Mechanizing Game-Based Proofs of Security Protocols

Bruno BLANCHET [1]
INRIA, École Normale Supérieure, CNRS, Paris, France

Abstract. After a short introduction to the field of security protocol verification, we present the automatic protocol verifier CryptoVerif. In contrast to most previous protocol verifiers, CryptoVerif does not rely on the Dolev-Yao model, but on the computational model. It produces proofs presented as sequences of games, like those manually done by cryptographers; these games are formalized in a probabilistic process calculus. CryptoVerif provides a generic method for specifying security properties of the cryptographic primitives. It can prove secrecy and correspondence properties (including authentication). It produces proofs valid for any number of sessions, in the presence of an active adversary. It also provides an explicit formula for the probability of success of an attack against the protocol, as a function of the probability of breaking each primitive and of the number of sessions.

Keywords. Security protocols; computational model; automatic proof; sequences of games; process calculi.

Introduction

A security protocol is a program that guarantees security properties, such as the secrecy of some piece of data, by relying on cryptographic primitives, such as encryption or signatures. Security protocols make it possible to securely exchange data on insecure networks such as Internet. The design of security protocols is well-known to be error-prone. This can be illustrated by the attack against the Needham-Schroeder public-key protocol [49] found by Lowe [46] 17 years after its publication. Errors in security protocols can have serious consequences, such as loss of money in e-commerce. Furthermore, security errors cannot be detected by testing, since they appear only in the presence of a malicious adversary. Therefore, one aims at proving that security protocols are correct. Manual proofs are complex and error-prone, so formal methods can play an important role by providing tools for proving security protocols correct or for finding attacks.

There exist two main models for analyzing security protocols:

- In the symbolic model, often called *Dolev-Yao* model [37], cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. Messages are terms on these primitives and the adversary can compute only using these primitives.

[1] Corresponding Author: Bruno Blanchet, École Normale Supérieure, DI, 45 rue d'Ulm, 75005 Paris, France; E-mail: blanchet@di.ens.fr

- In contrast, in the *computational* model, messages are bitstrings, cryptographic primitives are functions from bitstrings to bitstrings, and the adversary is any probabilistic Turing machine.

The computational model is close to the real execution of protocols, but the proofs are usually manual and informal. The Dolev-Yao model is an abstract model that makes it easier to build automatic verification tools, and many such tools exist: AVISPA [5], FDR [46], and ProVerif [20], for instance. Hubert Comon-Lundh's course will deal with the verification of security protocols in this model. However, security proofs in the Dolev-Yao model in general do not imply security in the computational model.

In order to mechanize proofs in the computational model, several approaches have been considered.

- In the indirect approach, following the seminal paper by Abadi and Rogaway [1], one shows the soundness of the Dolev-Yao model with respect to the computational model, that is, one proves that the security of a protocol in the Dolev-Yao model implies its security in the computational model, modulo additional assumptions. Combining such a result with a Dolev-Yao automatic verifier, one obtains automatic proofs of protocols in the computational model. This approach received much interest [6, 8, 29, 31, 39, 47] and a tool [30] was developed based on [31] to obtain computational proofs using the Dolev-Yao verifier AVISPA, for protocols that rely on public-key encryption and signatures. However, this approach has limitations: since the computational and Dolev-Yao models do not correspond exactly, soundness requires additional hypotheses. (For example, key cycles have to be excluded, or a specific security definition of encryption is needed [3].)
  In a related approach, Backes, Pfitzmann, and Waidner [9–11] 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. This framework has been used for a computationally-sound machine-checked proof of the Needham-Schroeder-Lowe protocol [54].
  Canetti [27] introduced the notion of universal composability. With Herzog [28], 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 [21] for verifying protocols in this framework.
- Techniques used previously in the Dolev-Yao model have also been adapted in order to obtain proofs in the computational model.
  For instance, Datta, Derek, Mitchell, Shmatikov, and Turuani [35, 36] have adapted the logic PCL (Protocol Composition Logic), first designed for proving protocols in the Dolev-Yao model, to the computational model. Other computationally sound logics include CIL (Computational Indistinguishability Logic) [12] and a specialized Hoare logic designed for proving asymmetric encryption schemes in the random oracle model [32, 33].
  Similarly, type systems [34, 43, 45, 53] can provide computational security guarantees. For instance, [43] handles shared-key and public-key encryption, with an unbounded number of sessions. This system relies on the Backes-Pfitzmann-Waidner library. A type inference algorithm is given in [7].

- In the direct approach, one aims at mechanizing proofs in the computational model, without using a Dolev-Yao protocol verifier. Computational proofs made by cryptographers are typically presented as sequences of games [18, 52]: the initial game represents the protocol to prove; the goal is to show that the probability of breaking a certain security property 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. Halevi [38] suggested to use tools for mechanizing these proofs, and several techniques have been used for reaching this goal.
  CryptoVerif [22–25], which will be the main topic of this course, is such a tool. It generates proofs by sequences of games automatically or with little user interaction. The games are formalized in a probabilistic process calculus. CryptoVerif provides a generic method for specifying security properties of many cryptographic primitives. It proves secrecy and authentication properties. It also provides a bound on the probability of success of an attack. It considerably extends early works by Laud [41, 42] which were limited either to passive adversaries or to a single session of the protocol. More recently, Tšahhirov and Laud [44, 55] developed a tool similar to CryptoVerif but that represents games by dependency graphs; it handles only public-key and shared-key encryption and proves secrecy properties.
  The tool CertiCrypt [13, 15, 16, 26] enables the machine-checked construction and verification of cryptographic proofs by sequences of games. It relies on the general-purpose proof assistant Coq, which is widely believed to be correct. EasyCrypt [14] generates CertiCrypt proofs from proof sketches that formally represent the sequence of games and hints, which makes the tool easier to use. Nowak *et al.* [4, 50, 51] follow a similar idea by providing Coq proofs for several basic cryptographic primitives.

In the tool CryptoVerif, games are represented in a process calculus inspired by the pi-calculus and by the calculi of [48] and of [43]. In this calculus, messages are bitstrings, and cryptographic primitives are functions from bitstrings to bitstrings. The calculus has a probabilistic semantics. The main tool for specifying security assumptions is observational equivalence: Q is observationally equivalent to Q' up to probability p, $Q \approx_p Q'$, when the adversary has probability at most p 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 security 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 standard security assumption on a message authentication code (MAC). Informally, this assumption 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 (*i.e.*, function). So, in cryptographic proofs, one defines a list containing the arguments of calls to the MAC oracle, and when verifying 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. Furthermore, relations between elements of arrays can easily be expressed by equalities, possibly involving computations on array indices.

CryptoVerif 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 exploits the security assumptions on cryptographic primitives in order to obtain a simpler game. As described in Section 2.2, these transformations can be specified in a generic way: we represent the security assumption of each cryptographic primitive by an observational equivalence $L \approx_p R$, where the processes L and R encode oracles: they input the arguments of the oracle and send its result back. Then, the prover can automatically transform a process Q that calls the oracles of L (more precisely, contains as subterms terms that perform the same computations as oracles of L) into a process Q' that calls the oracles of R instead. We have used this technique to specify several variants of shared-key and public-key encryption, signature, message authentication codes, hash functions, Diffie-Hellman key agreement, simply by giving the appropriate equivalence $L \approx_p R$ to the prover. Other game transformations are syntactic transformations, used in order to be able to apply an assumption on a cryptographic primitive, or to simplify the game obtained after applying such an assumption.

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, CryptoVerif 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 assumptions 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 assumptions on primitives can be applied, but only one leads to a proof of the protocol. Importantly, CryptoVerif 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 assumptions on the cryptographic primitives.

CryptoVerif has been implemented in Ocaml (29800 lines of code for version 1.12 of CryptoVerif) and is available at `http://www.cryptoverif.ens.fr/`.

Outline The next section presents the process calculus for representing games. Section 2 describes the game transformations that serve for proving protocols. Section 3 gives criteria for proving secrecy properties of protocols. Section 4 explains how the prover chooses which transformation to apply at each point. Section 5 presents applications of CryptoVerif and Section 6 concludes.

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 by $|S|$. 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.

