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: unforgeability under chosen-message attacks (UF-CMA) of the Full-Domain Hash 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 [8]: 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 [21] 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 paper by Diffie and Hellman [20], 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 [23]. 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 [22], 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, 35]). 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 (*i.e.* 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 [38] suggested to prove it by small changes [11], 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, 18, 25] 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 [5, 6, 3] have designed an abstract cryptographic library and shown its soundness with respect to computational primitives, under arbitrary active attacks. Backes and Pfitzmann [4] 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 [39]. Canetti [16] introduced the notion of universal composability. With Herzog [17], 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 [12] for verifying protocols in this framework. Lincoln, Mateus, Mitchell, Mitchell, Ramanathan, Scedrov, and Teague [29–31, 36, 33] developed a probabilistic polynomial-time calculus for the analysis of cryptographic protocols. Datta *et al* [19] 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 [17] 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. A type inference algorithm is sketched in [2].

Barthe, Cerderquist, and Tarento [7, 40] have formalized the generic model and the random oracle model in the interactive theorem prover Coq, and proved signature schemes in this framework. In contrast to our specialized prover, proofs in generic interactive theorem provers require a lot of human effort, in order to build a detailed enough proof for the theorem prover to check it.

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

Our prover, which we describe in this paper, was previously presented in [13, 14], but in a more restricted way. It was indeed applied only to classical, 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. [13, 14] 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 [13, 14] considered only secrecy properties. Finally, we also show how to model a random oracle.

Achievements. As in [13, 14], 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, has totally disappeared: the adversary eventually 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 [13, 14] in order to model games as done in computational security proofs. This calculus has been carefully designed to make the automatic proof of cryptographic protocols easier. 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 [9], 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 [13, 14], we adopt the exact security framework [10], instead of the asymptotic one. The cost of the reductions, and the probability loss will thus be precisely determined. We also adapt the syntax of our calculus, in order to be closer to the usual syntax of cryptographic games.

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 f comes 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*).

Let us now illustrate on an example how we represent games in our process calculus. As we shall see in the next sections, this example comes from the definition of security of the Full-Domain Hash (FDH) signature scheme [9]. This example uses the function symbols hash, pkgen, skgen, f, and invf (such that $x \mapsto \mathsf{invf}(sk, x)$ is the inverse of the function $x \mapsto \mathsf{f}(pk, x)$), which will all be explained later in detail. We define an oracle *Ogen* which chooses a random seed r, generates a key pair (pk, sk) from this seed, and returns the public key pk:

$$Ogen() := r \stackrel{R}{\leftarrow} \mathit{seed}; pk \leftarrow \mathsf{pkgen}(r); sk \leftarrow \mathsf{skgen}(r); \mathbf{return}(pk)$$

The seed r is chosen randomly with uniform probability in the type *seed* by the construct $r \stackrel{R}{\leftarrow} \mathit{seed}$. (The type *seed* must be a fixed-length type, because probabilistic bounded-time Turing machines can choose random numbers uniformly only in such types. The set of bit-strings *seed* is associated to a fixed value of the security parameter.)

Next, we define a signature oracle OS which takes as argument a bit-string m and returns its FDH signature, computed as $\mathsf{invf}(sk, \mathsf{hash}(m))$, where sk is the secret key, so this oracle could be defined by

$$OS(m : \mathit{bitstring}) := \mathbf{return}(\mathsf{invf}(sk, \mathsf{hash}(m)))$$

where $m : \mathit{bitstring}$ means that m is of type *bitstring*, that is, it is any bit-string. However, this oracle can be called several times, say at most qS times. We express this repetition by **foreach** $iS \leq qS$ **do** OS, meaning that we make available qS copies of OS, each with a different value of the index $iS \in [1, qS]$. Furthermore, in our calculus, variables defined in repeated oracles are arrays with a cell for each call to the oracle, so that we can remember the values used in all calls to the oracles. Here, m is then an array indexed by iS. Along similar lines, the copies of the oracle OS itself are indexed by iS, so that the caller can specify exactly which copy of OS he wants to call, by calling $OS[iS]$ for a specific value of iS. So we obtain the following formalization of this oracle:

$$\mathbf{foreach}\ iS \leq qS\ \mathbf{do}\ OS[iS](m[iS] : \mathit{bitstring}) := \mathbf{return}(\mathsf{invf}(sk, \mathsf{hash}(m[iS]))) \tag{1}$$

