Automated Security Proofs with Sequences of Games

Bruno Blanchet and David Pointcheval

CNRS, École Normale Supérieure, Paris – {blanchet,pointche}@di.ens.fr

Abstract. This paper presents the first automatic technique for proving not only protocols but also primitives in the exact security computational model. Automatic proofs of cryptographic protocols were up to now reserved to the Dolev-Yao model, which however makes quite strong assumptions on the primitives. On the other hand, with the proofs by reductions, in the complexity theoretic framework, more subtle security assumptions can be considered, but security analyses are manual. A process calculus is thus defined in order to take into account the probabilistic semantics of the computational model. It is already rich enough to describe all the usual security notions of both symmetric and asymmetric cryptography, as well as the basic computational assumptions. As an example, we illustrate the use of the new tool with the proof of a quite famous asymmetric primitive: UF-CMA of the FDH-signature scheme under the (trapdoor)-one-wayness of some permutations.

1 Introduction

There exist two main frameworks for analyzing the security of cryptographic protocols. The most famous one, among the cryptographic community, is the "provable security" in the reductionist sense [7, 37]: adversaries are probabilistic polynomial-time Turing machines which try to win a game, specific to the cryptographic primitive/protocol and to the security notion to be satisfied. The "computational" security is achieved by contradiction: if an adversary can win such an attack game with non-negligible probability, then a well-defined computational assumption is invalid (e.g., one-wayness, intractability of integer factoring, etc.) As a consequence, the actual security relies on the sole validity of the computational assumption. On the other hand, people from formal methods defined formal and abstract models, the so-called Dolev-Yao [19] framework, in order to be able to prove the security of cryptographic protocols too. However, these "formal" security proofs use the cryptographic primitives as ideal blackboxes. The main advantage of such a formalism is the automatic verifiability, or even provability, of the security, but under strong (and unfortunately unrealistic) assumptions. Our goal is to take the best of each framework, without the drawbacks, that is, to achieve automatic provability under classical (and realistic) computational assumptions.

The Computational Model. Since the seminal Diffie-Hellman's paper [18], complexity theory is tightly related to cryptography. Cryptographers indeed tried to use $\mathcal{NP}$-hard problems to build secure cryptosystems. Therefore, adversaries have been modeled by probabilistic polynomial-time Turing machines, and security notions have been defined by security games in which the adversary can interact with several oracles (which possibly embed some private information) and has to achieve a clear goal to win: for signature schemes, the adversary tries to forge a new valid message-signature pair, while it is able to ask for the signature of any message of its choice. Such an attack is called an existential forgery under chosen-message attacks [21, 22]. Similarly, for encryption, the adversary chooses two messages, and one of them is encrypted. Then the goal of the adversary is to guess which one has been encrypted [20], with a probability significantly better than one half. Again, several oracles may be available to the adversary, according to the kind of attack (chosen-plaintext and/or chosen-ciphertext attacks [34, 38, 25, 36]). One can see in these security notions that computation time and probabilities are of major importance: an unlimited adversary can always break them, with probability one; or in a shorter period of time, an adversary can guess the secret values, by chance, and thus win the attack game with possibly negligible but non-zero probability. Security proofs in this framework consist in showing that if such an adversary can win with significant probability, within reasonable time, then a well-defined problem can be broken with significant probability and within reasonable time too. Such an intractable problem and the reduction will quantify the security of the cryptographic protocol.

Indeed, in both symmetric and asymmetric scenarios, most security notions cannot be unconditionally guaranteed (whatever the computational power of the adversary). Therefore, security generally relies on a computational assumption: for instance, the existence of one-way functions, or permutations, possibly trapdoor. A one-way function is a function f which anyone can easily compute, but given $y = f(x)$ it is computationally intractable to recover x (or any pre-image of y). A one-way permutation is a bijective one-way function. For encryption, one would like the inversion to be possible for the recipient only: a trapdoor one-way permutation is a one-way permutation for which a secret information (the trapdoor) helps to invert the function on any point.

Given such objects, and thus computational assumptions about the intractability of the inversion (without trapdoors), we would like that security could be achieved without any additional assumptions. The only way to "formally" prove such a fact is by showing that an attacker against the cryptographic protocol can be used as a sub-part in an algorithm (the reduction) that can break the basic computational assumption.

Observational Equivalence and Sequence of Games. Initially, reductionist proofs consisted in presenting a reduction, and then proving that the view of the adversary provided by the reduction was (almost) indistinguishable to the view of the adversary during a real attack. Such an indistinguishability was quite technical and error-prone. Victor Shoup [43, 42, 44] suggested to prove it by small changes [10, 37], using a "sequence of games" (a.k.a. the game hopping technique) that the adversary plays, starting from the real attack game. Two consecutive games look either identical, or very close to each other in the view of the adversary, and thus involve a statistical distance, or a computational one. In the final game, the adversary has clearly no chance to win at all. Actually, the modifications of games can be seen as "rewriting rules" of the probability distributions of the variables involved in the games. They may consist of a simple renaming of some variables, and thus to perfectly identical distributions. They may introduce unlikely differences, and then the distributions are "statistically" indistinguishable. Finally, the rewriting rule may be true under a computational assumption only: then appears the computational indistinguishability.

In formal methods, games are replaced with processes using perfect primitives modeled by function symbols in an algebra of terms. "Observational equivalence" is a notion similar to indistinguishability: it expresses that two processes are perfectly indistinguishable by any adversary. The proof technique typically used for observational equivalence is however quite different from the one used for computational proofs. Indeed, in formal models, one has to exploit the absence of algebraic relations between function symbols in order to prove equivalence; in contrast to the computational setting, one does not have observational equivalence hypotheses (i.e. indistinguishability hypotheses), which specify security properties of primitives, and which can be combined in order to obtain a proof of the protocol.

Related Work. Following the seminal paper by Abadi and Rogaway [1], recent results [32, 16, 24] show the soundness of the Dolev-Yao model with respect to the computational model, which makes it possible to use Dolev-Yao provers in order to prove protocols in the computational model. However, these results have limitations, in particular in terms of allowed cryptographic primitives (they must satisfy strong security properties so that they correspond to Dolev-Yao style primitives), and they require some restrictions on protocols (such as the absence of key cycles).

Several frameworks exist for formalizing proofs of protocols in the computational model. Backes, Pfitzmann, and Waidner [4, 5, 2] have designed an abstract cryptographic library and shown its soundness with respect to computational primitives, under arbitrary active attacks. Backes and

Pfitzmann [3] relate the computational and formal notions of secrecy in the framework of this library. Recently, this framework has been used for a computationally-sound machine-checked proof of the Needham-Schroeder-Lowe protocol [45]. Canetti [14] introduced the notion of universal composability. With Herzog [15], 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 [11] for verifying protocols in this framework. Lincoln, Mateus, Mitchell, Mitchell, Ramanathan, Scedrov, and Teague [29–31, 39, 33] developed a probabilistic polynomial-time calculus for the analysis of cryptographic protocols. Datta *et al* [17] have designed a computationally sound logic that enables them to prove computational security properties using a logical deduction system. These frameworks can be used to prove security properties of protocols in the computational sense, but except for [15] which relies on a Dolev-Yao prover, they have not been automated up to now, as far as we know.

Laud [26] designed an automatic analysis for proving secrecy for protocols using shared-key encryption, with passive adversaries. He extended it [27] to active adversaries, but with only one session of the protocol. This work is the closest to ours. We extend it considerably by handling more primitives, a variable number of sessions, and evaluating the probability of an attack. More recently, he [28] designed a type system for proving security protocols in the computational model. This type system handles shared- and public-key encryption, with an unbounded number of sessions. This system relies on the Backes-Pfitzmann-Waidner library. Type inference has not been implemented yet, and we believe that it would not be obvious to automate.

Barthe, Cerderquist, and Tarento [6, 46] have formalized the generic model and the random oracle model in the interactive theorem prover Coq, and proved signature schemes in this framework. In contrast to our specialized prover, proofs in generic interactive theorem provers require a lot of human effort, in order to build a detailed enough proof for the theorem prover to check it.

Halevi [23] explains that implementing an automatic prover based on sequences of games would be useful, and suggests ideas in this direction, but does not actually implement one.

Our prover, which we describe in this paper, was previously presented in [12, 13], but in a more restricted way. It was indeed applied only to rather simple, Dolev-Yao-style protocols of the literature, such as the Needham-Schroeder public-key protocol. In this paper, we show that it can also be used for the proof of security of cryptographic primitives. [12, 13] considered only asymptotic proofs. In this paper, we have extended the prover for providing exact security proofs. We also extend it to the proof of authentication properties, while [12, 13] considered only secrecy properties. Finally, we also show how to model a random oracle.

Achievements. As in [12, 13], our goal is to fill the gap between the two usual techniques (computational and formal methods), but with a direct approach, in order to get the best of each: a computationally sound technique, which an automatic prover can apply. More precisely, we adapt the notion of observational equivalence so that it corresponds to the indistinguishability of games. To this aim, we also adapt the notion of processes: our processes run in time t and work with bit-strings. Furthermore, the process calculus has a probabilistic semantics, so that a measure can be defined on the distinguishability notion, or the observational equivalence, which extends the "perfect indistinguishability": the distance between two views of an adversary. This distance is due to the application of a transformation, which is purely syntactic. The transformations are rewriting rules, which yield a game either equivalent or almost equivalent under a "computational assumption". For example, we define a rewriting rule, which is true under the one-wayness of a specific function. The automatic prover tries to apply the rewriting rules until the winning event, which is executed in the original attack game when the adversary breaks the cryptographic protocol,

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

Fig. 1. Syntax of the process calculus

has totally disappeared: the adversary has a success probability 0. We can then upper-bound the success probability of the adversary in the initial game by the sum of all gaps.

Our prover also provides a manual mode in which the user can specify the main rewriting steps that the prover has to perform. This allows the system to prove protocols in situations in which the automatic proof strategy does not find the proof, and to direct the prover towards a specific proof, for instance a proof that yields a better reduction, since exact security is now dealt with.

2 A Calculus for Games

2.1 Description of the Calculus

In this section, we review the process calculus defined in [12, 13] in order to model games as done in computational security proofs [40, 41, 43]. The syntax is summarized in Figure 1. One should note that the main addition from previous models [33, 28] is the introduction of arrays, which allow us to formalize the random oracle model [8], but also the authenticity (unforgeability) in several cryptographic primitives, such as signatures, message authentication codes, but also encryption schemes. Arrays allow us to have full access to the whole memory state of the system, and replace lists often used in cryptographic proofs. For example, in the case of a random oracle, one generally stores the input and output of the random oracle in a list. In our calculus, they are stored in arrays.

Contrarily to [12, 13], we adopt the exact security framework [9, 35], instead of the asymptotic one. The cost of the reductions, and the probability loss will thus be precisely determined.

In this calculus, we denote by T types, which are subsets of $\mathit{bitstring}_\perp = \mathit{bitstring} \cup \{\perp\}$, where *bitstring* is the set of all bit-strings and $\perp$ is a special symbol. A type is said to be *fixed-length* when it is the set of all bit-strings of a certain length. A type T is said to be *large* when its cardinal is large enough so that we can consider collisions between elements of T chosen randomly with uniform probability quite unlikely, but still keeping track of the small probability. Such an information is useful for the strategy of the prover. The boolean type is predefined: $\mathit{bool} = \{\mathsf{true}, \mathsf{false}\}$, where $\mathsf{true} = 1$ and $\mathsf{false} = 0$.

The calculus also assumes a finite set of function symbols f. Each function symbol comes f with a type declaration $f : T_1 \times \ldots \times T_m \to T$. Then, the function symbol f corresponds to a function, also denoted f, from $T_1 \times \ldots \times T_m$ to T, such that $f(x_1, \ldots, x_m)$ is computable in time t_f, which is bounded by a function of the length of the inputs $x_1, \ldots, x_m$. Some predefined functions use the infix notation: $M = N$ for the equality test (taking two values of the same type T and returning a value of type *bool*), $M \wedge N$ for the boolean and (taking and returning values of type *bool*).

In this calculus, terms represent computations on bit-strings. Their syntax is given in Figure 1. The replication index i is an integer which serves in distinguishing different copies of a replicated process $!^{i \le n}$. Such a replication index is quite important, since it is typically used with arrays. (Each copy of the process often uses one cell of an array.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indexes $M_1, \ldots, M_m$ of the array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \ldots, M_m)$ returns the result of applying function f to $M_1, \ldots, M_m$.