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

Figure 1. Syntax of the process calculus

1. A Calculus for Games

1.1. Syntax and Informal Semantics

CryptoVerif represents games in the syntax of Figure 1. This calculus assumes a countable set of channel names, denoted by c. It uses parameters, denoted by n, which are integers that bound the number of executions of processes. It also uses types, denoted by T, which are subsets of $bitstring_\perp = bitstring \cup \{\perp\}$ where $bitstring$ is the set of all bitstrings and $\perp$ is a special symbol. Let *fixed-length* types be types that consist of the set of all bitstrings of a certain length. Particular types are predefined: $bool = \{\mathrm{true}, \mathrm{false}\}$, where false is 0 and true is 1; $bitstring$; $bitstring_\perp$; $[1, n]$ where n is a parameter. (We consider integers as bitstrings without leading zeroes.)

The calculus also uses function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \rightarrow T$, and represents an efficiently computable, deterministic function that maps each tuple in $T_1 \times \ldots \times T_m$ to an element of T. 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 indices are typically used as array indices.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indices $M_1, \ldots, M_m$ of the m-dimensional 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 $i \in [1, n]$; $\mathsf{newChannel}\ c; Q$ 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 $\mathsf{new}\ x[i_1, \ldots, i_m] : T; P$ chooses a new random number uniformly in T, stores it in $x[i_1, \ldots, i_m]$, and executes P. (The type T must be a fixed-length type, because probabilistic Turing machines can choose random numbers uniformly only in such types.) Function symbols represent deterministic functions, so all random numbers must be chosen by $\mathsf{new}\ x[i_1, \ldots, i_m] : T$. Deterministic functions make automatic syntactic manipulations easier: we can duplicate a term without changing its value. The process $\mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ P$ stores the bitstring value of M (which must be in T) in $x[i_1, \ldots, i_m]$ and executes P. The process $\mathsf{event}\ e(M_1, \ldots, M_l); P$ executes the event $e(M_1, \ldots, M_l)$, then runs P. This event records that a certain program point has been reached with certain values of $M_1, \ldots, M_l$, but otherwise does not affect the execution of the process. Next, we explain the process $\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$, where $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_{m'}$. The order and array indices 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: $\mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge x[u] = a\ \mathsf{then}\ P'\ \mathsf{else}\ P$ 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, $\mathsf{find}\ u_1[\widetilde{i}] \leq n_1, \ldots, u_m[\widetilde{i}] \leq n_m\ \mathsf{suchthat}\ \mathsf{defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P'\ \mathsf{else}\ P$ 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: $\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$ 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 $\mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ 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 true, it executes P. Otherwise, it chooses randomly with uniform[2] probability one j and one value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is true and executes P_j. The conditional $\mathsf{if}\ \mathsf{defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$ executes P if $M_1, \ldots, M_l$ are defined and M evaluates to true. Otherwise, it executes P'. This conditional is equivalent to $\mathsf{find}\ \mathsf{suchthat}\ \mathsf{defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$. The

[2] A probabilistic 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.

conjunct $\mathsf{defined}(M_1, \ldots, M_l)$ can be omitted when $l = 0$ and M can be omitted when it is true.

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 can be declared private by $\mathsf{newChannel}\ c$; the adversary can never have access to channel $c[M_1, \ldots, M_l]$ when c is private. (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. 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 channel $c[M'_l \ldots, M'_l]$, where $M'_1, \ldots, M'_l$ evaluate to the same bitstrings as $M_1, \ldots, M_l$, and with the same arity k, 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 and stored in $x_j[\widetilde{i}]$ if it is in 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. The syntax requires an output to be followed by an input process, as in [43]. If one needs to output several messages consecutively, one can simply 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.

An else branch of find or if may be omitted when it is $\mathsf{else}\ \overline{yield}\langle\rangle; 0$. (Note that "$\mathsf{else}\ 0$" would not be syntactically correct.) Similarly, $\overline{yield}\langle\rangle; 0$ may be omitted after an event or a restriction. A trailing 0 after an output may be omitted.

The *current replication indices* 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 indices, but it should be kept in mind that this is only an abbreviation. Variables x defined under a replication must be arrays with indices the current replication indices at the definition of x: for example, $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m} \mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ \ldots$ More formally, we require the following invariant:

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

1. in every definition of $x[i_1, \ldots, i_m]$ in Q_0, the indices $i_1, \ldots, i_m$ of x are the current replication indices at that definition, and
2. two different definitions of the same variable x in Q_0 are in different branches of a find (or if).

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

We use a type system, detailed in [23, Appendix A], to check that bitstrings of the proper type are given to each function and that array indices are used correctly.

Invariant 3 (Typing) The process Q_0 satisfies Invariant 3 if and only if it is well-typed.

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. The three invariants are checked by the prover for the initial game and preserved by all game transformations.

The formal semantics is defined by a probabilistic reduction relation [23, Appendix B]. Our semantics is such that all processes can be simulated by probabilistic Turing machines, and conversely.

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 are efficiently computable, 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} is efficiently computable. 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.

We denote by $\mathrm{var}(Q)$ the set of variables that occur in Q.

1.2. Example

Let us introduce two cryptographic primitives that we use below.

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 scheme MAC [17] consists of three function symbols:

- $\mathrm{mkgen} : T_{mr} \to T_{mk}$ 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.)

- $\mathrm{mac} : \mathit{bitstring} \times T_{mk} \rightarrow T_{ms}$ is the MAC algorithm taking as arguments 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.)
- $\mathrm{verify} : \mathit{bitstring} \times T_{mk} \times T_{ms} \rightarrow \mathit{bool}$ is a verification algorithm such that $\mathrm{verify}(m, k, t) = \mathrm{true}$ if and only if t is a valid MAC of message m under key k. (Since mac is deterministic, $\mathrm{verify}(m, k, t)$ is typically $\mathrm{mac}(m, k) = t$.)

We have $\forall m \in \mathit{bitstring}, \forall r \in T_{mr}, \mathrm{verify}(m, \mathrm{mkgen}(r), \mathrm{mac}(m, \mathrm{mkgen}(r))) = \mathrm{true}$.

The advantage of an adversary against unforgeability under chosen message attacks (UF-CMA) is

$$\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t, q_m, q_v, l) = \max_{\mathcal{A}} \Pr \left[\begin{array}{l} r \xleftarrow{R} T_{mr}; k \leftarrow \mathrm{mkgen}(r); \\ (m, s) \leftarrow \mathcal{A}^{\mathrm{mac}(.,k), \mathrm{verify}(.,k,.)} : \mathrm{verify}(m, k, s) \\ \wedge\, m \text{ was never queried to the oracle } \mathrm{mac}(., k) \end{array} \right]$$

where the adversary $\mathcal{A}$ is any probabilistic Turing machine that runs in time at most t, calls $\mathrm{mac}(., k)$ at most q_m times with messages of length at most l, and calls $\mathrm{verify}(., k, .)$ at most q_v times with messages of length at most l.

$\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t, q_m, q_v, l)$ is the probability that an adversary forges a MAC, that is, returns a pair (m, s) where s is a correct MAC for m, without having queried the MAC oracle $\mathrm{mac}(., k)$ on m. Intuitively, when the MAC is secure, this probability is small: the adversary has little chance of forging a MAC. Hence, the MAC guarantees the integrity of the MACed message because one cannot compute the MAC without the secret key.

Two frameworks exist for expressing security properties. In the asymptotic framework, used in [22, 23], the length of keys is determined by a security parameter η, and a MAC is UF-CMA when $\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t, q_m, q_v, l)$ is a negligible function of η when t is polynomial in η. ($f(\eta)$ is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) The assumption that functions are efficiently computable means that they are computable in time polynomial in η and in the length of their arguments. The goal is to show that the probability of success of an attack against the protocol is negligible, assuming the parameters n are polynomial in η and the network messages are of length polynomial in η. In contrast, in the exact security framework, on which we focus in this course, one computes the probability of success of an attack against the protocol as a function of the probability of breaking the primitives such as $\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t, q_m, q_v, l)$, of the runtime of functions, of the parameters n, and of the length of messages, thus providing a more precise security result. Intuitively, the probability $\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t, q_m, q_v, l)$ is assumed to be small (otherwise, the computed probability of attack will be large), but no formal assumption on this probability is needed to establish the security theorem.