Note that sk has no array index, since it is defined in the oracle *Ogen*, which is executed only once.

We also define a test oracle OT which takes as arguments a bit-string m' and a candidate signature s of type D and executes the event forge when s is a forged signature of m', that is, s is a correct signature of m' and the signature oracle has not been called on m'. The test oracle can be defined as follows:

$$\begin{array}{l} OT(m' : \mathit{bitstring}, s : D) := \textbf{if } \mathsf{f}(pk, s) = \mathsf{hash}(m') \textbf{ then} \\ \quad \textbf{find } u \leq qS \textbf{ suchthat } (\textbf{defined}(m[u]) \wedge m' = m[u]) \textbf{ then end} \\ \hfill \textbf{else event } \mathsf{forge} \end{array} \tag{2}$$

It first 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 ends the oracle, as if it executed **end**. 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 by the signing oracle, we simply end the oracle. Otherwise, we execute the event forge and implicitly end the oracle. Arrays and array lookups are crucial in this calculus, and will help to model many properties which were hard to capture.

Finally, we add a hash oracle, which is similar to the signing oracle OS but returns the hash of the message instead of its signature:

$$\textbf{foreach } iH \leq qH \textbf{ do } OH[iH](x[iH] : \mathit{bitstring}) := \textbf{return}(\mathsf{hash}(x[iH]))$$

To lighten the notation, some array indexes can be omitted in the input we give to our prover. Precisely, when x is defined under **foreach** $i_1 \leq n_1 \ldots$ **foreach** $i_m \leq n_m$, x is always an array with indexes $i_1, \ldots, i_m$, so we abbreviate all occurrences of $x[i_1, \ldots, i_m]$ by x. Here, all array indexes in OS and OH can then be omitted.

We can remark that the signature and test oracles only make sense after the generation oracle $Ogen$ has been called, since they make use of the keys pk and sk computed by $Ogen$. So we define OS and OT after $Ogen$ by a sequential composition. In contrast, OS and OT are simultaneously available, so we use a parallel composition $Q_S \mid Q_T$ where Q_S and Q_T are the processes (1) and (2) respectively. Similarly, OH is composed in parallel with the rest of the process. So we obtain the following game which models the security of the FDH signature scheme in the random oracle model:

$$\begin{array}{l} G_0 = \textbf{foreach } iH \leq qH \textbf{ do } OH(x : \mathit{bitstring}) := \textbf{return}(\mathsf{hash}(x)) \\ \quad \mid Ogen() := r \stackrel{R}{\leftarrow} \mathit{seed}; pk \leftarrow \mathsf{pkgen}(r); sk \leftarrow \mathsf{skgen}(r); \textbf{return}(pk); \\ \quad\quad (\textbf{foreach } iS \leq qS \textbf{ do } OS(m : \mathit{bitstring}) := \textbf{return}(\mathsf{invf}(sk, \mathsf{hash}(m))) \\ \quad\quad \mid OT(m' : \mathit{bitstring}, s : D) := \textbf{if } \mathsf{f}(pk, s) = \mathsf{hash}(m') \textbf{ then} \\ \quad\quad\quad \textbf{find } u \leq qS \textbf{ suchthat } (\textbf{defined}(m[u]) \wedge m' = m[u]) \textbf{ then end} \\ \hfill \textbf{else event } \mathsf{forge}) \end{array}$$

Our calculus obviously also has a construct for calling oracles. However, we do not need it explicitly in this paper, because oracles are called by the adversary, not by processes we write ourselves.