In cryptographic environments, participants communicate to each other, and the adversary may have partial or total control of the network. Communications are thus performed over channels, which names are c. Channels may be either public or private. The maximum length of a message sent on channel c is bounded by maxlen(c). Longer messages are truncated.

In our calculus, we represent games by processes. 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; **newChannel** $c; Q$ creates a new private channel c and executes Q. We illustrate all other constructs of processes on the following example:

$$\begin{array}{l}
Q_0 = \mathit{start}(); \mathbf{new}\ r : \mathit{seed}; \mathbf{let}\ \mathit{pk} : \mathit{pkey} = \mathsf{pkgen}(r)\ \mathbf{in}\ \mathbf{let}\ \mathit{sk} : \mathit{skey} = \mathsf{skgen}(r)\ \mathbf{in}\ \overline{c0}\langle \mathit{pk} \rangle; \\
\qquad (!^{!_{25} \le qH} c4[!_{25}](x : \mathit{bitstring}); \overline{c5[!_{25}]}\langle \mathsf{hash}(x) \rangle \\
\qquad \mid !^{!_{26} \le qS} c1[!_{26}](m : \mathit{bitstring}); \overline{c2[!_{26}]}\langle \mathsf{mf}(\mathit{sk}, \mathsf{hash}(m)) \rangle \\
\qquad \mid c3(m' : \mathit{bitstring}, s : D); \\
\qquad\quad \mathbf{if}\ (\mathsf{f}(\mathit{pk}, s) = \mathsf{hash}(m'))\ \mathbf{then} \\
\qquad\quad \mathbf{find}\ u \le qS\ \mathbf{suchthat}\ \mathbf{defined}(m[u]) \wedge (m' = m[u])\ \mathbf{then}\ \overline{0}\ \mathbf{else}\ \mathbf{event}\ \mathsf{bad})
\end{array}$$

As we shall see in the next sections, this process comes from the definition of security of the Full-Domain Hash (FDH) signature scheme [8]. This process uses the function symbols hash, pkgen, skgen, f, and mf (such that $x \mapsto \mathsf{mf}(\mathit{sk}, x)$ is the inverse of the function $x \mapsto \mathsf{f}(\mathit{pk}, x)$), which will all be explained later in detail. This process first waits for an empty message on channel *start*. After receiving this message, it chooses a random number r in the type *seed*, with uniform probability, by the construct **new** $r : \mathit{seed}$. (*seed* must be a fixed-length type, because probabilistic bounded-time Turing machines can choose random numbers uniformly only in such types.) Then it computes $\mathsf{pkgen}(r)$, and stores the result in variable pk; $\mathsf{pkgen}(r)$ must be of type *pkey*. Similarly, it computes $\mathsf{skgen}(r)$ and stores the result in sk. Then it outputs the public key pk on channel $c0$ by $\overline{c0}\langle \mathit{pk} \rangle$.

After this output, the process waits for inputs on channels $c4$, $c1$, or $c3$. More precisely, it consists of the parallel composition of three processes, denoted $Q_1 \mid Q_2 \mid Q_3$. The first two of these processes are replicated: For instance, $!^{!_{25} \le qH} Q$ represents qH copies of Q in parallel, each with a different value of the replication index $!_{25} \in [1, qH]$.

The first replicated process inputs a message on channel $c4[!_{25}]$. In our calculus, a channel $c[M_1, \ldots, M_l]$ consists of both a channel name c and optionally a tuple of terms $M_1, \ldots, M_l$. Channel names c allow us to define private channels to which the adversary can never have access, by **newChannel** c. (This is useful in 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. As a consequence, channels allow us to model the usual communication model in computational security proofs: using different channels for each input and output allows the adversary to control the network, in an active and adaptive way. For instance, for the input on channel $c4[!_{25}]$, the adversary can choose which copy of the process receives the message by sending it on $c4[i]$ for the appropriate value of i. When the process receives a message on $c4[!_{25}]$, it stores the message in $x[!_{25}]$. Indeed, all variables defined under replications are arrays with indexes the indexes of replications above their definition, so that they use distinct array cells in different copies of the process. To lighten the notation, when x is under $!^{i_1 \leq n_1} \dots !^{i_m \leq n_m}$, we abbreviate $x[i_1, \dots, i_m]$ by x and more generally $x[i_1, \dots, i_k, u_1, \dots, u_{m'}]$ by $x[u_1, \dots, u_{m'}]$. Here, x stands for $x[!_{25}]$. After receiving this message, the process replies by sending $\mathsf{hash}(x)$ on channel $c5[!_{25}]$.

The second replicated process is similar: it receives on channel $c1[!_{26}]$ a bit-string m (or more precisely $m[!_{26}]$), and replies by sending on channel $c2[!_{26}]$ the message $\mathsf{mf}(sk, \mathsf{hash}(m))$. This process represents the signing oracle of FDH.

The third process receives a message m', s on channel $c3$ (where m' is any bit-string and s is of type D). After receiving this message, it tests whether $\mathsf{f}(pk, s) = \mathsf{hash}(m')$, as the verification algorithm of FDH would do. When the equality holds, it executes the **then** branch; otherwise, it executes the **else** branch which is here omitted. In this case, it yields control to another process by executing the output $\overline{\mathit{yield}}\langle\rangle; 0$, which sends an empty message on channel *yield*. This output is abbreviated $\overline{0}$; a branch **else** $\overline{0}$ or a trailing $\overline{0}$ can be omitted. When the test $\mathsf{f}(pk, s) = \mathsf{hash}(m')$ succeeds, the process performs an array lookup: it looks for an index u in $[1, qS]$ such that $m[u]$ is defined and $m' = m[u]$. If such an u is found, that is, m' has already been received on $c1$ by the second process (signing oracle), we simply yield control to another process by executing $\overline{0}$. Otherwise, we execute the event bad and the omitted output $\overline{0}$. Since the test $\mathsf{f}(pk, s) = \mathsf{hash}(m')$ corresponds to the verification of the FDH signature s for message m', the event bad is executed when the adversary has forged a signature s for message m'.

As already said, arrays are crucial in this calculus, and will help to model many properties which were hard to capture: the general syntax of an array lookup is as follows **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ **then** $P_j)$ **else** P, where $\widetilde{i}$ denotes a tuple $i_1, \dots, i_{m'}$. This process tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \dots, u_{jm_j}$ for which $M_{j1}, \dots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions **defined**$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ for each j and each value of $u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$ in $[1, n_{j1}] \times \dots \times [1, n_{jm_j}]$. If none of these conditions is 1, it executes P. Otherwise, it chooses randomly with (almost) uniform probability one j and one value of $u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is 1, and executes P_j. (When the number of possibilities is not a power of 2, a probabilistic bounded-time Turing machine cannot choose these values exactly with uniform probability, but it can choose them with a probability distribution as close as we wish to uniform.)

As detailed in [12, 13], we require some *well-formedness invariants* to guarantee that bit-strings are of their expected type and that arrays are used properly (that each cell of an array is assigned at most once during execution, and that variables are accessed only after being initialized). The formal semantics of the calculus can be found in [12].

2.2 Observational Equivalence

We denote by $\Pr[Q \rightsquigarrow \overline{c}\langle a\rangle]$ the probability that Q outputs the bit-string a on channel c after some reductions. We denote by $\Pr[Q \rightsquigarrow \mathcal{E}]$ the probability that the process Q executes exactly the sequence of events $\mathcal{E}$, in the order of $\mathcal{E}$. (A sequence of events is a sequence of values of the form $e(a_1, \dots, a_m)$ where e is an event symbol and $a_1, \dots, a_m$ are bit-strings.)

Definition 1 (Observational equivalence). Let Q and Q' be two processes, and V a set of variables. Assume that Q and Q' satisfy the well-formedness invariants and that the variables of V are defined in Q and Q', with the same types, and these types are fixed-length types.

A context is a process with a hole $[\,]$. An *evaluation context* is a context generated by the grammar $C ::= [\,] \quad C \mid Q \quad Q \mid C \quad \textbf{newChannel}\ c; C$.

An evaluation context is said to be *acceptable* for Q, Q', V if and only if C does not contain events, $\mathrm{var}(C) \cap (\mathrm{var}(Q) \cup \mathrm{var}(Q')) \subseteq V$, and $C[Q]$ satisfies the well-formedness invariants. (Then $C[Q']$ also satisfies these invariants.)

We say that Q and Q' are *observationally equivalent* with public variables V and probability p, written $Q \approx_p^V Q'$, when for all t, for all evaluation contexts C acceptable for Q, Q', V that run in time at most t, for all channels c, for all bit-strings a, $|\Pr[C[Q] \rightsquigarrow \overline{c}\langle a\rangle] - \Pr[C[Q'] \rightsquigarrow \overline{c}\langle a\rangle]| \leq p(t)$ and $\sum_{\mathcal{E}} |\Pr[C[Q] \rightsquigarrow \mathcal{E}] - \Pr[C[Q'] \rightsquigarrow \mathcal{E}]| \leq p(t)$.

In this definition, we use an evaluation context to represent an algorithm that tries to distinguish Q from Q'. This definition formalizes that the probability that an algorithm C running in time t distinguishes the games Q and Q' is at most $p(t)$. The set of variables V represents variables that the context C is allowed to access directly (using **find**). When V is empty, we write $Q \approx_p Q'$ instead of $Q \approx_p^V Q'$. We say that a context C runs in time t, when for all processes Q, the time spent in C in any trace of $C[Q]$ is at most t, ignoring the time spent in Q. (The runtime of a context is bounded, since the length of messages received by C on channels, the length of variables read by C using **find**, and the length of random numbers created by C are all bounded; the number of instructions executed by C is bounded; and the time of a function evaluation is bounded by function of the length of its arguments.)

Definition 2. We say that Q *executes* $e(a_1, \dots, a_m)$ *with probability at most* p if and only if for all t, for all evaluation contexts C acceptable for Q, Q, $\emptyset$ that run in time t, $\sum_{\mathcal{E}, e(a_1,\dots,a_m) \in \mathcal{E}} \Pr[C[Q] \rightsquigarrow \mathcal{E}] \leq p(t)$.

The above definitions allow us to perform proofs using sequences of indistinguishable games. The following lemma is straightforward:

Lemma 3. *1. $\approx_p^V$ is reflexive and symmetric.*

2. If $Q \approx_p^V Q'$ and $Q' \approx_{p'}^V Q''$, then $Q \approx_{p+p'}^V Q''$.

3. If Q executes $e(a_1, \dots, a_m)$ with probability at most p and $Q \approx_{p'} Q'$, then Q' executes $e(a_1, \dots, a_m)$ with probability at most $p + p'$.

4. If $Q \approx_p^V Q'$, C is an evaluation context acceptable for Q, Q', V that runs in time t_C, and $V' \subseteq V \cup (\mathrm{var}(C) \setminus (\mathrm{var}(Q) \cup \mathrm{var}(Q')))$, then $C[Q] \approx_{p'}^{V'} C[Q']$ where $p'(t) = p(t + t_C)$.

5. If Q executes $e(a_1, \dots, a_m)$ with probability at most p and C is an evaluation context acceptable for Q, Q, $\emptyset$ that runs in time t_C, then $C[Q]$ executes $e(a_1, \dots, a_m)$ with probability at most p' where $p'(t) = p(t + t_C)$.

Properties 2 and 3 are key to computing probabilities coming from a sequence of games. Indeed, our prover will start from a game Q_0 corresponding to the initial attack, and build a sequence of observationally equivalent games $Q_0 \approx_{p_1}^V Q_1 \approx_{p_2}^V \dots \approx_{p_m}^V Q_m$. By Property 2, we conclude that $Q_0 \approx_{p_1+\dots+p_m}^V Q_m$. By Property 3, we can bound the probability that Q_0 executes an event from the probability that Q_m executes this event.

The elementary transformations used to build each game from the previous one can in particular come from the security of a cryptographic primitive. This security property is typically specified as an observational equivalence $L \approx_p R$. To use it to transform a game Q, the prover finds a context C such that $Q \approx_0^V C[L]$ by purely syntactic transformations, and builds a game Q' such that $Q' \approx_0^V C[R]$ by purely syntactic transformations. By Property 4, we have $C[L] \approx_{p'}^V C[R]$, so $Q \approx_{p'}^V Q'$.

3 Characterization of One-wayness and Security of Signatures

Before introducing the assumption (one-wayness) and the security notion (unforgeability) to achieve, let us present a few simple lemmas which will be used to show observational equivalences given as hypothesis to our prover.