Definition 2 Let T_r and T'_r be fixed-length types representing random coins; let T_k and T_e be types for keys and ciphertexts respectively. A symmetric encryption scheme SE [17] consists of three function symbols:

- $\mathrm{kgen} : T_r \rightarrow T_k$ is the key generation algorithm taking as argument random coins and returning a key,

- $\mathrm{enc} : \mathit{bitstring} \times T_k \times T'_r \to T_e$ is the encryption algorithm taking as arguments the cleartext, the key, and random coins, and returning the ciphertext,
- $\mathrm{dec} : T_e \times T_k \to \mathit{bitstring}_\perp$ is the decryption algorithm taking as arguments the ciphertext and the key, and returning either the cleartext when decryption succeeds or $\perp$ when decryption fails,

such that $\forall m \in \mathit{bitstring}, \forall r \in T_r, \forall r' \in T'_r, \mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = m$.

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. The advantage of an adversary against indistinguishability under chosen plaintext attacks (IND-CPA) is

$$\mathsf{Succ}^{\mathsf{ind-cpa}}_{\mathsf{SE}}(t, q_e, l) = \max_{\mathcal{A}} 2\Pr\left[\begin{array}{l} b \xleftarrow{R} \{0,1\}; r \xleftarrow{R} T_r; k \leftarrow \mathrm{kgen}(r); \\ b' \leftarrow \mathcal{A}^{r' \xleftarrow{R} T'_r; \mathrm{enc}(LR(.,.,b),k,r')} : b' = b \end{array}\right] - 1$$

where $\mathcal{A}$ is any probabilistic Turing machine that runs in time at most t and calls $r' \xleftarrow{R} T'_r; enc(LR(.,.,b), k, r')$ at most q_e times on messages of length at most l.

Given two bitstrings a_0 and a_1 of the same length, the left-right encryption oracle $r' \xleftarrow{R} T'_r; enc(LR(.,.,b), k, r')$ returns $r' \xleftarrow{R} T'_r; \mathrm{enc}(LR(a_0, a_1, b), k, r')$, that is, encrypts a_0 when $b = 0$ and a_1 when $b = 1$. $\mathsf{Succ}^{\mathsf{ind-cpa}}_{\mathsf{SE}}(t, q_e, l)$ is the probability that the adversary distinguishes the encryption of the messages a_0 given as first arguments to the left-right encryption oracle from the encryption of the messages a_1 given as second arguments. Intuitively, when the encryption scheme is IND-CPA secure, this probability is small: the ciphertext gives almost no information what the cleartext is (one cannot determine whether it is a_0 or a_1 without having the secret key).

Example 1 Let us consider the following trivial protocol:

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

A and B are assumed to share a key x_k for a symmetric encryption scheme 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. In other words, the protocol sends the key x'_k encrypted using an encrypt-then-MAC scheme [17]. 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:

$$Q_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in}$$
$$\mathsf{new}\ x_{mr} : T_{mr}; \mathsf{let}\ x_{mk} : T_{mk} = \mathrm{mkgen}(x_{mr})\ \mathsf{in}\ \overline{c}\langle\rangle; (Q_A \mid Q_B)$$
$$Q_A = !^{i \leq n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x'_r : T'_r;$$
$$\mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x'_r)\ \mathsf{in}\ \overline{c_A[i]}\langle x_m, \mathrm{mac}(x_m, x_{mk})\rangle$$
$$Q_B = !^{i' \leq n} c_B[i'](x'_m, x_{ma}); \mathsf{if}\ \mathrm{verify}(x'_m, x_{mk}, x_{ma})\ \mathsf{then}$$
$$\mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle$$

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 adversary, by outputting on channel c. After this output, n copies of processes for A and B are ready to be executed, when the adversary outputs on channels $c_A[i]$ or $c_B[i]$ respectively. In a session that runs as expected, the adversary 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 under x_{mk} of the ciphertext, and sends the ciphertext and the MAC on $c_A[i]$. The function $\mathrm{k2b} : T_k \to bitstring$ is the natural injection $\mathrm{k2b}(x) = x$; it is needed only for type conversion. The adversary is then expected to forward this message on $c_B[i]$. When Q_B receives this message, it verifies the MAC, decrypts, and stores the obtained key in x''_k. (The function $\mathrm{i}_\perp : bitstring \to bitstring_\perp$ is the natural injection; it is useful to check that decryption succeeded.) This key x''_k should be secret.

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

This very small example is sufficient to illustrate the main features of CryptoVerif. Section 5 presents results obtained on more realistic protocols.

1.3. Observational Equivalence

Let us now formally define game indistinguishability, which we name observational equivalence by analogy with that notion in the Dolev-Yao model. 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. The executed events can be used to distinguish games, so we introduce an additional algorithm, a *distinguisher* D that takes as input a sequence of events $\mathcal{E}$ and returns true or false. An example of distinguisher is D_e defined by $D_e(\mathcal{E}) = \mathrm{true}$ if and only if $e \in \mathcal{E}$: this distinguisher detects the execution of event e. More generally, distinguishers can detect various properties of the sequence of events $\mathcal{E}$ executed by the game. We denote by $\Pr[Q \rightsquigarrow D]$ the probability that Q executes a sequence of events $\mathcal{E}$ such that $D(\mathcal{E})$ returns true.

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 with public variables V if and only if $\mathrm{var}(C) \cap \mathrm{var}(Q) \subseteq V$ and $C[Q]$ satisfies Invariants 1, 2, and 3.

We say that Q and Q' are *observationally equivalent* up to probability p with public variables V, written $Q \approx_p^V Q'$, when for all evaluation contexts C acceptable for Q and Q' with public variables V, for all distinguishers D, $|\Pr[C[Q] \rightsquigarrow D] - \Pr[C[Q'] \rightsquigarrow D]| \leq p(C, D)$.

This definition formalizes that algorithms C and D distinguish Q and Q' with probability at most $p(C, D)$. The probability p typically depends on the runtime of C and D, but may also depend on other parameters, such as the number of messages sent by C to each replicated process. That is why p takes as arguments C and D themselves.

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'.) When V is empty, we write $Q \approx_p Q'$ instead of $Q \approx_p^V Q'$.

The following result is not difficult to prove:

Lemma 1 1. *Reflexivity:* $Q \approx_0^V Q$.
2. *Symmetry: if* $Q \approx_p^V Q'$, *then* $Q' \approx_p^V Q$.
3. *Transitivity: if* $Q \approx_p^V Q'$ *and* $Q' \approx_{p'}^V Q''$, *then* $Q \approx_{p+p'}^V Q''$.
4. *If* $Q \approx_p^V Q'$ *and* C *is an evaluation context acceptable for* Q *and* Q' *with public variables* V, *then* $C[Q] \approx_{p'}^{V'} C[Q']$, *where* $p'(C', D) = p(C'[C], D)$ *and* $V' \subseteq V \cup \mathrm{var}(C)$.

Proofs by sequences of games consist of a sequence of observationally equivalent games $Q_0 \approx_{p_1}^V Q_1 \approx_{p_2}^V \ldots \approx_{p_n}^V Q_n$. By transitivity, $Q_0 \approx_{p_1+\ldots+p_n}^V Q_n$, so by definition of observational equivalence, $\Pr[C[Q_0] \rightsquigarrow D] \leq \Pr[C[Q_n] \rightsquigarrow D] + (p_1 + \ldots + p_n)(C, D)$.

2. 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 3. These transformations are parameterized by the set V of variables that the context can access. As we shall see in Section 3, V contains variables that we would like to prove secret. (The context will contain test queries that access these variables.) These transformations transform a process Q_0 into a process Q_0' such that $Q_0 \approx_p^V Q_0'$; CryptoVerif evaluates the probability p.