As detailed in [13, 14], we require some *well-formedness invariants* to guarantee that several definitions of the same oracle cannot be simultaneously available, 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 [13].

2.2 Observational Equivalence

We denote by $\Pr[Q \rightsquigarrow a]$ the probability that the answer of Q to the oracle call $Ostart()$ is a, where $Ostart$ is an oracle called to start the experiment. 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 $\mathcal{E}$, when oracle $Ostart()$ is called.

In the next definition, we use a context C to represent an algorithm that tries to distinguish Q from Q'. A context C is put around a process Q by $C[Q]$. This construct means that Q is put in parallel with some other process Q' contained in C, possibly hiding some oracles defined in Q, so that, when considering $C'[C[Q]]$, C' cannot call these oracles. This will be detailed in the following of this section.

Definition 1 (Observational equivalence). *Let Q and Q' be two processes that satisfy the well-formedness invariants.*

A context C is said to be acceptable *for Q if and only if C does not contain events, C and Q have no common variables, and $C[Q]$ satisfies the well-formedness invariants.*

We say that Q and Q' are observationally equivalent *up to probability p, written $Q \approx_p Q'$, when for all t, for all contexts C acceptable for Q and Q' that run in time at most t, for all bit-strings a, $|\Pr[C[Q] \rightsquigarrow a] - \Pr[C[Q'] \rightsquigarrow a]| \leq p(t)$ and $\sum_{\mathcal{E}} |\Pr[C[Q] \rightsquigarrow \mathcal{E}] - \Pr[C[Q'] \rightsquigarrow \mathcal{E}]| \leq p(t)$.*

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 context C is not allowed to access directly the variables of Q (using **find**). 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. Indeed, we bound the length of messages in calls or returns to oracle O by a value $\mathrm{maxlen}(O, \mathrm{arg}_i)$ or $\mathrm{maxlen}(O, \mathrm{res}_i)$. Longer messages are truncated. The length of random numbers created by C is bounded; the number of instructions executed by C is bounded; and the time of a function evaluation is bounded by a function of the length of its arguments.)

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

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

Lemma 1. *1. $\approx_p$ is reflexive and symmetric.*
2. If $Q \approx_p Q'$ and $Q' \approx_{p'} Q''$, then $Q \approx_{p+p'} Q''$.
3. If Q executes event e with probability at most p and $Q \approx_{p'} Q'$, then Q' executes event e with probability at most $p + p'$.
4. If $Q \approx_p Q'$ and C is a context acceptable for Q and Q' that runs in time t_C, then $C[Q] \approx_{p'} C[Q']$ where $p'(t) = p(t + t_C)$.
5. If Q executes event e with probability at most p and C is a context acceptable for Q that runs in time t_C, then $C[Q]$ executes event e 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 G_0 corresponding to the initial attack, and build a sequence of observationally equivalent games $G_0 \approx_{p_1} G_1 \approx_{p_2} \ldots \approx_{p_m} G_m$. By Property 2, we conclude that $G_0 \approx_{p_1+\ldots+p_m} G_m$. By Property 3, we can bound the probability that G_0 executes an event from the probability that G_m executes this event.

The elementary transformations used to build each game from the previous one can in particular come from an algorithmic assumption on a cryptographic primitive. This assumption needs to be specified as an observational equivalence $L \approx_p R$. To use it to transform a game G, the prover finds a context C such that $G \approx_0 C[L]$ by purely syntactic transformations, and builds a game G' such that $G' \approx_0 C[R]$ by purely syntactic transformations. C is the simulator usually defined for reductions. By Property 4, we have $C[L] \approx_{p'} C[R]$, so $G \approx_{p'} G'$. The context C typically hides the oracles of L and R so that they are visible from C but not from the adversary C' against $G \approx_{p'} G'$. The context $C'[C[\,]]$ then defines the adversary against the algorithmic assumption $L \approx_p R$.