3.1 Preliminary Lemmas

In lemmas below, the notation $Q\{P'/P\}$ means that, in the process Q, we substitute any occurrence of P by P'. The first lemma is the so-called "Shoup's lemma" [43], where we consider a process Q which contains an event e without argument (e.g. a "bad" event).

Lemma 4. *If Q executes e with probability at most p, then $Q\{P'/\textbf{event } e.P\} \approx_p Q\{P''/\textbf{event } e.P\}$.*

Lemma 5. *If Q executes e with probability at most p, then $Q\{\textbf{event } e.P'/\textbf{event } e.P\}$ executes e with probability at most p.*

We denote by $n \times Q$ the process obtained by adding $!^{i \leq n}$ in front of Q and by adding the index i at the beginning of each sequence of array indexes and each sequence of indexes of channels in Q, for some fresh replication index i. The process $n \times Q$ encodes n independent copies of Q. The following lemma can be proved by choosing randomly the copy of Q that executes e, and simulating all other copies of Q.

Lemma 6. *If Q executes e with probability at most p and Q runs in time t_Q, then $n \times Q$ executes e with probability at most p' where $p'(t) = n \times p(t + (n-1)t_Q)$.*

3.2 Trapdoor One-Way Permutations

Most cryptographic protocols rely on the existence of trapdoor one-way permutations. They are families of permutations, which are easy to compute, but hard to invert, unless one has a trapdoor.

The Computational Model. A family of permutations $\mathcal{P}$ onto a set D is defined by the three following algorithms:

- The *key generation algorithm* kgen (which can be split in two sub-algorithms pkgen and skgen). On input a seed r, the algorithm kgen produces a pair (pk, sk) of matching public and secret keys. The public key pk specifies the actual permutation f_{pk} onto the domain D.
- The *evaluation algorithm* f. Given a public key pk and a value $x \in D$, it outputs $y = f_{pk}(x)$.
- The *inversion algorithm* mf. Given an element y, and the trapdoor sk, mf outputs the unique pre-image x of y with respect to f_{pk}.

The above properties simply require the algorithms to be efficient. The "one-wayness" property is more intricate, since it claims the "non-existence" of some efficient algorithm: one wants that the success probability of any adversary $\mathcal{A}$ with a reasonable time is small, where

$$\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(\mathcal{A}) = \Pr\left[r \overset{R}{\leftarrow} seed, (pk, sk) \leftarrow \mathsf{kgen}(r), x \overset{R}{\leftarrow} D, y \leftarrow \mathsf{f}(pk, x), x' \leftarrow \mathcal{A}(pk, y) : x = x'\right].$$

Eventually, we denote by $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t)$ the maximal success probability an adversary can get within time t.

Syntactic Rules. Let *seed* be a large, fixed-length type, *pkey*, *skey*, and D the types of public keys, secret keys, and the domain of the permutations respectively. A family of trapdoor one-way permutations can then be defined as a set of four function symbols: $\mathsf{skgen} : \mathit{seed} \to \mathit{skey}$ generates secret keys; $\mathsf{pkgen} : \mathit{seed} \to \mathit{pkey}$ generates public keys; $\mathsf{f} : \mathit{pkey} \times D \to D$ and $\mathsf{mf} : \mathit{skey} \times D \to D$, such that, for each pk, $x \mapsto \mathsf{f}(pk, x)$ is a permutation of D, whose inverse permutation is $x \mapsto \mathsf{mf}(sk, x)$ when $pk = \mathsf{pkgen}(r)$ and $sk = \mathsf{skgen}(r)$.

The one-wayness property can be formalized in our calculus by requiring that LR executes **event** bad with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t)$ in the presence of a context that runs in time t, where

$$\begin{aligned} LR = {}& c(); \mathbf{new}\ r_0 : \mathit{seed}; \mathbf{new}\ x_0 : D; \overline{c}\langle \mathsf{pkgen}(r_0), \mathsf{f}(\mathsf{pkgen}(r_0), x_0)\rangle; \\ & c(x' : D); \mathbf{if}\ x' = x_0\ \mathbf{then\ event}\ \mathsf{bad}; \overline{c}\langle\rangle\ \mathbf{else}\ \overline{c}\langle\rangle \end{aligned}$$

Indeed, the event bad is executed when the adversary, given the public key $\mathsf{pkgen}(r_0)$ and the image of some x_0 by f, manages to find x_0 (without having the trapdoor).

In order to use the one-wayness property in proofs of protocols, our prover needs a more general formulation of one-wayness, using "observationally equivalent" processes. We thus define two processes which are actually equivalent unless LR executes **event** bad. Let $\mathsf{pkgen}' : \mathit{seed} \to \mathit{pkey}$ and $\mathsf{f}' : \mathit{pkey} \times D \to D$ such that the functions associated to the primed symbols $\mathsf{pkgen}', \mathsf{f}'$ are equal to the functions associated to their corresponding unprimed symbol $\mathsf{pkgen}, \mathsf{f}$. We have the following equivalence

$$\begin{aligned} & !^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} \mathbf{new}\ r : \mathit{seed}; (!^{i_0 \leq n_0} () \to \mathsf{pkgen}(r), \\ & \qquad !^{i_{\mathrm{f}} \leq n_{\mathrm{f}}} \mathbf{new}\ x : D; (!^{i_1 \leq n_1} () \to \mathsf{f}(\mathsf{pkgen}(r), x), \\ & \qquad\quad !^{i_2 \leq n_2} (x' : D) \to x' = x, \\ & \qquad\quad !^{i_3 \leq n_3} () \to x)) \\ \approx_{p^{\mathsf{ow}}} & !^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} \mathbf{new}\ r : \mathit{seed}; (!^{i_0 \leq n_0} () \to \mathsf{pkgen}'(r), \\ & \qquad !^{i_{\mathrm{f}} \leq n_{\mathrm{f}}} \mathbf{new}\ x : D; (!^{i_1 \leq n_1} () \to \mathsf{f}'(\mathsf{pkgen}'(r), x), \\ & \qquad\quad !^{i_2 \leq n_2} (x' : D) \to \mathbf{find}\ u \leq n_3\ \mathbf{suchthat\ defined}(k[u]) \wedge \mathsf{true} \\ & \qquad\qquad\qquad \mathbf{then}\ x' = x\ \mathbf{else}\ \mathsf{false}, \\ & \qquad\quad !^{i_3 \leq n_3} () \to \mathbf{let}\ k : \mathit{bitstring} = \mathsf{mark}\ \mathbf{in}\ x)) \end{aligned} \tag{1}$$

where $p^{\mathsf{ow}}(t) = n_{\mathrm{k}} \times n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + (n_{\mathrm{k}} n_{\mathrm{f}} - 1)t_{\mathsf{f}} + (n_{\mathrm{k}} - 1)t_{\mathsf{pkgen}})$, t_{f} is the time of one evaluation of f, and t_{pkgen} is the time of one evaluation of pkgen. This equivalence uses a different syntax from processes, in order to represent functions: for example, $(x' : D) \to x' = x$ represents a function that takes as input $x' \in D$ and returns a boolean equal to true when $x' = x$. As formalized in [12, 13], these functions can be translated into processes that input their arguments on a channel and send the reply back to this channel. Using this specialized syntax allows us to model many equivalences that define cryptographic primitives, and it simplifies considerably the transformation of processes compared to using the general syntax of processes. (In order to use an equivalence $L \approx_p R$, we need to recognize processes that can easily be transformed into $C[L]$ for some context C, to transform them into $C[R]$. This is rather easy to do with functions: we just need to recognize terms that occur as a result of these functions. That would be much more difficult with general processes.)

In this equivalence, we consider n_{k} keys $\mathsf{pkgen}(r[i_{\mathrm{k}}])$ instead of a single one, and n_{f} antecedents of f for each key, $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$. The first function publishes the public key $\mathsf{pkgen}(r[i_{\mathrm{k}}])$. The second group of functions first picks a new $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$, and then makes available three functions: the first one returns the image of $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ by f, the second one returns true when it receives $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ as argument, and the third one returns $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ itself. The one-wayness property guarantees that when the third function has not been called, the adversary has little chance of finding $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$, so the second function returns false. Therefore, we can replace the left-hand side of the equivalence with its right-hand side, in which the third function records that it has been called by defining k, and the second function always returns false when no k is defined, that is, when the third function has not been called. We replace pkgen and f with pkgen' and f' in the right-hand side just to prevent repeated applications of the transformation with the same keys, which would lead to an infinite loop. This equivalence is proved in Appendix A.

Since $x \mapsto \mathsf{f}(\mathsf{pkgen}(r), x)$ and $x \mapsto \mathsf{mf}(\mathsf{skgen}(r), x)$ are inverse permutations, we have:

$$\forall r : \mathit{seed}, \forall x : D, \mathsf{mf}(\mathsf{skgen}(r), \mathsf{f}(\mathsf{pkgen}(r), x)) = x \tag{2}$$

Since $x \mapsto \mathsf{f}(\mathit{pk}, x)$ is injective, $\mathsf{f}(\mathit{pk}, x) = \mathsf{f}(\mathit{pk}, x')$ if and only if $x = x'$:

$$\forall \mathit{pk} : \mathit{pkey}, \forall x : D, \forall x' : D, (\mathsf{f}(\mathit{pk}, x) = \mathsf{f}(\mathit{pk}, x')) = (x = x') \tag{3}$$

Since $x \mapsto \mathsf{f}(\mathit{pk}, x)$ is a permutation, when x is a uniformly distributed random number, we can replace x with $\mathsf{f}(\mathit{pk}, x)$ everywhere, without changing the probability distribution. In order to enable automatic proof, we give a more restricted formulation of this result:

$$\begin{aligned}
&!^{i_{\mathrm{k}} \leq n_{\mathrm{k}}}\ \mathbf{new}\ r : \mathit{seed}; (!^{i_0 \leq n_0}\ () \rightarrow \mathsf{pkgen}(r),\\
&\qquad\qquad !^{i_{\mathrm{f}} \leq n_{\mathrm{f}}}\ \mathbf{new}\ x : D; (!^{i_1 \leq n_1}\ () \rightarrow \mathsf{mf}(\mathsf{skgen}(r), x), !^{i_2 \leq n_2}\ () \rightarrow x))\\
\approx_0\ &!^{i_{\mathrm{k}} \leq n_{\mathrm{k}}}\ \mathbf{new}\ r : \mathit{seed}; (!^{i_0 \leq n_0}\ () \rightarrow \mathsf{pkgen}(r),\\
&\qquad\qquad !^{i_{\mathrm{f}} \leq n_{\mathrm{f}}}\ \mathbf{new}\ x : D; (!^{i_1 \leq n_1}\ () \rightarrow x, !^{i_2 \leq n_2}\ () \rightarrow \mathsf{f}(\mathsf{pkgen}(r), x)))
\end{aligned} \tag{4}$$

which allows to perform the previous replacement only when x is used in calls to $\mathsf{mf}(\mathsf{skgen}(r), x)$, where r is a random number such that r occurs only in $\mathsf{pkgen}(r)$ and $\mathsf{mf}(\mathsf{skgen}(r), x)$ for some random numbers x.

3.3 Signatures

The Computational Model. A signature scheme $\mathsf{S} = (\mathsf{kgen}, \mathsf{sign}, \mathsf{verify})$ is defined by:

- The *key generation algorithm* kgen (which can be split in two sub-algorithms pkgen and skgen). On input a random seed r, the algorithm kgen produces a pair $(\mathit{pk}, \mathit{sk})$ of matching keys.
- The *signing algorithm* sign. Given a message m and a secret key sk, sign produces a signature σ. For sake of clarity, we restrict ourselves to the deterministic case.
- The *verification algorithm* verify. Given a signature σ, a message m, and a public key pk, verify tests whether σ is a valid signature of m with respect to pk.

As usual, security notions are defined by the goals the adversary wants to achieve, and its means [21, 22]. The highest security level is defined by the simplest goal: providing a new message-signature pair. This is called *existential forgery*. The corresponding security level is called *(existential) unforgeability* (UF).

On the other hand, various means can be made available to the adversary, helping it into its forgery. The strongest is definitely the *adaptive chosen-message attack* (CMA), where the attacker can ask the signer to sign any message of its choice, in an adaptive way: it can adapt its queries according to previous answers. Of course, in its answer, there is the natural restriction that the returned message has not been asked to the signing oracle.