2.1. Syntactic Transformations

RemoveAssign(x): When x is defined by an assignment let $x[i_1, \ldots, i_l] : T = M$ in P and x does not occur in M (non-cyclic assignment), we replace x with its value. 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 make sure that, if a condition of find guarantees that $x[M_1, \ldots, M_l]$ is defined in the initial game, then so does the corresponding condition of find in the transformed 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 $\mathrm{mkgen}(x_{mr})$ for x_{mk} in the whole process and removes the assignment let $x_{mk} : T_{mk} = \mathrm{mkgen}(x_{mr})$. After substitution, $\mathrm{mac}(x_m, x_{mk})$ becomes $\mathrm{mac}(x_m, \mathrm{mkgen}(x_{mr}))$ and $\mathrm{verify}(x'_m, x_{mk}, x_{ma})$ becomes $\mathrm{verify}(x'_m, \mathrm{mkgen}(x_{mr}), x_{ma})$, thus exhibiting terms required in Section 2.2. 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 $x[\widetilde{i}]$. 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 $x[\widetilde{i}]$ under a definition of $x_j[\widetilde{i}]$ are then replaced with $x_j[\widetilde{i}]$. Each branch of find $FB = \widetilde{u}[\widetilde{i}] \leq \widetilde{n}$ suchthat defined$(M_1, \ldots, M_l) \wedge M$ then P where $x[\widetilde{M}]$ is a subterm of some M_k for $k \leq l$ is replaced with m branches $FB\{x_j[\widetilde{M}]/x[\widetilde{M}]\}$ 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 [40]. 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 values of x_1 in T_1, ..., x_m in T_m, M evaluates to true. For example, considering MAC and encryption schemes as in Definitions 1 and 2 respectively, we have:

$$\forall r : T_{mr}, \forall m : \mathit{bitstring}, \mathrm{verify}(m, \mathrm{mkgen}(r), \mathrm{mac}(m, \mathrm{mkgen}(r))) = \mathrm{true} \quad \text{(mac)}$$

$$\forall m : \mathit{bitstring}; \forall r : T_r, \forall r' : T'_r, \mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = \mathrm{i}_\perp(m) \quad \text{(enc)}$$

  We express the poly-injectivity of the function $\mathrm{k2b}$ of Example 1 by

$$\begin{aligned} &\forall x : T_k, \forall y : T_k, (\mathrm{k2b}(x) = \mathrm{k2b}(y)) = (x = y) \\ &\forall x : T_k, \mathrm{k2b}^{-1}(\mathrm{k2b}(x)) = x \end{aligned} \quad \text{(k2b)}$$

  where $\mathrm{k2b}^{-1}$ is a function symbol that denotes the inverse of $\mathrm{k2b}$. We have similar formulas for $\mathrm{i}_\perp$.
- Equations that come from the process. For example, in the process if M then P else P', we have $M = \mathrm{true}$ in P and $M = \mathrm{false}$ in P'.
- The low probability of collision between random values. For example, when x is defined by new $x : T$ under replications bounded by $n_1, \ldots, n_m$, $x[M_1, \ldots, M_m] = x[M'_1, \ldots, M'_m]$ implies $M_1 = M'_1$, ..., $M_m = M'_m$ up to probability $p = \frac{(n_1 \ldots n_m)^2}{2|T|}$ (probability that two distinct cells of the array x are equal). This transformation is performed when the type T is *large*, which means that $|T|$ is large enough so that the probability p can be considered small.
  Similarly, when 1) x is defined by new $x : T$ and T is a large type, 2) for each value of M_1, there is at most one value of x (or of a part of x of a large type) that can yield that value of M_1, and 3) M_2 does not depend on x, then $M_1 \neq M_2$ up to a small probability. The fact that M_2 does not depend on x is proved using a dependency analysis.

The prover combines these properties to simplify terms, and uses simplified forms of terms to simplify processes. For example, if M simplifies to true, then if M then P else P' simplifies to P. Similarly, a branch of find is removed when the associated condition simplifies to false.

Details on the simplification procedure can be found in [23, Appendix C]. The asymptotic version of the following proposition is proved in [23, Appendix E.1].

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 $Q_0 \approx_p^V Q'_0$, where $p = 0$ for the transformations **RemoveAssign** and **SArename**, and p is the probability of eliminated collisions for **Simplify**.*

2.2. Applying the Security Assumptions on 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 below.

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

$G ::=$	group of oracles
$\quad !^{i \leq n} \mathsf{new}\ y_1 : T_1; \ldots; \mathsf{new}\ y_l : T_l; (G_1, \ldots, G_m)$	replication, restrictions
$\quad O(x_1 : T_1, \ldots, x_l : T_l) := OP$	oracle
$OP ::=$	oracle processes
$\quad M$	term
$\quad \mathsf{new}\ x[\widetilde{i}] : T; OP$	random number
$\quad \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ OP$	assignment
$\quad \mathsf{find}\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}\ \mathsf{suchthat}$	
$\quad\quad \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ OP_j)\ \mathsf{else}\ OP$	array lookup

Intuitively, $O(x_1 : T_1, \ldots, x_l : T_l) := OP$ represents an oracle O that takes as argument values $x_1, \ldots, x_l$ of types $T_1, \ldots, T_l$ respectively and returns a result computed by OP. The observational equivalence $(G_1, \ldots, G_m) \approx_p (G'_1, \ldots, G'_m)$ expresses that the adversary has probability at most p of distinguishing oracles in the left-hand side from corresponding oracles in the right-hand side. Formally, oracles can be encoded as processes that input their arguments and output their result on a channel, as detailed in [23]. Denoting by $[\![(G_1, \ldots, G_m)]\!]$ the encoding of $(G_1, \ldots, G_m)$ as a process, the observational equivalence $(G_1, \ldots, G_m) \approx_p (G'_1, \ldots, G'_m)$ is then an abbreviation for $[\![(G_1, \ldots, G_m)]\!] \approx_p [\![(G'_1, \ldots, G'_m)]\!]$.

For example, the security of a MAC (Definition 1) is represented by the equivalence $L \approx_{p_{\mathrm{mac}}} R$ where:

$$
\begin{aligned}
L = {} & !^{i'' \leq n''} \mathsf{new}\ r : T_{mr}; (\\
& !^{i \leq n} Omac(x : bitstring) := \mathrm{mac}(x, \mathrm{mkgen}(r)), \\
& !^{i' \leq n'} Overify(m : bitstring, ma : T_{ms}) := \mathrm{verify}(m, \mathrm{mkgen}(r), ma))
\end{aligned}
$$

$$
\begin{aligned}
R = {} & !^{i'' \leq n''} \mathsf{new}\ r : T_{mr}; (\\
& !^{i \leq n} Omac(x : bitstring) := \mathrm{mac}'(x, \mathrm{mkgen}'(r)),\\
& !^{i' \leq n'} Overify(m : bitstring, ma : T_{ms}) :=\\
& \quad \mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge (m = x[u])\\
& \quad \wedge \mathrm{verify}'(m, \mathrm{mkgen}'(r), ma)\ \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false})
\end{aligned}
$$

$$
\begin{aligned}
p_{\mathrm{mac}}(C, D) = n'' \, \mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t_C + (n'' - 1)(\mathrm{time}(\mathrm{mkgen}) + n\,\mathrm{time}(\mathrm{mac}, \mathrm{maxl}(x))\\
+ n'\,\mathrm{time}(\mathrm{verify}, \mathrm{maxl}(m)), n, n', \max(\mathrm{maxl}(x), \mathrm{maxl}(m)))
\end{aligned}
\qquad (\mathrm{mac}_{\mathrm{eq}})
$$

where mac', verify', and mkgen' are function symbols with the same types as mac, verify, 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

$$
\forall r : T_{mr}, \forall m : bitstring, \mathrm{verify}'(m, \mathrm{mkgen}'(r), \mathrm{mac}'(m, \mathrm{mkgen}'(r))) = \mathrm{true} \qquad (\mathrm{mac}')
$$

which restates (mac) for mac', verify', and mkgen'.) Intuitively, the equivalence $L \approx_{p_{\mathrm{mac}}} R$ leaves MAC computations unchanged (except for the use of primed function symbols in R), and allows one to replace a MAC verification $\mathrm{verify}(m, \mathrm{mkgen}(r), ma)$ with a lookup in the array x of messages whose mac has been computed with key $\mathrm{mkgen}(r)$: if m is found in the array x and $\mathrm{verify}(m, \mathrm{mkgen}(r), ma)$, we return true; otherwise, the verification fails (up to negligible probability), so we return false. (If the verification succeeds with m not in the array x, then the adversary has forged a MAC.) Obviously, the form of L requires that r is used only to compute or verify MACs, for the equivalence to be correct. In the probability $p_{\mathrm{mac}}(C, D)$, t_C is the runtime of context C; n'' is the maximum number of considered MAC keys; n' and n'' are respectively the maximum number of calls to $Omac$ and $Overify$ for each MAC key (n, n', n'' are in fact functions of C); $\mathrm{time}(f, l_1, \ldots, l_k)$ is the maximum runtime of f, called with arguments of length at most $l_1, \ldots, l_k$ (the lengths $l_1, \ldots, l_k$ are omitted when the type of the argument already bounds its length); $\mathrm{maxl}(x)$ is the maximum length of x. Formally, the following result shows the correctness of our modeling. It is a fairly easy consequence of Definition 1, and its asymptotic version is proved in [23, Appendix E.3].