If the security assumptions are initially not in the form of an equivalence $L \approx_p R$, one needs to manually prove such an equivalence that formalizes the desired security assumption. The design of such equivalences can be delicate, but this is a one-time effort: the same equivalence can be reused for proofs that rely on the same assumption. For instance, we give below such an equivalence for one-wayness, and use it not only for the proof of the FDH signature scheme, but also for proofs of encryption schemes as mentioned in Section 4.2. Similarly, the definition of security of a signature (UF-CMA) says that some event is executed with negligible probability. When we want to prove the security of a protocol using a signature scheme, we use a manual proof of an equivalence that corresponds to that definition, done once for $\mathsf{UF\text{-}CMA}$ in Appendix B.3.

The prover automatically establishes certain equivalences $G_0 \approx_p G_m$ as mentioned above. However, the user can give only the left-hand side of the equivalence G_0; the right-hand side G_m is obtained by the prover. As a consequence, the prover is in general not appropriate for proving automatically properties $L \approx_p R$ in which L and R are both given a priori: the right-hand side found by the prover is unlikely to correspond exactly to the desired right-hand side. On the other hand, the prover can check security properties on the right-hand side G_m it finds, for example that the event forge cannot be executed by G_m. Using $G_0 \approx_p G_m$, it concludes that G_0 executes forge with probability at most p.

3 Characterization of One-wayness and Unforgeability

In this section, we introduce the assumption (one-wayness) and the security notion (unforgeability) to achieve.

3.1 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* invf. Given an element y, and the trapdoor sk, invf 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 $\mathcal{A}$ within a reasonable time is small, where this success is commonly defined by

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

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: $\mathsf{skgen} : seed \to skey$ generates secret keys; $\mathsf{pkgen} : seed \to pkey$ generates public keys; $\mathsf{f} : pkey \times D \to D$ and $\mathsf{invf} : skey \times D \to D$, such that, for each pk, $x \mapsto \mathsf{f}(pk, x)$ is a permutation of D, whose inverse permutation is $x \mapsto \mathsf{invf}(sk, x)$ 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** invert with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t)$ in the presence of a context that runs in time t, where

$$\begin{aligned} LR = {} & Ogen() := r_0 \stackrel{R}{\leftarrow} seed; x_0 \stackrel{R}{\leftarrow} D; \mathbf{return}(\mathsf{pkgen}(r_0), \mathsf{f}(\mathsf{pkgen}(r_0), x_0)); \\ & Oeq(x' : D) := \mathbf{if}\ x' = x_0\ \mathbf{then\ event}\ \mathsf{invert} \end{aligned}$$

$$
\begin{aligned}
&\textbf{foreach } i_{\mathrm{k}} \le n_{\mathrm{k}} \textbf{ do } r \xleftarrow{R} \mathit{seed}; (\mathit{Opk}() := \textbf{return}(\mathsf{pkgen}(r))\\
&\quad \mid \textbf{foreach } i_{\mathrm{f}} \le n_{\mathrm{f}} \textbf{ do } x \xleftarrow{R} D; (\mathit{Oy}() := \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), x))\\
&\qquad \mid \textbf{foreach } i_1 \le n_1 \textbf{ do } \mathit{Oeq}(x' : D) := \textbf{return}(x' = x)\\
&\qquad \mid \mathit{Ox}() := \textbf{return}(x)))\\
\approx_{p^{\mathsf{ow}}} &\textbf{foreach } i_{\mathrm{k}} \le n_{\mathrm{k}} \textbf{ do } r \xleftarrow{R} \mathit{seed}; (\mathit{Opk}() := \textbf{return}(\mathsf{pkgen}'(r))\\
&\quad \mid \textbf{foreach } i_{\mathrm{f}} \le n_{\mathrm{f}} \textbf{ do } x \xleftarrow{R} D; (\mathit{Oy}() := \textbf{return}(\mathsf{f}'(\mathsf{pkgen}'(r), x))\\
&\qquad \mid \textbf{foreach } i_1 \le n_1 \textbf{ do } \mathit{Oeq}(x' : D) :=\\
&\qquad\quad \textbf{if defined}(k) \textbf{ then return}(x' = x) \textbf{ else return}(\mathsf{false})\\
&\qquad \mid \mathit{Ox}() := k \leftarrow \mathsf{mark}; \textbf{return}(x)))
\end{aligned}
\tag{3}
$$