When one designs a signature scheme, one wants to computationally rule out existential forgeries under adaptive chosen-message attacks. More formally, one wants that the success probability of any adversary $\mathcal{A}$ with a reasonable time is small, where

$$\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(\mathcal{A}) = \Pr\left[r \overset{R}{\leftarrow} \mathit{seed}, (\mathit{pk}, \mathit{sk}) \leftarrow \mathsf{kgen}(r), (m, \sigma) \leftarrow \mathcal{A}^{\mathsf{sign}(\cdot, \mathit{sk})}(\mathit{pk}) : \mathsf{verify}(m, \mathit{pk}, \sigma) = 1 \right].$$

As above, we denote by $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, l, t)$ the maximal success probability an adversary can get within time t, after at most n_{s} queries to the signing oracle, where the maximum length of all messages in queries is l.

Syntactic Rules. Let *seed* be a large, fixed-length type, *pkey*, *skey*, and *signature* the types of public keys, secret keys, and signatures respectively. A signature scheme is defined as a set of 4 function symbols: $\mathsf{skgen} : \mathit{seed} \rightarrow \mathit{skey}$ generates secret keys; $\mathsf{pkgen} : \mathit{seed} \rightarrow \mathit{pkey}$ generates public keys; $\mathsf{sign} : \mathit{bitstring} \times \mathit{skey} \rightarrow \mathit{signature}$ generates signatures; and $\mathsf{verify} : \mathit{bitstring} \times \mathit{pkey} \times \mathit{signature} \rightarrow \mathit{bool}$ verifies signatures.

The signature verification succeeds for signatures generated by sign, that is,

$$\forall m : \mathit{bitstring}, \forall r : \mathit{seed}, \mathsf{verify}(m, \mathsf{pkgen}(r), \mathsf{sign}(m, \mathsf{skgen}(r))) = \mathsf{true}$$

According to the previous definition of $\mathsf{UF} - \mathsf{CMA}$, the following process LR executes **event** bad with probability at most $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, l, t)$ in the presence of a context that runs in time t, where

$$\begin{aligned} LR = {} & \mathit{start}(); \textbf{new } r : \mathit{seed}; \textbf{let } \mathit{pk} : \mathit{pkey} = \mathsf{pkgen}(r) \textbf{ in let } \mathit{sk} : \mathit{skey} = \mathsf{skgen}(r) \textbf{ in } \overline{c_0}\langle \mathit{pk} \rangle; \\ & (!^{i_{\mathrm{s}} \leq n_{\mathrm{s}}} c_1[i](m : \mathit{bitstring}); \overline{c_2[i]}\langle \mathsf{sign}(m, \mathit{sk}) \rangle \\ & \mid c_3(m' : \mathit{bitstring}, s : \mathit{signature}); \textbf{if } \mathsf{verify}(m', \mathit{pk}, s) \textbf{ then} \\ & \quad \textbf{find } u_{\mathrm{s}} \leq n_{\mathrm{s}} \textbf{ suchthat defined}(m[u_{\mathrm{s}}]) \wedge m' = m[u_{\mathrm{s}}] \textbf{ then } \overline{0} \textbf{ else event } \mathsf{bad}) \end{aligned} \tag{5}$$

and l is the maximum length of m and m'. This is indeed clear since **event** bad is raised if a signature is accepted (by the verification algorithm), while it has not been generated by the signing algorithm. This definition is the one used when one wants to prove that a signature scheme is secure using our prover. However, when one wants to prove the security of a protocol using a signature scheme, such a formal definition is not enough for our prover, which needs "observationally equivalent" processes. So, similarly to what we have done for one-wayness, we present in Appendix C a definition using observationally equivalent processes, and show its soundness from (5).

4 Example: FDH Signature

The Full-Domain Hash (FDH) signature scheme [8] is defined as follows: Let $\mathsf{pkgen}, \mathsf{skgen}, \mathsf{f}, \mathsf{mf}$ define a family of trapdoor one-way permutations. Let hash be a hash function, in the random oracle model. The FDH signature scheme uses the functions pkgen and skgen as key-generation functions, the signing algorithm is $\mathsf{sign}(m, \mathit{sk}) = \mathsf{mf}(\mathit{sk}, \mathsf{hash}(m))$, and the verification algorithm is $\mathsf{verify}(m', \mathit{pk}, s) = (\mathsf{f}(\mathit{pk}, s) = \mathsf{hash}(m'))$. In this section, we explain how our automatic prover finds the well-known bound for $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}$ for the FDH signature scheme.

The input given to the prover contains two parts. First, it contains the definition of security of primitives used to build the FDH scheme, that is, the definition of one-way trapdoor permutations (1), (2), (3), and (4) as detailed in Section 3.2 and the formalization of a hash function in the random oracle model, given as follows:

$$\begin{aligned} & !^{i_{\mathrm{h}} \leq n_{\mathrm{h}}} \ (x : \mathit{bitstring}) \rightarrow \mathsf{hash}(x) \ [\mathit{all}] \\ \approx_0 \ & !^{i_{\mathrm{h}} \leq n_{\mathrm{h}}} \ (x : \mathit{bitstring}) \rightarrow \textbf{find } u \leq n_{\mathrm{h}} \textbf{ suchthat defined}(x[u], r[u]) \wedge (x = x[u]) \textbf{ then } r[u] \\ & \qquad \textbf{else new } r : D; r \end{aligned} \tag{6}$$

This equivalence expresses that we can replace a call to a hash function with a random oracle, that is, a function that returns a fresh random number when it is called with a new argument, and the previously returned result when it is called with the same argument as in a previous call. Such a random oracle is implemented in our calculus by a lookup in the array x of the arguments of hash. When a u such that $x[u]$, $r[u]$ are defined and $x = x[u]$ is found, hash has already been called with x, at call number u, so we return the result of that call, $r[u]$. Otherwise, we create a fresh random number r. (The indication $[all]$ on the first line of (6) instructs the prover to replace all occurrences of hash in the game.)

Second, the input file contains as initial game the process Q_0 of Section 2.1. This game corresponds to the definition of security of the FDH signature scheme (5) as detailed in Section 3.3. An important remark is that we need to add to the standard definition of security of a signature scheme the hash oracle. This is necessary so that, after transformation of hash into a random oracle, the adversary can still call the hash oracle. (The adversary does not have access to the arrays that encode the values of the random oracle.) Our goal is to bound the probability $p(t)$ that **event** bad is executed in this game in the presence of a context that runs in time t: $p(t) = \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(qS, l, t + t_H) \geq \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(qS, l, t)$ where t_H is the total time spent in the hash oracle and l is the maximum length of m and m'.

Given this input, our prover automatically produces a proof that this game executes **event** bad with probability $p(t) \leq (qH + qS + 1)\mathsf{Succ}_{\mathcal{P}}^{\mathsf{ow}}(t + t' + (qH + qS)t_{\mathsf{f}})$ where t' is the runtime of the context put around the equivalence (1) in the proof. (The prover displays the corresponding game.) The time t' can be evaluated by manual inspection. We obtain $t' = (qH + qS + 1)t_{\mathbf{find}}(qH + qS + 1, l) + t_{\mathbf{find}}(qS, l)$ where l is the maximum length of a bit-string in m, m', or x and $t_{\mathbf{find}}(n, l)$ is the time of a **find** that looks up a bit-string of length at most l in an array of at most n cells, so $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(qS, l, t) \leq (qH + qS + 1)\mathsf{Succ}_{\mathcal{P}}^{\mathsf{ow}}(t + (qH + qS + 1)t_{\mathbf{find}}(qH + qS + 1, l) + t_{\mathbf{find}}(qS, l) + (qH + qS)t_{\mathsf{f}})$. If we ignore the time of **find**, we obtain the usual upper-bound [9] $(qH + qS + 1)\mathsf{Succ}_{\mathcal{P}}^{\mathsf{ow}}(t + (qH + qS)t_{\mathsf{f}})$. The prover also outputs the sequence of games that leads to this proof, and a succinct explanation of the transformation performed between consecutive games of the sequence. This proof is explained in Appendix B. The input and output of the prover, as well as the prover itself, are available at `http://www.di.ens.fr/~blanchet/cryptoc/FDH/`; the runtime of the prover on this example is 14 ms on a Pentium M 1.8 GHz.

In order to build this proof, the prover uses the following strategy. It tries to apply all observational equivalences it has as hypotheses, that is here, (1), (4), and (6). When one of these equivalences can be applied, it transforms the game accordingly, by replacing the left-hand side with the right-hand side of the equivalence. When they fail, these equivalences may suggest syntactic transformations to apply in order to enable them. In this case, the prover applies the suggested syntactic transformation and retries the initial transformation. For example, if the game contains the term $\mathsf{mf}(sk, y)$ while (4) expects $\mathsf{mf}(\mathsf{skgen}(r), y)$, the prover will first expand the assignment that defines sk so that sk is substituted with its value, possibly yielding the desired term. After each application of an observational equivalence, the obtained game is simplified as much as possible.

5 Conclusion

Besides proving the security of many protocols, we have also used our prover for proving other cryptographic schemes. For example, using a few manual indications from the user (5 instructions), our prover can also show that the basic Bellare-Rogaway construction [8] without redundancy (*i.e.* $\mathcal{E}(m, r) = f(r) \| G(r) \oplus m$) is IND-CPA. With a few extensions concerning the simplification of games, which we plan to implement in the near future, it could also show that the enhanced version with redundancy (*i.e.* $\mathcal{E}(m, r) = f(r) \| G(r) \oplus m \| H(m, r)$) is IND-CCA2.

Acknowledgments

We thank Jacques Stern for initiating our collaboration on this topic. This work was partly supported by ARA SSIA Formacrypt.

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. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *CSFW'04*. IEEE, June 2004.
3. M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *26th IEEE Symposium on Security and Privacy*, pages 171–182. IEEE, May 2005.
4. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *CCS'03*, pages 220–230. ACM, Oct. 2003.
5. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *ESORICS'03*, LNCS 2808, pages 271–290. Springer, Oct. 2003.
6. G. Barthe, J. Cederquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In *IJCAR'04*, LNCS 3097, pages 385–399. Springer, July 2004.
7. M. Bellare. Practice-Oriented Provable Security. In *ISW '97*, LNCS 1396. Springer-Verlag, Berlin, 1997.
8. M. Bellare and P. Rogaway. Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols. In *Proc. of the 1st CCS*, pages 62–73. ACM Press, New York, 1993.
9. M. Bellare and P. Rogaway. The Exact Security of Digital Signatures – How to Sign with RSA and Rabin. In *Eurocrypt '96*, LNCS 1070, pages 399–416. Springer-Verlag, Berlin, 1996.
10. M. Bellare and P. Rogaway. The Game-Playing Technique and its Application to Triple Encryption, 2004. Cryptology ePrint Archive 2004/331.
11. B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, May 2004.
12. B. Blanchet. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, Nov. 2005. Available at `http://eprint.iacr.org/2005/401`.
13. B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, May 2006. To appear.
14. 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`.
15. R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). Cryptology ePrint Archive, Report 2004/334, 2004. Available at `http://eprint.iacr.org/2004/334`.
16. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP'05*, LNCS 3444, pages 157–171. Springer, Apr. 2005.
17. A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *ICALP'05*, LNCS 3580, pages 16–29. Springer, July 2005.
18. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, November 1976.
19. D. Dolev and A. C. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
20. S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
21. S. Goldwasser, S. Micali, and R. Rivest. A "Paradoxical" Solution to the Signature Problem. In *Proc. of the 25th FOCS*, pages 441–448. IEEE, New York, 1984.
22. S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptative Chosen-Message Attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.
23. 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`.
24. R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In *ESOP'05*, LNCS 3444, pages 172–185. Springer, Apr. 2005.
25. J. Katz and M. Yung. Complete Characterization of Security Notions for Probabilistic Private-Key Encryption. In *Proc. of the 32nd STOC*. ACM Press, New York, 2000.