Proposition 2 *If* $(\mathrm{mkgen}, \mathrm{mac}, \mathrm{verify})$ *is a UF-CMA message authentication code and the symbols* mkgen', mac', *and* verify' *represent the same functions as* mkgen, mac, *and* verify *respectively, then* $[\![L]\!] \approx_{p_{\mathrm{mac}}} [\![R]\!]$.

Similarly, if $(\mathrm{kgen}, \mathrm{enc}, \mathrm{dec})$ is an IND-CPA symmetric encryption scheme (Definition 2), then we have the following equivalence:

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

where $p_{\mathrm{enc}}(C, D) = n' \, \mathsf{Succ}_{\mathsf{SE}}^{\mathsf{ind-cpa}}(t_C + t_D + (n'-1)(\mathrm{time}(\mathrm{kgen}) + n\,\mathrm{time}(\mathrm{enc}, \mathrm{maxl}(x)) + n\,\mathrm{time}(Z, \mathrm{maxl}(x))), n, \mathrm{maxl}(x))$, enc' and kgen' are function symbols with the same types as enc and kgen respectively, and $\mathrm{Z} : \mathit{bitstring} \to \mathit{bitstring}$ is the function that returns a bitstring of the same length as its argument, consisting only of zeroes. Using equations such as $\forall x : T, \mathrm{Z}(\mathrm{T2b}(x)) = \mathrm{Z}_T$, we can prove that $\mathrm{Z}(\mathrm{T2b}(x))$ does not depend on x when x is of a fixed-length type T and $\mathrm{T2b} : T \to \mathit{bitstring}$ is the natural injection. The representation of other primitives can be found in [23, Appendix D.3]. The equivalences that formalize the security assumptions on primitives are designed and proved correct by hand from security assumptions in a more standard form, as in the MAC example. Importantly, these manual proofs are done only once for each primitive, and the obtained equivalence can be reused for proving many different protocols automatically.

Assuming $L \approx_p R$, Lemma 1 yields $C[[\![L]\!]] \approx_{p'}^{V} C[[\![R]\!]]$ with $p'(C', D) = p(C'[C], D)$, for all evaluation contexts C acceptable for $[\![L]\!]$ and $[\![R]\!]$ with no public variables, so we can transform a process Q_0 such that $Q_0 \approx_0^V C[[\![L]\!]]$ into a process Q_0' such that $Q_0 \approx_0^V C[[\![L]\!]] \approx_{p'}^{V} C[[\![R]\!]] \approx_0^V Q_0'$. In order to check that $Q_0 \approx_0^V C[[\![L]\!]]$, the prover uses syntactic conditions detailed in [23, Appendix D.1] and sketched in Example 3 below. The following proposition shows the soundness of the transformation; its asymptotic version is proved in [23, Appendix E.4].

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 $[\![L]\!] \approx_p [\![R]\!]$, then $Q_0 \approx_{p'}^{V} Q_0'$ where $p'(C', D) = p(C'[C], D)$ and C is an evaluation context such that $Q_0 \approx_0^V C[[\![L]\!]] \approx_{p'}^{V} C[[\![R]\!]] \approx_0^V Q_0'$.*

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 $\mathrm{mkgen}, \mathrm{mkgen}' : T_{mr} \to T_{mk}$, $\mathrm{mac}, \mathrm{mac}' : \mathit{bitstring} \times T_{mk} \to T_{ms}$, $\mathrm{verify}, \mathrm{verify}' : \mathit{bitstring} \times T_{mk} \times T_{ms} \to \mathit{bool}$, $\mathrm{kgen}, \mathrm{kgen}' : T_r \to T_k$, $\mathrm{enc}, \mathrm{enc}' : \mathit{bitstring} \times T_k \times T_r' \to T_e$, $\mathrm{dec} : T_e \times T_k \to \mathit{bitstring}_\perp$, $\mathrm{k2b} : T_k \to \mathit{bitstring}$, $\mathrm{i}_\perp : \mathit{bitstring} \to \mathit{bitstring}_\perp$, $\mathrm{Z} : \mathit{bitstring} \to \mathit{bitstring}$, and the constant $\mathrm{Z}_k : \mathit{bitstring}$; the equations (mac), (mac'), (enc), and $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (which expresses that all keys have the same length); the indication that $\mathrm{k2b}$ and $\mathrm{i}_\perp$ are poly-injective (which generates the equations $(\mathrm{k2b})$ and similar equations for $\mathrm{i}_\perp$); equivalences $L \approx_p R$ for MAC ($\mathrm{mac}_{\mathrm{eq}}$) and encryption ($\mathrm{enc}_{\mathrm{eq}}$); and the process Q_0 of Example 1. Let $V = \{x_k''\}$.

The prover first applies **RemoveAssign**(x_{mk}) to the process Q_0 of Example 1, as described in Example 2, yielding Q_1. The process can then be transformed using the security of the MAC. In the equivalence $L \approx_{p_{\mathrm{mac}}} R$ ($\mathrm{mac}_{\mathrm{eq}}$) that expresses the security of the MAC, L is an abbreviation for the process:

$$[\![L]\!] = !^{i'' \leq n''} c_{\mathrm{mkgen}}[i'']();\, \mathsf{new}\ r : T_{mr};\, \overline{c_{\mathrm{mkgen}}[i'']}\langle\rangle;\, (\\ !^{i \leq n} c_{\mathrm{mac}}[i'', i](x : \mathit{bitstring});\, \overline{c_{\mathrm{mac}}[i'', i]}\langle \mathrm{mac}(x, \mathrm{mkgen}(r))\rangle \mid \\ !^{i' \leq n'} c_{\mathrm{verify}}[i'', i'](m : \mathit{bitstring}, ma : T_{ms});\, \overline{c_{\mathrm{verify}}[i'', i']}\langle \mathrm{verify}(m, \mathrm{mkgen}(r), ma)\rangle)$$

The process Q_1 can be written under the form $C[[\![L]\!]]$, $Q_1 \approx_0^V C[[\![L]\!]]$, for the following context C:

$C = \mathsf{newChannel}\ c_{\mathrm{mkgen}}; \mathsf{newChannel}\ c_{\mathrm{mac}}; \mathsf{newChannel}\ c_{\mathrm{verify}}; ([\,] \mid start();$

$\quad \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in}\ \overline{c_{\mathrm{mkgen}}[1]}\langle\rangle; c_{\mathrm{mkgen}}[1](); \overline{c}\langle\rangle; (Q_{CA} \mid Q_{CB}))$

$Q_{CA} = !^{i \leq n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x'_r : T'_r;$

$\qquad \mathsf{let}\ x_m : bitstring = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x'_r)\ \mathsf{in}$

$\qquad \overline{c_{\mathrm{mac}}[1,i]}\langle x_m\rangle; c_{\mathrm{mac}}[1,i](x_{ma}); \overline{c_A[i]}\langle x_m, x_{ma}\rangle$

$Q_{CB} = !^{i' \leq n} c_B[i'](x'_m, x_{ma}); \overline{c_{\mathrm{verify}}[1,i']}\langle x'_m, x_{ma}\rangle; c_{\mathrm{verify}}[1,i'](b); \mathsf{if}\ b\ \mathsf{then}$

$\qquad \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle$

Instead of generating the coins x_{mr} for the MAC key itself, this context sends a message on channel $c_{\mathrm{mkgen}}[1]$, which is received by $[\![L]\!]$, so that $[\![L]\!]$ generates these coins. Similarly, instead of computing the MAC, the context C sends the message to MAC on channel $c_{\mathrm{mac}}[1,i]$, so that $[\![L]\!]$ computes the MAC and sends it back on $c_{\mathrm{mac}}[1,i]$. Instead of verifying the MAC, C sends the message and the candidate MAC on channel $c_{\mathrm{verify}}[1,i']$, so that $[\![L]\!]$ verifies the MAC and sends the result back on $c_{\mathrm{verify}}[1,i']$. The channels c_{mkgen}, c_{mac} and c_{verify} are declared private by newChannel, so that the adversary cannot directly access $[\![L]\!]$.