Fig. 1. Definition of one-wayness

Indeed, the event invert 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** invert. We prove in Appendix B.2 the equivalence of Figure 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. In this equivalence, the function symbols $\mathsf{pkgen}' : \mathit{seed} \rightarrow \mathit{pkey}$ and $\mathsf{f}' : \mathit{pkey} \times D \rightarrow D$ are such that the functions associated to the primed symbols $\mathsf{pkgen}', \mathsf{f}'$ are equal to the functions associated to their corresponding unprimed symbol $\mathsf{pkgen}, \mathsf{f}$, respectively. 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.

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 oracle $\mathit{Opk}[i_{\mathrm{k}}]$ publishes the public key $\mathsf{pkgen}(r[i_{\mathrm{k}}])$. The second group of oracles first picks a new $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$, and then makes available three oracles: $\mathit{Oy}[i_{\mathrm{k}}, i_{\mathrm{f}}]$ returns the image of $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ by f, $\mathit{Oeq}[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]$ returns true when it receives $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ as argument, and $\mathit{Ox}[i_{\mathrm{k}}, i_{\mathrm{f}}]$ returns $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$ itself. The one-wayness property guarantees that when $\mathit{Ox}[i_{\mathrm{k}}, i_{\mathrm{f}}]$ has not been called, the adversary has little chance of finding $x[i_{\mathrm{k}}, i_{\mathrm{f}}]$, so $\mathit{Oeq}[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]$ returns false. Therefore, we can replace the left-hand side of the equivalence with its right-hand side, in which $\mathit{Ox}[i_{\mathrm{k}}, i_{\mathrm{f}}]$ records that it has been called by defining $k[i_{\mathrm{k}}, i_{\mathrm{f}}]$, and $\mathit{Oeq}[i_{\mathrm{k}}, i_{\mathrm{f}}, i_1]$ always returns false when $k[i_{\mathrm{k}}, i_{\mathrm{f}}]$ is not defined, that is, when $\mathit{Ox}[i_{\mathrm{k}}, i_{\mathrm{f}}]$ has not been called.

In the left-hand side of the equivalences used to specify primitives, the oracles must consist of a single return instruction. This restriction 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 such oracles: we just need to recognize terms that occur as a result of these oracles. That would be much more difficult with general processes.)

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

$$\forall r : \mathit{seed}, \forall x : D, \mathsf{invf}(\mathsf{skgen}(r), \mathsf{f}(\mathsf{pkgen}(r), x)) = x \tag{4}$$

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

$$\forall pk : \mathit{pkey}, \forall x : D, \forall x' : D, (\mathsf{f}(pk, x) = \mathsf{f}(pk, x')) = (x = x') \tag{5}$$