26. P. Laud. Handling encryption in an analysis for secure information flow. In *ESOP'03*, LNCS 2618, pages 159–173. Springer, Apr. 2003.
27. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy*, pages 71–85, May 2004.
28. P. Laud. Secrecy types for a simulatable cryptographic library. In *CCS'05*, pages 26–35. ACM, Nov. 2005.
29. P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS'98*, pages 112–121, Nov. 1998.
30. P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *FM'99*, LNCS 1708, pages 776–793. Springer, Sept. 1999.
31. P. Mateus, J. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In *CONCUR 2003*, LNCS 2761, pages 327–349. Springer, Sept. 2003.
32. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *TCC'04*, LNCS 2951, pages 133–151. Springer, Feb. 2004.
33. J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 2006. To appear.
34. M. Naor and M. Yung. Universal One-Way Hash Functions and Their Cryptographic Applications. In *Proc. of the 21st STOC*, pages 33–43. ACM Press, New York, 1989.
35. K. Ohta and T. Okamoto. On Concrete Security Treatment of Signatures Derived from Identification. In *Crypto '98*, LNCS 1462, pages 354–369. Springer-Verlag, Berlin, 1998.
36. D. H. Phan and D. Pointcheval. About the Security of Ciphers (Semantic Security and Pseudo-Random Permutations). In *11th Annual Workshop on Selected Areas in Cryptography (SAC '04)*, LNCS 3357, pages 185–200. Springer-Verlag, Berlin, 2004.
37. D. Pointcheval. *Advanced Course on Contemporary Cryptology*, chapter Provable Security for Public-Key Schemes, pages 133–189. Advanced Courses CRM Barcelona. Birkhuser Publishers, Basel, juin 2005. ISBN: 3-7643-7294-X (248 pages).
38. C. Rackoff and D. R. Simon. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. In *Crypto '91*, LNCS 576, pages 433–444. Springer-Verlag, Berlin, 1992.
39. A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *FOSSACS'04*, LNCS 2987, pages 468–483. Springer, Mar. 2004.
40. V. Shoup. Practical Threshold Signatures. In *Eurocrypt '00*, LNCS 1807, pages 207–220. Springer-Verlag, Berlin, 2000.
41. V. Shoup. Using Hash Functions as a Hedge against Chosen Ciphertext Attack. In *Eurocrypt '00*, LNCS 1807, pages 275–288. Springer-Verlag, Berlin, 2000.
42. V. Shoup. A Proposal for an ISO Standard for Public-Key Encryption, december 2001. ISO/IEC JTC 1/SC27.
43. V. Shoup. OAEP Reconsidered. *Journal of Cryptology*, 15(4):223–249, September 2002.
44. V. Shoup. Sequences of games: a tool for taming complexity in security proofs, 2004. Cryptology ePrint Archive 2004/332.
45. C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. Unpublished manuscript, Feb. 2006.
46. S. Tarento. Machine-checked security proofs of cryptographic signature schemes. In *ESORICS'05*, LNCS 3679, pages 140–158. Springer, Sept. 2005.

A Proof of the Definition of One-wayness as an Equivalence

Proof (of (1)*).* We denote by L the left-hand side of (1) and by R its right-hand side. We denote by $[\![L]\!]$ and $[\![R]\!]$ the translations of L and R into processes, respectively. We have:

$$\begin{aligned}[\![L]\!] = {}&!^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} c_{\mathrm{k}}[i_{\mathrm{k}}]() ; \mathbf{new}\ r : \mathit{seed} ; \overline{c_{\mathrm{k}}[i_{\mathrm{k}}]}\langle\rangle ; (!^{i_0 \leq n_0} c_0[i_{\mathrm{k}}, i_0]() ; \overline{c_0[i_{\mathrm{k}}, i_0]}\langle \mathsf{pkgen}(r) \rangle \\ &\quad\ \mid !^{i_{\mathrm{f}} \leq n_{\mathrm{f}}} c_{\mathrm{f}}[i_{\mathrm{k}}, i_{\mathrm{f}}]() ; \mathbf{new}\ x : D ; \overline{c_{\mathrm{f}}[i_{\mathrm{k}}, i_{\mathrm{f}}]}\langle\rangle ; (!^{i_1 \leq n_1} c_1[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]() ; \overline{c_1[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]}\langle \mathsf{f}(\mathsf{pkgen}(r), x) \rangle \\ &\qquad \mid !^{i_2 \leq n_2} c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2](x' : D) ; \overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x \rangle \\ &\qquad \mid !^{i_3 \leq n_3} c_3[i_{\mathrm{k}}, i_{\mathrm{f}}, i_3]() ; \overline{c_3[i_{\mathrm{k}}, i_{\mathrm{f}}, i_3]}\langle x \rangle))\end{aligned}$$

$$\begin{aligned}
\llbracket R\rrbracket = {} & !^{i_{\mathrm{k}}\le n_{\mathrm{k}}} c_{\mathrm{k}}[i_{\mathrm{k}}](); \mathbf{new}\ r : \mathit{seed}; \overline{c_{\mathrm{k}}[i_{\mathrm{k}}]}\langle\rangle; (!^{i_0\le n_0} c_0[i_{\mathrm{k}}, i_0](); \overline{c_0[i_{\mathrm{k}}, i_0]}\langle \mathsf{pkgen}'(r)\rangle \\
& \qquad \mid !^{i_{\mathrm{f}}\le n_{\mathrm{f}}} c_{\mathrm{f}}[i_{\mathrm{k}}, i_{\mathrm{f}}](); \mathbf{new}\ x : D; \overline{c_{\mathrm{f}}[i_{\mathrm{k}}, i_{\mathrm{f}}]}\langle\rangle; (!^{i_1\le n_1} c_1[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1](); \overline{c_1[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]}\langle \mathsf{f}'(\mathsf{pkgen}'(r), x)\rangle \\
& \qquad\quad \mid !^{i_2\le n_2} c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2](x' : D); \\
& \qquad\qquad \mathbf{find}\ u \le n_3\ \mathbf{suchthat}\ \mathbf{defined}(k[u]) \wedge \mathsf{true}\ \mathbf{then}\ \overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x\rangle\ \mathbf{else} \\
& \qquad\qquad \overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false}\rangle \\
& \qquad\quad \mid !^{i_3\le n_3} c_3[i_{\mathrm{k}}, i_{\mathrm{f}}, i_3](); \mathbf{let}\ k : \mathit{bitstring} = \mathsf{mark}\ \mathbf{in}\ \overline{c_3[i_{\mathrm{k}}, i_{\mathrm{f}}, i_3]}\langle x\rangle))
\end{aligned}$$

We show that $\llbracket L\rrbracket \approx_{p^{\mathsf{ow}}} \llbracket R\rrbracket$, which proves (1).

Let

$$\begin{aligned}
LR' = {} & c_{\mathrm{k}}(); \mathbf{new}\ r : \mathit{seed}; \mathbf{let}\ pk : \mathit{pkey} = \mathsf{pkgen}(r)\ \mathbf{in}\ \overline{c_{\mathrm{k}}}\langle\rangle; (!^{i_0\le n_0} c_0[i_0](); \overline{c_0[i_0]}\langle pk\rangle \\
& \qquad \mid !^{i_{\mathrm{f}}\le n_{\mathrm{f}}} c_{\mathrm{f}}[i_{\mathrm{f}}](); \mathbf{new}\ x : D; \mathbf{let}\ y : D = \mathsf{f}(pk, x)\ \mathbf{in}\ \overline{c_{\mathrm{f}}[i_{\mathrm{f}}]}\langle\rangle; (!^{i_1\le n_1} c_1[i_{\mathrm{f}}, i_1](); \overline{c_1[i_{\mathrm{f}}, i_1]}\langle y\rangle \\
& \qquad\quad \mid !^{i_2\le n_2} c_2[i_{\mathrm{f}}, i_2](x' : D); \\
& \qquad\qquad \mathbf{find}\ u \le n_3\ \mathbf{suchthat}\ \mathbf{defined}(k[u]) \wedge \mathsf{true}\ \mathbf{then}\ \overline{c_2[i_{\mathrm{f}}, i_2]}\langle x' = x\rangle\ \mathbf{else} \\
& \qquad\qquad \mathbf{if}\ x' = x\ \mathbf{then}\ \mathbf{event}\ \mathsf{bad}\ \mathbf{else}\ \overline{c_2[i_{\mathrm{f}}, i_2]}\langle \mathsf{false}\rangle \\
& \qquad\quad \mid !^{i_3\le n_3} c_3[i_{\mathrm{f}}, i_3](); \mathbf{let}\ k : \mathit{bitstring} = \mathsf{mark}\ \mathbf{in}\ \overline{c_3[i_{\mathrm{f}}, i_3]}\langle x\rangle))
\end{aligned}$$

and for $a \in [1, n_{\mathrm{f}}]$,

$$\begin{aligned}
C_a = {} & \mathbf{newChannel}\ c; ([\,] \mid c_{\mathrm{k}}(); \overline{c}\langle\rangle; c(pk, y); \overline{c_{\mathrm{k}}}\langle\rangle; \\
& \quad (!^{i_0\le n_0} c_0[i_0](); \overline{c_0[i_0]}\langle pk\rangle \\
& \quad \mid !^{i_{\mathrm{f}}\le n_{\mathrm{f}}} c_{\mathrm{f}}[i_{\mathrm{f}}](); \mathbf{if}\ i_{\mathrm{f}} = a\ \mathbf{then} \\
& \qquad\qquad\qquad \overline{c_{\mathrm{f}}[i_{\mathrm{f}}]}\langle\rangle; (!^{i_1\le n_1} c_1[i_{\mathrm{f}}, i_1](); \overline{c_1[i_{\mathrm{f}}, i_1]}\langle y\rangle \\
& \qquad\qquad\qquad\qquad \mid !^{i_2\le n_2} c_2[i_{\mathrm{f}}, i_2](x' : D); \overline{c}\langle x'\rangle.c(); \overline{c_2[i_{\mathrm{f}}, i_2]}\langle \mathsf{false}\rangle) \\
& \qquad\qquad \mathbf{else}\ \mathbf{new}\ x : D; \overline{c_{\mathrm{f}}[i_{\mathrm{f}}]}\langle\rangle; \\
& \qquad\qquad\qquad (!^{i_1\le n_1} c_1[i_{\mathrm{f}}, i_1](); \overline{c_1[i_{\mathrm{f}}, i_1]}\langle \mathsf{f}(pk, x)\rangle \\
& \qquad\qquad\qquad \mid !^{i_2\le n_2} c_2[i_{\mathrm{f}}, i_2](x' : D); \overline{c_2[i_{\mathrm{f}}, i_2]}\langle x' = x\rangle \\
& \qquad\qquad\qquad \mid !^{i_3\le n_3} c_3[i_{\mathrm{f}}, i_3](); \overline{c_3[i_{\mathrm{f}}, i_3]}\langle x\rangle))))
\end{aligned}$$

We show that LR' executes **event** bad with probability at most $n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$ in the presence of a context that runs in time t, where C_a runs in time $t_{C_a} = (n_{\mathrm{f}} - 1)t_{\mathsf{f}}$.[1] By definition of one-wayness, the process LR defined in Section 3.2 executes **event** bad with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t)$. By Lemma 3, Property 5, $C_a[LR]$ executes **event** bad with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$. Let C be an evaluation context acceptable for $LR', LR', \emptyset$ that runs in time t. Consider a trace of $C[LR']$ that executes **event** bad. Let $a \in [1, n_{\mathrm{f}}]$ such that the first time **event** bad is executed in this trace, $i_{\mathrm{f}} = a$. Then the prefix of this trace up to the point at which it executes **event** bad for the first time can be simulated exactly by a trace of the same probability of $C[C_a[LR]]$. More precisely, the simulation proceeds as follows:

- When LR' receives a message on c_{k}, it picks a new seed r. Correspondingly $C_a[LR]$ also picks a seed r_0 by calling LR. It also chooses a random value x_0, so a single configuration of LR' of

[1] As usual in exact security proofs, we consider only the runtime of function evaluations and array lookups, and ignore the time for communications, random number generations, etc. We could obviously perform a more detailed time evaluation if desired.

probability p corresponds to $|D|$ configurations of $C_a[LR]$ that differ only by the value of x_0, each of probability $p/|D|$. Both LR' and $C_a[LR]$ reply by sending an empty message on c_{k}.