Informally, the conditions verified by CryptoVerif to prove that $Q_1 \approx_0^V C[[\![L]\!]]$ show that there is a correspondence between the variables of L and terms or variables of Q_1. In the example, $r[1]$ in L corresponds to x_{mr} in Q_1, $x[1,a]$ to $x_m[a]$, $m[1,a']$ to $x'_m[a']$, and $ma[1,a']$ to $x_{ma}[a']$. This correspondence must be such that

- A variable $x[\widetilde{a}]$ bound by $\mathsf{new}\ x : T$ in L must correspond to a variable $z[\widetilde{a''}]$ bound by $\mathsf{new}\ z : T$ in Q_1, and the relation that associates $z[\widetilde{a''}]$ to $x[\widetilde{a'}]$ must be an injective function (so that independent random numbers in L correspond to independent random numbers in Q_1).
- An oracle argument $x[\widetilde{a}]$ in L must correspond to a term of the same type as x, and when two terms correspond to the same $x[\widetilde{a}]$, they must evaluate to the same value.
- If L contains an oracle $O(x_1 : T_1, \ldots, x_l : T_l) := M$, the term obtained by replacing the variables of M with their corresponding terms or variables of Q_1 is a term of Q_1. The variables z of Q_1 corresponding to variables x bound by $\mathsf{new}\ x : T$ in L occur only in such terms, at occurrences corresponding to occurrences of x in L. These variables z do not belong to V. In the example, $\mathrm{mac}(x[1,a], \mathrm{mkgen}(r[1]))$ in L corresponds to $\mathrm{mac}(x_m[a], \mathrm{mkgen}(x_{mr}))$ in Q_1 and $\mathrm{verify}(m[1,a'], \mathrm{mkgen}(r[1]), ma[1,a'])$ corresponds to $\mathrm{verify}(x'_m[a'], \mathrm{mkgen}(x_{mr}), x_{ma}[a'])$. The variable x_{mr} does not occur anywhere else in Q_1 and $x_{mr} \notin V$.

CryptoVerif then transforms Q_1 into $C[[\![R]\!]]$, which after some syntactic reorganizations yields the following process Q_2:

$Q_2 = start(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in}\ \mathsf{new}\ x_{mr} : T_{mr}; \overline{c}\langle\rangle; (Q_{2A} \mid Q_{2B})$

$Q_{2A} = !^{i \leq n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x'_r : T'_r;$

$\quad \mathsf{let}\ x_m : bitstring = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x'_r)\ \mathsf{in}\ \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x_{mr}))\rangle$

$Q_{2B} = !^{i' \leq n} c_B[i'](x'_m, x_{ma});$

find $u \leq n$ suchthat defined$(x_m[u]) \wedge x'_m = x_m[u] \wedge$

$\quad \mathrm{verify}'(x'_m, \mathrm{mkgen}'(x_{mr}), x_{ma})$

then (if true then let $\mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)$ in $\overline{c_B[i']}\langle\rangle$)

else (if false then let $\mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)$ in $\overline{c_B[i']}\langle\rangle$)

The initial definition of x_{mr} is removed and replaced with a new definition, which we still call x_{mr}. The term $\mathrm{mac}(x_m, \mathrm{mkgen}(x_{mr}))$ is replaced with $\mathrm{mac}'(x_m, \mathrm{mkgen}'(x_{mr}))$. The term $\mathrm{verify}(x'_m, \mathrm{mkgen}(x_{mr}), x_{ma})$ becomes find $u \leq n$ suchthat defined$(x_m[u])$ $\wedge x'_m = x_m[u] \wedge \mathrm{verify}'(x'_m, \mathrm{mkgen}'(x_{mr}), x_{ma})$ then true else false, which yields Q_{2B} after transformation of oracle 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 $\mathrm{mkgen}(x_{mr})$. If the MAC of x'_m has never been computed, the verification always fails (it returns false) by the security assumption on the MAC. Otherwise, it returns true when $\mathrm{verify}'(x'_m, \mathrm{mkgen}'(x_{mr}), x_{ma})$. By instantiating the probability formula given in ($\mathrm{mac}_{\mathsf{eq}}$), $Q_1 \approx_{p'_{\mathrm{mac}}} Q_2$ where $p'_{\mathrm{mac}}(C, D) = p_{\mathrm{mac}}(C[C'], D) = \mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{MAC}}(t_C + \mathrm{time}(\mathrm{kgen}) + n\,\mathrm{time}(\mathrm{enc}, \mathrm{length}(T_k)) + n\,\mathrm{time}(\mathrm{dec}, \mathrm{maxl}(x'_m)), n, n, \max(\mathrm{maxl}(x'_m), \mathrm{maxl}(x_m)))$ since we use one MAC key ($n'' = 1$), there are at most n calls to mac and verify for that key ($n' = n$), and the runtime of the adversary against ($\mathrm{mac}_{\mathsf{eq}}$) is $t_{C[C']} = t_C + \mathrm{time}(\mathrm{kgen}) + n\,\mathrm{time}(\mathrm{enc}, \mathrm{length}(T_k)) + n\,\mathrm{time}(\mathrm{dec}, \mathrm{maxl}(x'_m))$.

Applying **Simplify** yields a game Q_3: Q_{2A} is unchanged and Q_{2B} becomes

$Q_{3B} = !^{i' \leq n} c_B[i'](x'_m, x_{ma});$

find $u \leq n$ suchthat defined$(x_m[u], x'_k[u]) \wedge x'_m = x_m[u] \wedge$

$\quad \mathrm{verify}'(x'_m, \mathrm{mkgen}'(x_{mr}), x_{ma})$ then

let $x''_k : T_k = x'_k[u]$ in $\overline{c_B[i']}\langle\rangle$

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

After applying **RemoveAssign**(x_k), which yields Q_4, we use the security of encryption, yielding Q_5: $\mathrm{enc}(\mathrm{k2b}(x'_k), \mathrm{kgen}(x_r), x'_r)$ becomes $\mathrm{enc}'(\mathrm{Z}(\mathrm{k2b}(x'_k)), \mathrm{kgen}'(x_r), x'_r)$. We have $Q_4 \approx_{p'_{\mathrm{enc}}} Q_5$ where $p'_{\mathrm{enc}}(C, D) = p_{\mathrm{enc}}(C[C''], D) = \mathsf{Succ}^{\mathsf{ind-cpa}}_{\mathsf{SE}}(t_C + t_D + (n + n^2)\mathrm{time}(\mathrm{mkgen}) + n\,\mathrm{time}(\mathrm{mac}, \mathrm{maxl}(m)) + n^2\,\mathrm{time}(\mathrm{verify}, \mathrm{maxl}(m')) + n^2\,\mathrm{time}(=\!bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(T_k))$. (The evaluation of the runtime of the context C'' is rather naive since we consider that $\mathrm{mkgen}(x_{mr})$ is computed once in each execution of Q_{4A} and once for each find test in Q_{4B}, and similarly verify is computed once for each find test in Q_{4B}. By noticing that it is enough to compute $\mathrm{mkgen}(x_{mr})$ once, and verify once in each execution of Q_{4B}, one would

obtain $\mathsf{Succ}^{\mathsf{ind-cpa}}_{\mathsf{SE}}(t_C + t_D + \mathrm{time}(\mathrm{mkgen}) + n\,\mathrm{time}(\mathrm{mac}, \mathrm{maxl}(m)) + n\,\mathrm{time}(\mathrm{verify}, \mathrm{maxl}(m')) + n^2\,\mathrm{time}(=\ bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(T_k))$.) After **Simplify**, $\mathrm{enc}'(\mathrm{Z}(\mathrm{k2b}(x'_k)), \mathrm{kgen}'(x_r), x'_r)$ becomes $\mathrm{enc}'(\mathrm{Z}_k, \mathrm{kgen}'(x_r), x'_r)$, using $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (which expresses that all keys have the same length).