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

$$
\begin{aligned}
&\textbf{foreach } i_{\mathrm{k}} \leq n_{\mathrm{k}} \textbf{ do } r \stackrel{R}{\leftarrow} \mathit{seed}; (\mathit{Opk}() := \textbf{return}(\mathsf{pkgen}(r)) \\
&\quad \mid \textbf{foreach } i_{\mathrm{f}} \leq n_{\mathrm{f}} \textbf{ do } x \stackrel{R}{\leftarrow} D; (\mathit{Oant}() := \textbf{return}(\mathsf{invf}(\mathsf{skgen}(r), x)) \\
&\qquad\qquad \mid \mathit{Oim}() := \textbf{return}(x))) \\
\approx_0 &\textbf{ foreach } i_{\mathrm{k}} \leq n_{\mathrm{k}} \textbf{ do } r \stackrel{R}{\leftarrow} \mathit{seed}; (\mathit{Opk}() := \textbf{return}(\mathsf{pkgen}(r)) \\
&\quad \mid \textbf{foreach } i_{\mathrm{f}} \leq n_{\mathrm{f}} \textbf{ do } x \stackrel{R}{\leftarrow} D; (\mathit{Oant}() := \textbf{return}(x) \\
&\qquad\qquad \mid \mathit{Oim}() := \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), x))))
\end{aligned}
\tag{6}
$$

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

3.2 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.

We consider here *(existential) unforgeability under adaptive chosen-message attack* (UF-CMA) [23], that is, the attacker can ask the signer to sign any message of its choice, in an adaptive way, and has to provide a signature on a new message. In its answer, there is indeed 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

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

As above, we denote by $\mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{S}}(n_{\mathrm{s}}, \ell, t)$ 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. Let *pkey*, *skey*, and *signature* the types of public keys, secret keys, and signatures respectively. A signature scheme is defined as a set of four 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 $\mathsf{UF\text{-}CMA}$, the following process LR executes **event** forge with probability at most $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \ell, t)$ in the presence of a context that runs in time t, where

$$\begin{aligned} LR = Ogen() := r \stackrel{R}{\leftarrow} seed; pk \leftarrow \mathsf{pkgen}(r); sk \leftarrow \mathsf{skgen}(r); \mathbf{return}(pk);\\ (\mathbf{foreach}\ i_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{do}\ OS(m : \mathit{bitstring}) := \mathbf{return}(\mathsf{sign}(m, sk))\\ \mid OT(m' : \mathit{bitstring}, s : \mathit{signature}) := \mathbf{if}\ \mathsf{verify}(m', pk, s)\ \mathbf{then}\\ \mathbf{find}\ u_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{suchthat}\ (\mathbf{defined}(m[u_{\mathrm{s}}]) \wedge m' = m[u_{\mathrm{s}}])\\ \mathbf{then\ end\ else\ event}\ \mathsf{forge}) \end{aligned} \tag{7}$$

and ℓ is the maximum length of m and m'. This is indeed clear since **event** forge is raised if a signature is accepted (by the verification algorithm), while the signing algorithm has not been called on the signed message.

4 Examples

4.1 FDH Signature

The Full-Domain Hash (FDH) signature scheme [9] is defined as follows: Let $\mathsf{pkgen}, \mathsf{skgen}, \mathsf{f}, \mathsf{invf}$ 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 $\mathsf{sign}(m, sk) = \mathsf{invf}(sk, \mathsf{hash}(m))$, and the verification algorithm is $\mathsf{verify}(m', pk, s) = (\mathsf{f}(pk, s) = \mathsf{hash}(m'))$. 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 (3), (4), (5), and (6) as detailed in Section 3.1 and the formalization of a hash function in the random oracle model:

$$\begin{aligned} &\mathbf{foreach}\ i_{\mathrm{h}} \leq n_{\mathrm{h}}\ \mathbf{do}\ OH(x : \mathit{bitstring}) := \mathbf{return}(\mathsf{hash}(x))\ [\mathit{all}]\\ \approx_0\ &\mathbf{foreach}\ i_{\mathrm{h}} \leq n_{\mathrm{h}}\ \mathbf{do}\ OH(x : \mathit{bitstring}) :=\\ &\quad \mathbf{find}\ u \leq n_{\mathrm{h}}\ \mathbf{suchthat}\ (\mathbf{defined}(x[u], r[u]) \wedge x = x[u])\ \mathbf{then\ return}(r[u])\\ &\quad \mathbf{else}\ r \stackrel{R}{\leftarrow} D; \mathbf{return}(r) \end{aligned} \tag{8}$$

This equivalence expresses that we can replace a call to a hash function with a random oracle, that is, an oracle 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 $[\mathit{all}]$ on the first line of (8) instructs the prover to replace all occurrences of hash in the game.)