- When LR' receives a message on $c_0[i_0]$, it replies by sending $\mathsf{pkgen}(r)$ on $c_0[i_0]$. Correspondingly $C_a[LR]$ replies by sending the public key $pk = \mathsf{pkgen}(r_0)$.
- When LR' receives a message on $c_{\mathrm{f}}[i_{\mathrm{f}}]$, it picks a random value $x[i_{\mathrm{f}}]$. Correspondingly, $C_a[LR]$ either picks a random value $x[i_{\mathrm{f}}]$ if $i_{\mathrm{f}} \neq a$, or reuses the value of x_0 previously chosen by LR when $i_{\mathrm{f}} = a$. In the latter case, before executing this step, a single configuration of LR' of probability p corresponded to $|D|$ configurations of $C_a[LR]$ that differed only by the value of x_0, each of probability $p/|D|$; after executing this step, each configuration of LR' of probability $p/|D|$ corresponds to a single configuration of $C_a[LR]$, in which the chosen value of x_0 is the value of $x[a]$ in LR'. Both configurations have the same probability $p/|D|$. Both LR' and $C_a[LR]$ reply by sending an empty message on $c_{\mathrm{f}}[i_{\mathrm{f}}]$.
- When LR' receives a message on $c_1[i_{\mathrm{f}}, i_1]$, it replies by sending $\mathsf{f}(\mathsf{pkgen}(r), x[i_{\mathrm{f}}])$ on $c_1[i_{\mathrm{f}}, i_1]$. Correspondingly, $C_a[LR]$ replies either with $\mathsf{f}(pk, x[i_{\mathrm{f}}])$ when $i_{\mathrm{f}} \neq a$, or $y = \mathsf{f}(pk, x_0)$ when $i_{\mathrm{f}} = a$.
- When LR' receives a message x' on $c_2[i_{\mathrm{f}}, i_2]$, it replies by sending $x' = x[i_{\mathrm{f}}]$ on $c_2[i_{\mathrm{f}}, i_2]$ when $i_{\mathrm{f}} \neq a$ (since it never executes **event** bad with $i_{\mathrm{f}} \neq a$ in the considered trace prefix), by executing **event** bad when $x' = x[i_{\mathrm{f}}]$ and $i_{\mathrm{f}} = a$, and by sending false on $c_2[i_{\mathrm{f}}, i_2]$ when $x' \neq x[i_{\mathrm{f}}]$ and $i_{\mathrm{f}} = a$. (Since the considered trace prefix executes **event** bad with $i_{\mathrm{f}} = a$ at the end of this prefix, no $k[a, u]$ is defined at the end of this prefix, so no $k[a, u]$ is defined at any point in this trace prefix.) Correspondingly, $C_a[LR]$ replies by sending $x' = x[i_{\mathrm{f}}]$ on $c_2[i_{\mathrm{f}}, i_2]$ when $i_{\mathrm{f}} \neq a$ and by calling LR when $i_{\mathrm{f}} = a$ in order to execute **event** bad when $x' = x_0$; when $x' \neq x_0$, it returns from LR and sends false on $c_2[i_{\mathrm{f}}, i_2]$.
- When LR' receives a message on $c_3[i_{\mathrm{f}}, i_3]$, we have $i_{\mathrm{f}} \neq a$ since no $k[a, u]$ is defined at any point in the considered trace prefix, as mentioned above, and LR' replies by sending $x[i_{\mathrm{f}}]$ on $c_3[i_{\mathrm{f}}, i_3]$. Correspondingly, $C_a[LR]$ replies by sending $x[i_{\mathrm{f}}]$ on $c_3[i_{\mathrm{f}}, i_3]$ in this case.

Hence $\sum_{\mathcal{E}, \mathsf{bad} \in \mathcal{E}} \Pr[C[LR'] \rightsquigarrow \mathcal{E}] \leq \sum_{a \in [1, n_{\mathrm{f}}]} \sum_{\mathcal{E}, \mathsf{bad} \in \mathcal{E}} \Pr[C[C_a[LR]] \rightsquigarrow \mathcal{E}] \leq \sum_{a \in [1, n_{\mathrm{f}}]} \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a}) = n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$. So LR' executes **event** bad with probability at most $n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$.

By Lemma 6, $n_{\mathrm{k}} \times LR'$ executes **event** bad with probability at most $n_{\mathrm{k}} \times n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a} + (n_{\mathrm{k}} - 1) t_{LR'})$, where $t_{LR'} = t_{\mathsf{pkgen}} + n_{\mathrm{f}} t_{\mathsf{f}}$. (Note that, in an implementation of LR', one does not need to explicitly perform a **find** to test whether some $k[u]$ is defined. We can use one mutable bit for each i_{f} which we set when the last process of LR' is executed. We can then test this bit to determine whether $k[u]$ is defined for some u. The time of this test can then be neglected.) Let $t' = t_{C_a} + (n_{\mathrm{k}} - 1) t_{LR'} = (n_{\mathrm{k}} n_{\mathrm{f}} - 1) t_{\mathsf{f}} + (n_{\mathrm{k}} - 1) t_{\mathsf{pkgen}}$ and $p^{\mathsf{ow}}(t) = n_{\mathrm{k}} \times n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t')$. By Lemma 4,

$$(n_{\mathrm{k}} \times LR')\{\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{true} \rangle / \textbf{event } \mathsf{bad}\} \approx_{p^{\mathsf{ow}}} (n_{\mathrm{k}} \times LR')\{\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle / \textbf{event } \mathsf{bad}\}$$

The process **if** $x' = x$ **then** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{true} \rangle$ **else** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle$ can be replaced with the output $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x \rangle$, the process **find** $\ldots$ **then** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x \rangle$ **else** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x \rangle$ can be replaced with $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle x' = x \rangle$, and the assignments to pk and y can be expanded without changing the behavior of the process, so $(n_{\mathrm{k}} \times LR')\{\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{true} \rangle / \textbf{event } \mathsf{bad}\} \approx_0 [\![L]\!]$. The process **if** $x' = x$ **then** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle$ **else** $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle$ can be replaced with $\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle$, and the assignments to pk and y can be expanded, so $(n_{\mathrm{k}} \times LR')\{\overline{c_2[i_{\mathrm{k}}, i_{\mathrm{f}}, i_2]}\langle \mathsf{false} \rangle / \textbf{event } \mathsf{bad}\} \approx_0 [\![R]\!]$. Hence $[\![L]\!] \approx_{p^{\mathsf{ow}}} [\![R]\!]$. □

B Proof of the FDH Signature Example

Starting from the initial game Q_0 given in Section 2.1, the prover first tries to apply a cryptographic transformation. It succeeds applying the security of the hash function (6). Then each argument of a call to hash is first stored in an intermediate variable, x_{29} for m', x_{31} for m, and x_{33} for x, and each occurrence of a call to hash is replaced with a lookup in the three arrays that contain arguments of calls to hash, x_{29}, x_{31}, and x_{33}. When the argument of hash is found in one of these arrays, the returned result is the same as the result previously returned by hash. Otherwise, we pick a fresh random number and return it. Therefore, we obtain the following game:

```
$start()$; new $r : seed$; let $pk : pkey = \mathsf{pkgen}(r)$ in let $sk : skey = \mathsf{skgen}(r)$ in $\overline{c0}\langle pk\rangle$;
(
  $!^{!_{25} \leq qH}$
  $c4[!_{25}](x : bitstring)$;
  let $x_{33} : bitstring = x$ in
  find  suchthat defined$(x_{29}, r_{28}) \wedge (x_{33} = x_{29})$ then
    $\overline{c5[!_{25}]}\langle r_{28}\rangle$
  $\oplus$ $@i_{39} \leq qS$ suchthat defined$(x_{31}[@i_{39}], r_{30}[@i_{39}]) \wedge (x_{33} = x_{31}[@i_{39}])$ then
    $\overline{c5[!_{25}]}\langle r_{30}[@i_{39}]\rangle$
  $\oplus$ $@i_{38} \leq qH$ suchthat defined$(x_{33}[@i_{38}], r_{32}[@i_{38}]) \wedge (x_{33} = x_{33}[@i_{38}])$ then
    $\overline{c5[!_{25}]}\langle r_{32}[@i_{38}]\rangle$
  else
    new $r_{32} : D$;
    $\overline{c5[!_{25}]}\langle r_{32}\rangle$
|
  $!^{!_{26} \leq qS}$
  $c1[!_{26}](m : bitstring)$;
  let $x_{31} : bitstring = m$ in
  find  suchthat defined$(x_{29}, r_{28}) \wedge (x_{31} = x_{29})$ then
    $\overline{c2[!_{26}]}\langle \mathsf{mf}(sk, r_{28})\rangle$
  $\oplus$ $@i_{37} \leq qS$ suchthat defined$(x_{31}[@i_{37}], r_{30}[@i_{37}]) \wedge (x_{31} = x_{31}[@i_{37}])$ then
    $\overline{c2[!_{26}]}\langle \mathsf{mf}(sk, r_{30}[@i_{37}])\rangle$
  $\oplus$ $@i_{36} \leq qH$ suchthat defined$(x_{33}[@i_{36}], r_{32}[@i_{36}]) \wedge (x_{31} = x_{33}[@i_{36}])$ then
    $\overline{c2[!_{26}]}\langle \mathsf{mf}(sk, r_{32}[@i_{36}])\rangle$
  else
    new $r_{30} : D$;
    $\overline{c2[!_{26}]}\langle \mathsf{mf}(sk, r_{30})\rangle$
|
  $c3(m' : bitstring, s : D)$;
  let $x_{29} : bitstring = m'$ in
  find  suchthat defined$(x_{29}, r_{28}) \wedge (x_{29} = x_{29})$ then                                   1
    if $(\mathsf{f}(pk, s) = r_{28})$ then
    find $u \leq qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad
  $\oplus$ $@i_{35} \leq qS$ suchthat defined$(x_{31}[@i_{35}], r_{30}[@i_{35}]) \wedge (x_{29} = x_{31}[@i_{35}])$ then     2
    if $(\mathsf{f}(pk, s) = r_{30}[@i_{35}])$ then                                                  3
    find $u \leq qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad      4
  $\oplus$ $@i_{34} \leq qH$ suchthat defined$(x_{33}[@i_{34}], r_{32}[@i_{34}]) \wedge (x_{29} = x_{33}[@i_{34}])$ then
    if $(\mathsf{f}(pk, s) = r_{32}[@i_{34}])$ then
    find $u \leq qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$  else event bad
  else
    new $r_{28} : D$;
    if $(\mathsf{f}(pk, s) = r_{28})$ then
    find $u \le qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad
)
```

This game is automatically simplified as follows: The test $x_{29} = x_{29}$ at line 1 is replaced with true. The variables x_{29}, x_{31}, and x_{33} are substituted with their value, respectively m', m, and x. After this substitution, the values assigned to x_{29}, x_{31}, and x_{33} are no longer important, so they are replaced with constants cst_bitstring. (The fact that these variables are defined is tested in conditions of **find**, so the assignments cannot be removed completely.) Finally, the **find** at line 4 always succeeds, with $u = @i_{35}$, due to the **find** are line 2. So line 4 can be replaced with $\overline{0}$, and therefore lines 3 and 4 can be replaced with $\overline{0}$.

Then, the prover tries to apply a cryptographic transformation. All transformations fail, but when applying (4), the game contains $\mathsf{mf}(sk, y)$ while (4) expects $\mathsf{mf}(\mathsf{skgen}(r), y)$, which suggests to remove assignments to variable sk for it to succeed. So the prover performs this removal: it substitutes $\mathsf{skgen}(r)$ for sk and removes the assignment **let** $sk : skey = \mathsf{skgen}(r)$. The transformation (4) is then retried. It now succeeds, which leads to replacing r_j with $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ and $\mathsf{mf}(\mathsf{skgen}(r), r_j)$ with r_j. We obtain the following game:

```
$start()$; new $r : seed$; let $pk : pkey = \mathsf{pkgen}(r)$ in $\overline{c0}\langle pk\rangle$;
(
  $!^{!_{25} \le qH}$
  $c4[!_{25}](x : bitstring)$;
  let $x_{33} : bitstring$ = cst_bitstring in
  find  suchthat defined$(m', x_{29}, r_{28}) \wedge (x = m')$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}(\mathsf{pkgen}(r), r_{28})\rangle$
  $\oplus$ $@i_{39} \le qS$ suchthat defined$(m[@i_{39}], x_{31}[@i_{39}], r_{30}[@i_{39}]) \wedge (x = m[@i_{39}])$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}(\mathsf{pkgen}(r), r_{30}[@i_{39}])\rangle$
  $\oplus$ $@i_{38} \le qH$ suchthat defined$(x[@i_{38}], x_{33}[@i_{38}], r_{32}[@i_{38}]) \wedge (x = x[@i_{38}])$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}(\mathsf{pkgen}(r), r_{32}[@i_{38}])\rangle$
  else
    new $r_{32} : D$;
    $\overline{c5[!_{25}]}\langle \mathsf{f}(\mathsf{pkgen}(r), r_{32})\rangle$
|
  $!^{!_{26} \le qS}$
  $c1[!_{26}](m : bitstring)$;
  let $x_{31} : bitstring$ = cst_bitstring in
  find  suchthat defined$(m', x_{29}, r_{28}) \wedge (m = m')$ then
    $\overline{c2[!_{26}]}\langle r_{28}\rangle$
  $\oplus$ $@i_{37} \le qS$ suchthat defined$(m[@i_{37}], x_{31}[@i_{37}], r_{30}[@i_{37}]) \wedge (m = m[@i_{37}])$ then
    $\overline{c2[!_{26}]}\langle r_{30}[@i_{37}]\rangle$
  $\oplus$ $@i_{36} \le qH$ suchthat defined$(x[@i_{36}], x_{33}[@i_{36}], r_{32}[@i_{36}]) \wedge (m = x[@i_{36}])$ then
    $\overline{c2[!_{26}]}\langle r_{32}[@i_{36}]\rangle$
  else
    new $r_{30} : D$;
    $\overline{c2[!_{26}]}\langle r_{30}\rangle$
|
  $c3(m' : bitstring, s : D)$;
  let $x_{29} : bitstring$ = cst_bitstring in
  find  suchthat defined$(r_{28}) \wedge$ true then
    if $(\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_{28}))$ then
```

```
      find $u \le qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad
    $\oplus$ $@i_{35} \le qS$ suchthat defined$(m[@i_{35}], x_{31}[@i_{35}], r_{30}[@i_{35}]) \wedge (m' = m[@i_{35}])$ then
      $\overline{0}$
    $\oplus$ $@i_{34} \le qH$ suchthat defined$(x[@i_{34}], x_{33}[@i_{34}], r_{32}[@i_{34}]) \wedge (m' = x[@i_{34}])$ then
      if $(\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_{32}[@i_{34}]))$ then
      find $u \le qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad
    else
      new $r_{28} : D$;
      if $(\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_{28}))$ then
      find $u \le qS$ suchthat defined$(m[u]) \wedge (m' = m[u])$ then $\overline{0}$ else event bad
)
```