So we obtain the following game:

$$Q_6 = start(); \mathsf{new}\ x_r : T_r; \mathsf{new}\ x_{mr} : T_{mr}; \overline{c}\langle\rangle; (Q_{6A} \mid Q_{6B})$$

$$Q_{6A} = !^{i \leq n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x'_r : T'_r;$$

$$\mathsf{let}\ x_m : bitstring = \mathrm{enc}'(\mathrm{Z}_k, \mathrm{kgen}'(x_r), x'_r)\ \mathsf{in}\ \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x_{mr}))\rangle$$

$$Q_{6B} = Q_{3B}$$

By transitivity of $\approx$ (Lemma 1), $Q_0 \approx^V_{p'_{\mathrm{mac}} + p'_{\mathrm{enc}}} Q_6$ since the probability is 0 for steps other than applying the security of MAC and encryption.

Using lists instead of arrays simplifies games transformations: 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 $\mathrm{mac}(x_i, k)$ with the same key in the initial process, each $\mathrm{verify}(m_j, k, ma_j)$ is replaced with a find with one branch for each occurrence of mac. Therefore, the prover distinguishes automatically the cases in which the verified 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 steps of the proof of the protocol. (A similar situation arises for other cryptographic primitives specified using find.)

3. Criteria for Proving Secrecy Properties

Let us now define syntactic criteria that allow us to prove secrecy properties of protocols. The proofs of asymptotic versions of these results can be found in [23, Appendix E.5].

Definition 4 (One-session secrecy) Suppose that the variable x of type T is defined in Q under a single $!^{i \leq n}$. *Q preserves the one-session secrecy of* x up to probability p when, for all evaluation contexts C acceptable for $Q \mid Q_x$ without public variables that do not contain S, $2\Pr[C[Q \mid Q_x] \rightsquigarrow D_{\mathsf{S}}] - 1 \leq p(C)$ where $D_{\mathsf{S}}(\mathcal{E}) = (\mathsf{S} \in \mathcal{E})$,

$$\begin{aligned} Q_x = {} & c_0(); \mathsf{new}\ b : bool; \overline{c_0}\langle\rangle; \\ & (c(u : [1, n]); \mathsf{if}\ \mathsf{defined}(x[u])\ \mathsf{then}\ \mathsf{if}\ b\ \mathsf{then}\ \overline{c}\langle x[u]\rangle\ \mathsf{else}\ \mathsf{new}\ y : T; \overline{c}\langle y\rangle \\ & \mid c'(b' : bool); \mathsf{if}\ b = b'\ \mathsf{then}\ \mathsf{event}\ \mathsf{S}) \end{aligned}$$

c_0, c, c', b, b', u, y, and S do not occur in Q.

Intuitively, the adversary C distinguishes the value of each secret $x[u]$ from a random number with probability at most $p(C)$. The adversary performs a single test query on $x[u]$, modeled by sending u on channel c in Q_x. This test query returns $x[u]$ when the random bit b is true and a random number otherwise. The adversary then tries to guess b, by sending its guess b' on channel c'. When the guess is correct, event S is executed.

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 preserves the one-session secrecy of x up to probability 0.*

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) Assume that the variable x of type T is defined in Q under a single $!^{i \leq n}$. *Q preserves the secrecy of x* up to probability p when, for all evaluation contexts C acceptable for $Q \mid R_x$ without public variables that do not contain S, $2\Pr[C[Q \mid R_x] \rightsquigarrow D_{\mathsf{S}}] - 1 \leq p(C)$ where $D_{\mathsf{S}}(\mathcal{E}) = (\mathsf{S} \in \mathcal{E})$,

$$
\begin{aligned}
R_x = {} & c_0(); \mathsf{new}\ b : bool; \overline{c_0}\langle\rangle; \\
& (!^{i \leq n'} c(u : [1, n]); \mathsf{if}\ \mathsf{defined}(x[u])\ \mathsf{then}\ \mathsf{if}\ b\ \mathsf{then}\ \overline{c}\langle x[u]\rangle\ \mathsf{else} \\
& \quad \mathsf{find}\ u' \leq n'\ \mathsf{suchthat}\ \mathsf{defined}(y[u'], u[u']) \wedge u[u'] = u\ \mathsf{then}\ \overline{c}\langle y[u']\rangle \\
& \quad \mathsf{else}\ \mathsf{new}\ y : T; \overline{c}\langle y\rangle \\
& \mid c'(b' : bool); \mathsf{if}\ b = b'\ \mathsf{then}\ \mathsf{event}\ \mathsf{S})
\end{aligned}
$$

$c_0, c, c', b, b', u, u', y$, and S do not occur in Q, and $n' \geq n$.

Intuitively, the adversary C distinguishes the secret array x from an array of independent random numbers with probability at most $p(C)$. In this definition, the adversary can perform several test queries, modeled by R_x, which all return the value of x if b is true and a random number if b is false. This corresponds to the "real-or-random" definition of security [2]. (As shown in [2], 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 $x[u]$.)

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

If $\mathcal{T}$ 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, for all $k \leq l$, M_k evaluates to a'_k in the trace $\mathcal{T}$ at the definition of $x[\widetilde{a}]$.*

For all evaluation contexts C acceptable for Q with public variables $\{x\}$, let $p(C) = \Pr[\exists(\mathcal{T}, \widetilde{a}, \widetilde{a'}), C[Q]$ reduces according to $\mathcal{T} \wedge \widetilde{a} \neq \widetilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$. Then Q preserves the secrecy of x up to probability $2p$.

The collisions $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])$ are eliminated using the same equational prover as for **Simplify** in Section 2.1, which yields a bound on $p(C)$. Intuitively,

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 probability $p(C)$), so $x[\widetilde{a}]$ and $x[\widetilde{a'}]$ are defined by different restrictions, so they are independent random numbers.

As we show in [22], secrecy composed with correspondence assertions [56] 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". The verification of correspondence assertions in CryptoVerif in presented in [22].)

Lemma 2 *If $Q \approx_p^{\{x\}} Q'$ and Q preserves the one-session secrecy of x up to probability p' then Q' preserves the one-session secrecy of x up to probability $p''(C) = p'(C) + 2p(C[[\,] \mid Q_x], D_{\mathsf{S}})$. A similar 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 2 with $V = \{x\}$. By Propositions 1 and 3, we obtain a process Q'_0 such that $Q_0 \approx_p^V Q'_0$. 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 up to a certain probability 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 up to probability $2(p'_{\mathrm{mac}} + p'_{\mathrm{enc}})(C[[\,] \mid Q_x], D_{\mathsf{S}})$. (The runtimes of Q_x and D_{S} can be neglected inside this formula.) 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.

4. 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 2.2), the prover executes **Simplify** and tests whether the desired security properties are proved, as described in Section 3. 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.

Examples of advised transformations include:

- 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 $\mathrm{enc}(M', x_k, x'_r)$ and we look for $\mathrm{enc}(x_m, \mathrm{kgen}(x_r), x'_r)$, we advise **RemoveAssign**(x_k). If Q_0 contains let $x_k = \mathrm{mkgen}(x_r)$ and we look for $\mathrm{mac}(x_m, \mathrm{mkgen}(x_r))$, 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 want to prove that 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).
  When we want to prove that x is secret or one-session secret, we have an assignment let $x[\widetilde{i}] : T = y[\widetilde{M}]$ in P, y is defined by restrictions, y has several definitions, and some variable accesses to y are not of the form let $y'[\widetilde{i'}] : T = y[\widetilde{M'}]$ in P', we advise **SArename**(y).