Second, the input file contains as initial game the process G_0 of Section 2.1. As detailed in Section 3.2, this game corresponds to the definition of security of the FDH signature scheme (7). 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** forge is executed in this game in the presence of a context that runs in time t: $p(t) = \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(qS, \ell, t + t_H) \geq \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(qS, \ell, t)$ 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** forge with probability $p(t) \leq (qH + qS + 1)\mathsf{Succ}_{\mathcal{P}}^{\mathsf{ow}}(t + (qH + qS)t_{\mathsf{f}} + (3qS + 2qH + qS^2 + 2qSqH + qH^2)t_{\mathsf{eq}}(\ell))$ where ℓ is the maximum length of a bit-string in m, m', or x and $t_{\mathsf{eq}}(\ell)$ is the time of a comparison between bit-strings of length at most ℓ. (Evaluating a **find** implies evaluating the condition of the

find for each value of the indexes, so here the lookup in an array of size n of bit-strings of length ℓ is considered as taking time $n \times t_{\mathsf{eq}}(\ell)$, although there are in fact more efficient algorithms for this particular case of array lookup.) If we ignore the time of bit-string comparisons, we obtain the usual upper-bound [10] $(qH + qS + 1)\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + (qH + qS)t_{\mathsf{f}})$. 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. 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. The prover has been implemented in Ocaml and contains 14800 lines of code.

We sketch here the main proof steps. Starting from the initial game G_0 given in Section 2.1, the prover tries to apply all observational equivalences it has as hypotheses, that is here, (3), (6), and (8). It succeeds applying the security of the hash function (8), so it transforms the game accordingly, by replacing the left-hand side with the right-hand side of the equivalence. Each call to hash is then replaced with a lookup in the arguments of all calls to hash. 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.

The obtained game is then simplified. In particular, when the argument m' of OT is found in the arguments m of the call to hash in OS, the **find** in OT always succeeds, so its **else** branch can be removed (that is, when m' has already been passed to the signature oracle, it is not a forgery).

Then, the prover tries to apply an observational equivalence. All transformations fail, but when applying (6), the game contains $\mathsf{invf}(sk, y)$ while (6) expects $\mathsf{invf}(\mathsf{skgen}(r), y)$, 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 $sk \leftarrow \mathsf{skgen}(r)$. The transformation (6) is then retried. It now succeeds, which leads to replacing r_j with $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ and $\mathsf{invf}(\mathsf{skgen}(r), r_j)$ with r_j, where r_j represents the random numbers that are the result of the random oracle. (The term $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ can then be computed by oracle Oy of (3) and r_j can be computed by Ox.) More generally, in our prover, when a transformation $\mathcal{T}$ fails, it may return transformations $\mathcal{T}'$ to apply in order to enable $\mathcal{T}$ [14, Section 5]. In this case, the prover applies the suggested transformations $\mathcal{T}'$ and retries the transformation $\mathcal{T}$.

The obtained game is then simplified. In particular, by injectivity of f (5), the prover replaces 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 test $s = r_j$ can then be computed by oracle Oeq of (3).)