This game is automatically simplified as follows. In this game, it is useless to test whether $x_{33}[i]$ is defined, since when we require that $x_{33}[i]$ is defined, we also require that $r_{32}[i]$ is defined, and if $r_{32}[i]$ is defined, then $x_{33}[i]$ has been defined before. So the prover removes $x_{33}[i]$ from **defined** tests, and removes the assignments to x_{33}, which is no longer used. The situation is similar for x_{29} and x_{31}.

By injectivity of f, the prover replaces three occurrences of terms of the form $\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_j)$ with $s = r_j$, knowing $pk = \mathsf{pkgen}(r)$.

The prover then tries to apply cryptographic transformations. It succeeds using the definition of one-wayness (1). This transformation leads to replacing $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ with $\mathsf{f}'(\mathsf{pkgen}'(r), r_j)$, r_j with **let** $k_j : bitstring = \mathsf{mark}$ **in** r_j, and $s = r_j$ with **find** $@i_j \le N$ **suchthat defined**$(k_j[@i_j]) \wedge$ true **then** $s = r_j$ **else** false. The difference of probability is $(qH + qS + 1)\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (qH + qS)t_{\mathsf{f}})$ where t' is the runtime of the context put around the equivalence (1). After this transformation, we obtain the following game:

```
$start()$; new $r : seed$; let $pk : pkey = \mathsf{pkgen}'(r)$ in $\overline{c0}\langle pk\rangle$;
(
  $!^{!_{25} \le qH}$
  $c4[!_{25}](x : bitstring)$;
  find  suchthat defined$(m', r_{28}) \wedge (x = m')$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}'(\mathsf{pkgen}'(r), r_{28})\rangle$
  $\oplus$ $@i_{39} \le qS$ suchthat defined$(m[@i_{39}], r_{30}[@i_{39}]) \wedge (x = m[@i_{39}])$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}'(\mathsf{pkgen}'(r), r_{30}[@i_{39}])\rangle$
  $\oplus$ $@i_{38} \le qH$ suchthat defined$(x[@i_{38}], r_{32}[@i_{38}]) \wedge (x = x[@i_{38}])$ then
    $\overline{c5[!_{25}]}\langle \mathsf{f}'(\mathsf{pkgen}'(r), r_{32}[@i_{38}])\rangle$
  else
    new $r_{32} : D$;
    $\overline{c5[!_{25}]}\langle \mathsf{f}'(\mathsf{pkgen}'(r), r_{32})\rangle$
|
  $!^{!_{26} \le qS}$
  $c1[!_{26}](m : bitstring)$;                                                        1
  find  suchthat defined$(m', r_{28}) \wedge (m = m')$ then                           2
    let $k_{41} : bitstring = \mathsf{mark}$ in                                        3
    $\overline{c2[!_{26}]}\langle r_{28}\rangle$
  $\oplus$ $@i_{37} \le qS$ suchthat defined$(m[@i_{37}], r_{30}[@i_{37}]) \wedge (m = m[@i_{37}])$ then
    let $k_{42} : bitstring = \mathsf{mark}$ in
    $\overline{c2[!_{26}]}\langle r_{30}[@i_{37}]\rangle$
  $\oplus$ $@i_{36} \le qH$ suchthat defined$(x[@i_{36}], r_{32}[@i_{36}]) \wedge (m = x[@i_{36}])$ then      4
```

```
    let $k_{43}$ : $bitstring$ = mark in                                                                        5
    $\overline{c2[!_{26}]}\langle r_{32}[@i_{36}]\rangle$
  else
    new $r_{30} : D$;
    let $k_{40}$ : $bitstring$ = mark in
    $\overline{c2[!_{26}]}\langle r_{30}\rangle$
|
  $c3(m' : bitstring, s : D)$;
  find  suchthat defined($r_{28}$) $\wedge$ true then
    find $@i_{46} \leq qS$ suchthat defined($k_{41}[@i_{46}]$) $\wedge$ true then                                6
      if ($s = r_{28}$) then
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad     7
    else
      if false then                                                                                             8
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad
  $\oplus$ $@i_{35} \leq qS$ suchthat defined($r_{30}[@i_{35}], m[@i_{35}]$) $\wedge$ ($m' = m[@i_{35}]$) then
    $\overline{0}$
  $\oplus$ $@i_{34} \leq qH$ suchthat defined($x[@i_{34}], r_{32}[@i_{34}]$) $\wedge$ ($m' = x[@i_{34}]$) then      9
    find $@i_{49} \leq qS$ suchthat defined($k_{43}[@i_{49}]$) $\wedge$ ($@i_{36}[@i_{49}] = @i_{34}$) then        10
      if ($s = r_{32}[@i_{34}]$) then
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad     11
    else
      if false then                                                                                             12
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad
  else
    new $r_{28} : D$;
    find $@i_{44} \leq qS$ suchthat defined($k_{41}[@i_{44}]$) $\wedge$ true then
      if ($s = r_{28}$) then
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad     13
    else
      if false then                                                                                             14
      find $u \leq qS$ suchthat defined($m[u]$) $\wedge$ ($m' = m[u]$) then $\overline{0}$ else event bad
)
```

The prover then simplifies the obtained game automatically. The tests **if** false **then**... at lines 8, 12, and 14 are obviously simplified out. The **find**s at lines 7, 11, and 13 always succeed. At line 7, $k_{41}[@i_{46}]$ is defined according to the condition of the **find** at line 6. Since k_{41} is defined only at line 3, $m[@i_{46}]$ is then defined (line 1) and $m[@i_{46}] = m'$ by the condition of the **find** at line 2. So the **find** at line 7 succeeds with $u = @i_{46}$. The reasoning is similar for line 13. At line 11, $k_{43}[@i_{49}]$ is defined by the condition of the **find** at line 10. Since k_{43} is defined only at line 5, $m[@i_{49}]$ is defined (line 1), and $m[@i_{49}] = x[@i_{36}[@i_{49}]]$ by the condition of the **find** at line 4, $@i_{36}[@i_{49}] = @i_{34}$ by the condition of the **find** at line 10, and $m' = x[@i_{34}]$ by the condition of the **find** at line 9, so $m[@i_{49}] = x[@i_{36}[@i_{49}]] = x[@i_{34}] = m'$. So the **find** at line 11 succeeds with $u = @i_{49}$.

After these simplifications, **event** bad has been removed, so the probability that **event** bad is executed in the final game is 0. Therefore, exploiting Lemma 3, Properties 2 and 3, the system concludes that the initial game executes **event** bad with probability $p(t) \leq (qH + qS + 1)\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (qH + qS)t_{\mathsf{f}})$ where t' is the runtime of context put around the equivalence (1). (The only transformation that introduced a difference of probability is the application of one-wayness (1).)

C Definition of Security of Signatures

Lemma 7. *Let* $\mathsf{skgen}' : \mathit{seed} \rightarrow \mathit{skey}$, $\mathsf{pkgen}' : \mathit{seed} \rightarrow \mathit{pkey}$, $\mathsf{sign}' : \mathit{bitstring} \times \mathit{skey} \rightarrow \mathit{signature}$, *and* $\mathsf{verify}' : \mathit{bitstring} \times \mathit{pkey} \times \mathit{signature} \rightarrow \mathit{bool}$ *such that the functions associated to the primed symbols* $\mathsf{skgen}', \mathsf{pkgen}', \mathsf{sign}', \mathsf{verify}'$ *are equal to the functions associated to their corresponding unprimed symbol* $\mathsf{skgen}, \mathsf{pkgen}, \mathsf{sign}, \mathsf{verify}$. *We have the following equivalence:*

```
1.  $!^{i_{\mathrm{k}} \leq n_{\mathrm{k}}}\mathbf{new}\ r : \mathit{seed}; ($
2.      $!^{i_1 \leq n_1}() \rightarrow \mathsf{pkgen}(r),$
3.      $!^{i_{\mathrm{s}} \leq n_{\mathrm{s}}}(m : \mathit{bitstring}) \rightarrow \mathsf{sign}(m, \mathsf{skgen}(r))),$
4.  $!^{i_{\mathrm{v}} \leq n_{\mathrm{v}}}(m' : \mathit{bitstring}, y : \mathit{pkey}, s : \mathit{signature}) \rightarrow \mathsf{verify}(m', y, s)\ [\mathit{all}]$
5.  $\approx_{p^{\mathsf{uf-cma}}}$
6.  $!^{i_{\mathrm{k}} \leq n_{\mathrm{k}}}\mathbf{new}\ r : \mathit{seed}; ($
7.      $!^{i_1 \leq n_1}() \rightarrow \mathsf{pkgen}'(r),$
8.      $!^{i_{\mathrm{s}} \leq n_{\mathrm{s}}}(m : \mathit{bitstring}) \rightarrow \mathsf{sign}'(m, \mathsf{skgen}'(r))),$
9.  $!^{i_{\mathrm{v}} \leq n_{\mathrm{v}}}(m' : \mathit{bitstring}, y : \mathit{pkey}, s : \mathit{signature}) \rightarrow$
10.     $\mathbf{find}\ u_{\mathrm{k}} \leq n_{\mathrm{k}}, u_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{suchthat}\ \mathbf{defined}(r[u_{\mathrm{k}}], m[u_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge$
11.                 $y = \mathsf{pkgen}'(r[u_{\mathrm{k}}]) \wedge m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}] \wedge \mathsf{verify}'(m', y, s)\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}$
12.     $\mathbf{find}\ u_{\mathrm{k}} \leq n_{\mathrm{k}}\ \mathbf{suchthat}\ \mathbf{defined}(r[u_{\mathrm{k}}]) \wedge y = \mathsf{pkgen}'(r[u_{\mathrm{k}}])\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}$
13.     $\mathsf{verify}(m', y, s)$
```

where $p^{\mathsf{uf-cma}}(t) = n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(l_{\mathrm{s}}, l_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)(t_{\mathsf{pkgen}} + t_{\mathsf{skgen}} + n_{\mathrm{s}} t_{\mathsf{sign}}(l_{\mathrm{s}})) + (n_{\mathrm{k}} + n_{\mathrm{v}} - 1)(t_{\mathsf{verify}}(l_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{s}}, l_{\mathrm{v}})) + n_{\mathrm{v}} t_{\mathbf{find}}(n_{\mathrm{k}}, l_{\mathit{pkey}}))$*;* t_{pkgen}*,* t_{skgen}*,* $t_{\mathsf{sign}}(l)$*,* $t_{\mathsf{verify}}(l)$ *are the times for one evaluation of* pkgen*,* skgen*,* sign*,* verify *respectively, with a message of length at most* l*;* $t_{\mathbf{find}}(n, l)$ *is the time of a* **find** *that looks up a bit-string of length at most* l *in an array of at most* n *cells;* l_{pkey} *is the maximum length of a key in pkey;* $l_{\mathrm{s}} = \max_{i_{\mathrm{k}} \in [1, n_{\mathrm{k}}], i_{\mathrm{s}} \in [1, n_{\mathrm{s}}]} \mathrm{length}(m[i_{\mathrm{k}}, i_{\mathrm{s}}])$*; and* $l_{\mathrm{v}} = \max_{i_{\mathrm{v}} \in [1, n_{\mathrm{v}}]} \mathrm{length}(m'[i_{\mathrm{v}}])$.