5. Experimental Results

CryptoVerif has been tested on a number of protocols given in the literature. We proved secrecy of keys for the Otway-Rees and Yahalom protocols as well as original and corrected versions of the Needham-Schroeder shared-key and public-key and Denning-Sacco public-key protocols, as reported in [23]. We proved authentication properties for these protocols as well as for original and corrected versions of the Woo-Lam shared-key and public-key protocols [22]. The proof succeeded in most cases (it failed for only 3 properties that in fact hold). For some proofs, for public-key protocols, we needed to provide manual indications of the game transformations to perform, mainly because several game transformations are sometimes applicable, and the proof succeeds only for a particular choice of the applied game transformation.

For each proof, the prover outputs the sequence of games it has built, a succinct explanation of the transformation performed between consecutive games, and an indication of whether the proof succeeded or failed. When the proof fails, the prover still outputs a sequence of games, but the last game of this sequence does not show the desired property and cannot be transformed further by the prover. Manual inspection of this game often makes it possible to understand why the proof failed: because there is an attack (if there is an attack on the last game), because of a limitation of the prover (if it should in fact be able to prove the property or to transform the game further), for other reasons (such as the protocol cannot be proved from the given assumptions; this situation may not lead immediately to a practical attack in the computational model).

CryptoVerif can also be used for proving cryptographic schemes, such as the FDH signature scheme [25]. It has been used for studying more complex protocols: the Kerberos protocol, with and without its public-key extension PKINIT [24], as well as parts of the record protocol and of the handshake protocol of TLS [19].

6. Conclusion

CryptoVerif produces proofs by sequences of games, in the computational model. The security assumptions on primitives are given as observational equivalences, which are proved once for each primitive and can be reused for proving many different protocols. The protocol or cryptographic scheme to prove is specified in a process calculus. CryptoVerif provides the sequence of games that leads to the proof and a bound on the probability of success of an attack. The user is allowed, but does not have, to provide manual indications on the game transformations to perform.

The essential idea of simulating proofs by sequences of games in an automatic tool can be applied to any protocol or cryptographic scheme. However, CryptoVerif applies in a fairly direct way the security assumptions on the primitives and cannot perform deep mathematical reasoning. Therefore, it is best suited for proving security protocols that use rather high-level primitives such as encryption and signatures. It is more limited for proving the security of such primitives from lower-level primitives, since more subtle mathematical arguments are often needed.

Future work includes adding support for more primitives, for example associativity for exclusive or and primitives with internal state. Improvements in the proof strategy and the possibility to give more precise manual hints would also be useful. Future case studies will certainly suggest additional extensions. In the long term, it would be interesting to certify CryptoVerif, possibly to combine it with the Coq-based framework CertiCrypt [15]. Grand challenges include the proof of protocol implementations in the computational model, by analyzing them (as started in [19] for instance) or by generating them from specifications, and taking into account side-channel attacks.

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 the ANR ProSe project (decision ANR 2010-VERS-004).

References

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

[2] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, 153(1):27–39, Mar. 2006.

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

[4] R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal cryptographic proofs: the case of BBS. In *AVoCS'09*, volume 23 of *Electronic Communications of the EASST*, Sept. 2009.

[5] A. Armando et al. The AVISPA tool for automated validation of Internet security protocols and applications. In *CAV 2005*, volume 3576 of *LNCS*, pages 281–285. Springer, July 2005.

[6] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *CCS'09*, pages 66–78. ACM, Nov. 2009.

[7] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *CCS'06*, pages 370–379. ACM, Nov. 2006.

[8] M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *CCS'10*, pages 387–398. ACM Press, Oct. 2010.

[9] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *CSFW'04*, pages 204–218. IEEE, June 2004.

[10] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Transactions on Dependable and Secure Computing*, 2(2):109–123, Apr. 2005.

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

[12] G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *CCS'10*, pages 375–386. ACM Press, Oct. 2010.

[13] G. Barthe, B. Grégoire, S. Z. Béguelin, and Y. Lakhnech. Beyond provable security. Verifiable IND-CCA security of OAEP. In *CT-RSA 2011*, volume 6558 of *LNCS*, pages 180–196. Springer, Feb. 2011.

[14] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Aug. 2011.

[15] G. Barthe, B. Grégoire, and S. Zanella. Formal certification of code-based cryptographic proofs. In *POPL'09*, pages 90–101. ACM, Jan. 2009.

[16] S. Z. Béguelin, G. Barthe, S. Heraud, B. Grégoire, and D. Hedin. A machine-checked formalization of sigma-protocols. In *CSF'10*, pages 246–260. IEEE, July 2010.

[17] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT'00*, volume 1976 of *LNCS*, pages 531–545. Springer, Dec. 2000.

[18] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Eurocrypt 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, May 2006. Extended version available at `http://eprint.iacr.org/2004/331`.

[19] K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *CCS'08*, pages 459–468. ACM, Oct. 2008.

[20] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW-14*, pages 82–96. IEEE, June 2001.

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

[22] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *CSF'07*, pages 97–111. IEEE, July 2007. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[23] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008. Updated version available at `http://eprint.iacr.org/2005/401`.

[24] B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ASIACCS'08*, pages 87–99. ACM, Mar. 2008.

[25] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *CRYPTO 2006*, volume 4117 of *LNCS*, pages 537–554. Springer, Aug. 2006.

[26] S. Z. Béguelin, B. Grégoire, G. Barthe, and F. Olmedo. Formally certifying the security of digital signature schemes. In *IEEE Symposium on Security and Privacy*, pages 237–250. IEEE, May 2009.

[27] 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`.

[28] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *TCC'06*, volume 3876 of *LNCS*, pages 380–403. Springer, Mar. 2006. Extended version available at `http://eprint.iacr.org/2004/334`.

[29] H. Comon-Lundh and V. Cortier. Computational soundness of observational equivalence. In *CCS'08*, pages 109–118. ACM, Oct. 2008.

[30] V. Cortier, H. Hördegen, and B. Warinschi. Explicit randomness is not necessary when modeling probabilistic encryption. In *ICS 2006*, volume 186 of *ENTCS*, pages 49–65. Elsevier, Sept. 2006.

[31] 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.

[32] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *CCS'08*, pages 371–380. ACM, Oct. 2008.

[33] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Automated proofs for asymmetric encryption. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*, pages 300–321. Springer, 2010.

[34] J. Courant, C. Ene, and Y. Lakhnech. Computationally sound typing for non-interference: The case of deterministic encryption. In *FSTTCS'07*, volume 4855 of *LNCS*, pages 364–375. Springer, Dec. 2007.

[35] 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.

[36] A. Datta, A. Derek, J. C. Mitchell, and B. Warinschi. Computationally sound compositional logic for key exchange protocols. In *CSFW'06*, pages 321–334. IEEE, July 2006.

[37] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[38] 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`.

[39] 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.

[40] 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.

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

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

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

[44] P. Laud and I. Tšahhirov. A user interface for a game-based protocol verification tool. In *FAST2009*, volume 5983 of *LNCS*, pages 263–278. Springer, Nov. 2009.

[45] P. Laud and V. Vene. A type system for computationally secure information flow. In *FCT'05*, volume 3623 of *LNCS*, pages 365–377. Springer, Aug. 2005.

[46] 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.

[47] 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.

[48] J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1–3):118–164, Mar. 2006.

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

[50] D. Nowak. A framework for game-based security proofs. In *ICICS 2007*, volume 4861 of *LNCS*, pages 319–333. Springer, Dec. 2007.

[51] D. Nowak. On formal verification of arithmetic-based cryptographic primitives. In *ICISC 2008*, volume 5461 of *LNCS*, pages 368–382. Springer, Dec. 2008.

[52] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, Nov. 2004. Available at `http://eprint.iacr.org/2004/332`.

[53] G. Smith and R. Alpízar. Secure information flow with random assignment and encryption. In *FMSE'06*, pages 33–43, Nov. 2006.

[54] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *CSFW'06*, pages 153–166. IEEE, July 2006.

[55] I. Tšahhirov and P. Laud. Application of dependency graphs to security protocol analysis. In *TGC'07*, volume 4912 of *LNCS*, pages 294–311. Springer, Nov. 2007.

[56] 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.