
\begin{array}{l}
C_x = c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \\
\quad if\ defined(x[u_1, \ldots, u_m])\ then\ \overline{c}\langle x[u_1, \ldots, u_m]\rangle \\
C'_x = c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \\
\quad if\ defined(x[u_1, \ldots, u_m])\ then\ new\ y : T; \overline{c}\langle y\rangle
\end{array}
$$

$c \notin \text{fc}(P)$, $u_1, \ldots, u_m \notin \text{var}(P)$, 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 C_x and C'_x.

Proposition 4 (One-session secrecy) *Consider a process such that there exists a set of variables S such that 1) the de nitions of x are either restrictions new $x[\widetilde{i}]$: and $x \in S$ or assignments let $x[\widetilde{i}]$: $= z[\ _1, \ldots, \ _l]$ where z is de ned by restrictions new $z[i'_1, \ldots, i'_q]$: and $z \in S$ and 2) all accesses to variables $y \in S$ in are of the form "let $y'[\widetilde{i}]$: $' = y[\ _1, \ldots, \ _l]$ with $y' \in S$ Then $| \ _x \approx_0 \ | \ '_x$ hence 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* $y'[\widetilde{i}]$: $' = y[\ _1, \ldots, \ _l]$" when $y \in S$.

Definition 5 (Secrecy) The process *preserves the se crecy of* x when $| R_x \approx \ | R'_x$, 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*$(y[u'], u_1[u'], \ldots, u_m[u'])$
$\wedge\ u_1[u'] = u_1 \wedge \ldots \wedge u_m[u'] = u_m$
then $\overline{c}\langle y[u']\rangle$ *else new* y : $; \overline{c}\langle y\rangle$

$c \notin \mathrm{fc}(\ ), u_1, \ldots, u_m, u' \notin \mathrm{var}(\ ), \mathcal{E}(x) = [1, n_1] \times \ldots \times [1, n_m] \to \ $, and $\ _\eta(n) \geq \ _\eta(n_1) \times \ldots \times \ _\eta(n_m)$.

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 satis es the hypothesis of Proposition 4*

When $\mathcal{T}$ is a trace of $C[\ ]$ for some evaluation context C we de ne $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}])$ *the de ning restriction of $x[\widetilde{a}]$ in trace $\mathcal{T}$ as follows if $x[\widetilde{a}]$ is de ned by new $x[\widetilde{a}]$: in $\mathcal{T}$* $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = x[\widetilde{a}]$ *if $x[\widetilde{a}]$ is de ned by let $x[\widetilde{a}]$: $= z[\ _1, \ldots, \ _l]$* $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = z[a'_1, \ldots, a'_l]$ *where $\ , \ _k \Downarrow a'_k$ for all $\leq l$ and is the environment in $\mathcal{T}$ at the de nition of $x[\widetilde{a}]$*

Assume that for all evaluation contexts C accept able for 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 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 $e(\widetilde{\ })$ has been executed then some events $e_i(\widetilde{\ }_i)$ 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 $\approx^{\{x\}}$ $'$ and preserves the one session secrecy of x then $'$ 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 $_0$ preserves the (one-session) secrecy of x, we transform $_0$ by the transformations described in Section 3 with $V = \{x\}$. By Propositions 1 and 3, we obtain a process $'_0$ such that $_0 \approx^V \ '_0$. We use Propositions 4 or 5 to show that $'_0$ preserves the (one-session) secrecy of x, and finally conclude that $_0$ 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 : $_k = x'_k[a]$ 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 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 of L, but we find in $_0$ only part of , the other parts being variable accesses $x[\ldots]$ while we expect function applications. In this case, we advise **RemoveAssign**(x). For example, if $_0$ contains $enc(\ ', x_k, x'_r)$ and we look for $enc(x_m, kgen(x_r), x_{r'})$, we advise **RemoveAssign**(x_k). If $_0$ 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 $_0$, we advise **SArename**(x).

- When we check whether x is secret or one-session secret, we have an assignment $let \ x[\tilde{i}] : \ = y[\widetilde{\ }] \ in$, 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 $\{ \ _B\}_K$ where is the exchanged key. This message may reveal some information on . After removing this confirmation round, our prover shows the one-session secrecy of . However, it cannot show the secrecy of , 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 , 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,

[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 cryp-