As for one-wayness, this equivalence considers n_{k} keys instead of a single one. We denote by n_{s} the number of signature queries for each key and by n_{v} the total number of verification queries. We use primed function symbols to avoid the repeated application of the transformation of the left-hand side into the right-hand side. Note that we use verify and not verify' at line 13 in order to allow a repeated application of the transformation with a different key. The first three lines of each side of the equivalence express that the generation of public keys and the computation of the signature are left unchanged in the transformation. The verification of a signature $\mathsf{verify}(m', y, s)$ is replaced with a lookup in the previously computed signatures: if the signature is verified using one of the keys $\mathsf{pkgen}'(r[u_{\mathrm{k}}])$ (that is, if $y = \mathsf{pkgen}'(r[u_{\mathrm{k}}])$), then it can be valid only when it has been computed by the signature oracle $\mathsf{sign}'(m, \mathsf{skgen}'(r[u_{\mathrm{k}}]))$, that is, when $m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}]$ for some u_{s}. Lines 10-11 try to find such u_{k} and u_{s} and return true when they succeed. Line 12 returns false when no such u_{s} is found in lines 10-11, but $y = \mathsf{pkgen}'(r[u_{\mathrm{k}}])$ for some u_{k}. The last line handles the case when the key y is not $\mathsf{pkgen}'(r[u_{\mathrm{k}}])$. In this case, we verify the signature as before. The indication *all* at line 4 instructs the prover to transform all occurrences of function verify into the corresponding right-hand side.

Proof. We denote by L the left-hand side of the equivalence above and by R its right-hand side. We denote by $[\![L]\!]$ and $[\![R]\!]$ the translations of L and R into processes, respectively, and show that $[\![L]\!] \approx_{p^{\mathsf{uf-cma}}} [\![R]\!]$.

By definition of $\mathsf{UF}-\mathsf{CMA}$, the process LR defined in Section 3.3 executes **event** bad with probability at most $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, l, t)$ in the presence of a context that runs in time t where l is the maximum length of m and m'. By Lemma 6, $n_{\mathrm{k}} \times LR$ executes **event** bad with probability at most $n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(l'_{\mathrm{s}}, l'_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)t_{LR})$, where $t_{LR} = t_{\mathsf{pkgen}} + t_{\mathsf{skgen}} + n_{\mathrm{s}} t_{\mathsf{sign}}(l'_{\mathrm{s}}) + t_{\mathsf{verify}}(l'_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{s}}, l'_{\mathrm{v}})$, $l'_{\mathrm{s}} = \max_{i_{\mathrm{k}} \in [1, n_{\mathrm{k}}], i_{\mathrm{s}} \in [1, n_{\mathrm{s}}]} \mathrm{length}(m[i_{\mathrm{k}}, i_{\mathrm{s}}])$, and $l'_{\mathrm{v}} = \max_{i_{\mathrm{k}} \in [1, n_{\mathrm{k}}]} \mathrm{length}(m'[i_{\mathrm{k}}])$.

$$\begin{array}{l}
n_{\mathrm{k}} \times LR = \\
\quad !^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} start[i_{\mathrm{k}}](); \mathbf{new}\ r : seed; \mathbf{let}\ pk : pkey = \mathsf{pkgen}(r)\ \mathbf{in}\ \mathbf{let}\ sk : skey = \mathsf{skgen}(r)\ \mathbf{in}\ \overline{c_0[i_{\mathrm{k}}]}\langle pk \rangle; \\
\qquad (!^{i_{\mathrm{s}} \leq n_{\mathrm{s}}} c_1[i_{\mathrm{k}}, i](m : bitstring); \overline{c_2[i_{\mathrm{k}}, i]}\langle \mathsf{sign}(m, sk) \rangle \\
\qquad \mid c_3[i_{\mathrm{k}}](m' : bitstring, s : signature); \mathbf{if}\ \mathsf{verify}(m', pk, s)\ \mathbf{then} \\
\qquad\quad \mathbf{find}\ u_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{suchthat}\ \mathbf{defined}(m[i_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge m' = m[i_{\mathrm{k}}, u_{\mathrm{s}}]\ \mathbf{then}\ \overline{0}\ \mathbf{else}\ \mathbf{event}\ \mathsf{bad})
\end{array}$$

Let LR' be obtained from $n_{\mathrm{k}} \times LR$ by replacing **event** bad with **event** $\mathsf{bad}; \overline{yield}\langle\rangle; !^{i_{\mathrm{v}} \leq n_{\mathrm{v}}} c_3[i_{\mathrm{k}}](m' : bitstring, s : signature);$ **event** bad. ($n_{\mathrm{k}} \times LR$ executes **event** bad only when it receives the first forged signature. The previous replacement allows LR' to execute **event** bad several times, after receiving a forged signature.) By Lemma 5, LR' executes **event** bad with probability at most $n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(l'_{\mathrm{s}}, l'_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)t_{LR})$. Let

$$\begin{array}{l}
C = \mathbf{newChannel}\ start; \mathbf{newChannel}\ c_0; \mathbf{newChannel}\ c_1; \mathbf{newChannel}\ c_2; \mathbf{newChannel}\ c_3; \\
\quad ([\,] \mid !^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} c'_{\mathrm{k}}[i_{\mathrm{k}}](); \overline{start[i_{\mathrm{k}}]}\langle\rangle; c_0[i_{\mathrm{k}}](pk); \overline{c'_{\mathrm{k}}[i_{\mathrm{k}}]}\langle\rangle; \\
\qquad\quad (!^{i_1 \leq n_1} c'_1[i_{\mathrm{k}}, i_1](); \overline{c'_1[i_{\mathrm{k}}, i_1]}\langle pk \rangle \\
\qquad\quad \mid !^{i_{\mathrm{s}} \leq n_{\mathrm{s}}} c'_{\mathrm{s}}[i_{\mathrm{k}}, i_{\mathrm{s}}](m : bitstring); \overline{c_1[i_{\mathrm{k}}, i_{\mathrm{s}}]}\langle m \rangle; c_2[i_{\mathrm{k}}, i_{\mathrm{s}}](s : signature); \overline{c'_{\mathrm{s}}[i_{\mathrm{k}}, i_{\mathrm{s}}]}\langle s \rangle) \\
\qquad \mid !^{i_{\mathrm{v}} \leq n_{\mathrm{v}}} c'_{\mathrm{v}}[i_{\mathrm{v}}](m' : bitstring, y : pkey, s : signature); \\
\qquad\quad \mathbf{find}\ u_{\mathrm{k}} \leq n_{\mathrm{k}}\ \mathbf{suchthat}\ \mathbf{defined}(pk[u_{\mathrm{k}}]) \wedge y = pk[u_{\mathrm{k}}]\ \mathbf{then} \\
\qquad\qquad \mathbf{if}\ \mathsf{verify}(m', y, s)\ \mathbf{then} \\
\qquad\qquad\quad \mathbf{find}\ u_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{suchthat}\ \mathbf{defined}(m[u_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}]\ \mathbf{then} \\
\qquad\qquad\qquad \overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true} \rangle\ \mathbf{else}\ \overline{c_3[u_{\mathrm{k}}]}\langle m', s \rangle \\
\qquad\qquad \mathbf{else}\ \overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{false} \rangle \\
\qquad\quad \mathbf{else}\ \overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{verify}(m', y, s) \rangle)
\end{array}$$

By Lemma 3, Property 5, $C[LR']$ executes **event** bad with probability at most $n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(l_{\mathrm{s}}, l_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)t_{LR} + t_C)$ where C runs in time $t_C = n_{\mathrm{v}}(t_{\mathsf{verify}}(l_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{k}}, l_{pkey}) + t_{\mathbf{find}}(n_{\mathrm{s}}, l_{\mathrm{v}}))$. Let $t' = (n_{\mathrm{k}} - 1)t_{LR} + t_C \leq (n_{\mathrm{k}} - 1)(t_{\mathsf{pkgen}} + t_{\mathsf{skgen}} + n_{\mathrm{s}} t_{\mathsf{sign}}(l_{\mathrm{s}})) + (n_{\mathrm{k}} + n_{\mathrm{v}} - 1)(t_{\mathsf{verify}}(l_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{s}}, l_{\mathrm{v}})) + n_{\mathrm{v}} t_{\mathbf{find}}(n_{\mathrm{k}}, l_{pkey})$, since $l'_{\mathrm{s}} \leq l_{\mathrm{s}}$ and $l'_{\mathrm{v}} \leq l_{\mathrm{v}}$.

Let $p^{\mathsf{uf-cma}}(t) = n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(l_{\mathrm{s}}, l_{\mathrm{v}}), t + t')$. By eliminating communications on restricted channels $start, c_0, c_1, c_2, c_3$, we can easily see that $C[LR'] \approx_0 LR''$ where

$$\begin{array}{l}
LR'' = !^{i_{\mathrm{k}} \leq n_{\mathrm{k}}} c'_{\mathrm{k}}[i_{\mathrm{k}}](); \mathbf{new}\ r : seed; \overline{c'_{\mathrm{k}}[i_{\mathrm{k}}]}\langle\rangle; \\
\qquad\qquad (!^{i_1 \leq n_1} c'_1[i_{\mathrm{k}}, i_1](); \overline{c'_1[i_{\mathrm{k}}, i_1]}\langle \mathsf{pkgen}(r) \rangle \\
\qquad\qquad \mid !^{i_{\mathrm{s}} \leq n_{\mathrm{s}}} c'_{\mathrm{s}}[i_{\mathrm{k}}, i_{\mathrm{s}}](m : bitstring); \overline{c'_{\mathrm{s}}[i_{\mathrm{k}}, i_{\mathrm{s}}]}\langle \mathsf{sign}(m, \mathsf{skgen}(r)) \rangle)
\end{array}$$

```
$\mid\,!^{i_{\mathrm{v}}\leq n_{\mathrm{v}}} c'_{\mathrm{v}}[i_{\mathrm{v}}](m' : \mathit{bitstring}, y : \mathit{pkey}, s : \mathit{signature});$
    $\textbf{find } u_{\mathrm{k}} \leq n_{\mathrm{k}} \textbf{ suchthat defined}(\mathit{pk}[u_{\mathrm{k}}]) \wedge y = \mathit{pk}[u_{\mathrm{k}}] \textbf{ then}$
      $\textbf{if } \mathsf{verify}(m', y, s) \textbf{ then}$
        $\textbf{find } u_{\mathrm{s}} \leq n_{\mathrm{s}} \textbf{ suchthat defined}(m[u_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}] \textbf{ then}$
          $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle \textbf{ else event } \mathsf{bad}$
      $\textbf{else } \overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{false}\rangle$
    $\textbf{else } \overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{verify}(m', y, s)\rangle$
```

Indeed, when C sends a message m', s on $c_3[u_{\mathrm{k}}]$, C has checked that m', s is a forged signature under the key $\mathit{pk}[u_{\mathrm{k}}]$, so LR' always executes **event** bad when receiving this message. So by Lemma 3, Property 3, LR'' executes **event** bad with probability at most $p^{\mathsf{uf-cma}}$. By Lemma 4,

$$\mathit{LR}''\{\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle/\textbf{event } \mathsf{bad}\} \approx_{p^{\mathsf{uf-cma}}} \mathit{LR}''\{\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{false}\rangle/\textbf{event } \mathsf{bad}\}.$$

The process **find**... **then** $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle$ **else** $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle$ can be replaced with the output $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle$ and **if** $\mathsf{verify}(m', y, s)$ **then** $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle$ **else** $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{false}\rangle$ can be replaced with $\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{verify}(m', y, s)\rangle$ without changing the behavior of the process, so $\mathit{LR}''\{\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{true}\rangle/\textbf{event } \mathsf{bad}\} \approx_0 [\![L]\!]$. By reorganizing **find**s, we can prove the equivalence $\mathit{LR}''\{\overline{c'_{\mathrm{v}}[i_{\mathrm{v}}]}\langle \mathsf{false}\rangle/\textbf{event } \mathsf{bad}\} \approx_0 [\![R]\!]$. Hence $[\![L]\!] \approx_{p^{\mathsf{uf-cma}}} [\![R]\!]$. □