The prover then tries to apply an observational equivalence. It succeeds using the definition of one-wayness (3). This transformation leads to replacing $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ with $\mathsf{f}'(\mathsf{pkgen}'(r), r_j)$, r_j with $k_j \leftarrow \mathsf{mark}; r_j$, and $s = r_j$ with **find** $u_j \leq N$ **suchthat** $(\mathbf{defined}(k_j[u_j]) \wedge \mathsf{true})$ **then** $s = r_j$ **else** false. The difference of probability is $p^{\mathsf{ow}}(t + t') = n_{\mathrm{k}} \times n_{\mathrm{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (n_{\mathrm{k}} n_{\mathrm{f}} - 1)t_{\mathsf{f}} + (n_{\mathrm{k}} - 1)t_{\mathsf{pkgen}}) = (qH + qS + 1)\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (qH + qS)t_{\mathsf{f}})$ where $n_{\mathrm{k}} = 1$ is the number of key pairs considered, $n_{\mathrm{f}} = qH + qS + 1$ is the number of antecedents of f, and $t' = (3qS + 2qH + qS^2 + 2qSqH + qH^2)t_{\mathsf{eq}}(\ell)$ is the runtime of the context put around the equivalence (3).

Finally, the obtained game is simplified again. Thanks to some equational reasoning, the prover manages to show that the **find** in OT always succeeds, so its **else** branch can be removed. The prover then detects that the forge event cannot be executed in the resulting game, so the desired property is proved, and the probability that forge is executed in the initial game is the sum of the differences of probability between games of the sequence, which here comes only from the application of one-wayness (3).

4.2 Encryption Schemes

Besides proving the security of many protocols [14], we have also used our prover for proving other cryptographic schemes. For example, our prover can show that the basic Bellare-Rogaway construction [9] without redundancy (*i.e.* $\mathcal{E}(m, r) = \mathsf{f}(r) \| \mathsf{hash}(r) \,\mathsf{xor}\, m$) is IND-CPA, with the following manual proof:

`crypto hash`	apply the security of hash (8)
`remove_assign binder pk`	remove assignments to pk

`crypto f r`	apply the security of f (3) with random seed r
`crypto xor *`	apply the security of xor as many times as possible
`success`	check that the desired property is proved

These manual indications are necessary because (3) can also be applied without removing the assignments to pk, but with different results: $\mathsf{f}(pk, x)$ is computed by applying f to the results of oracles Opk and Ox if assignments to pk are not removed, and by oracle Oy if assignments to pk are removed.

With similar manual indications, it can show that the enhanced variant with redundancy $\mathcal{E}(m, r) = \mathsf{f}(r)\|\mathsf{hash}(r) \mathbin{\mathsf{xor}} m\|\mathsf{hash}'(\mathsf{hash}(r) \mathbin{\mathsf{xor}} m, r)$ is IND-CCA2. With an improved treatment of the equational theory of xor, we believe that it could also show that $\mathcal{E}(m, r) = \mathsf{f}(r)\|\mathsf{hash}(r) \mathbin{\mathsf{xor}} m\|\mathsf{hash}'(m, r)$ is IND-CCA2.

5 Conclusion

We have presented a new tool to automatically prove the security of both cryptographic primitives and cryptographic protocols. As usual, assumptions and expected security notions have to be stated. For the latter, specifications are quite similar to the usual definitions, where a "bad" event has to be shown to be unlikely. However, the former may seem more intricate, since it has to be specified as an observational equivalence. Anyway, this has to be done only once for all proofs, and several specifications have already been given in [13–15]: one-wayness, UF-CMA signatures, UF-CMA message authentication codes, IND-CPA symmetric stream ciphers, IND-CPA and IND-CCA2 public-key encryption, hash functions in the random oracle model, xor, with detailed proofs for the first three. Thereafter, the protocol/scheme itself has to be specified, but the syntax is quite close to the notations classically used in cryptography. Eventually, the prover provides the sequence of transformations, and thus of games, which lead to a final experiment (indistinguishable from the initial one) in which the "bad" event never appears. Since several paths may be used for such a sequence, the user is allowed (but does not have) to interact with the prover, in order to make it follow a specific sequence. Of course, the prover will accept only if the sequence is valid. Contrary to most of the formal proof techniques, the failure of the prover does not lead to an attack. It just means that the prover did not find an appropriate sequence of games.

Acknowledgments We thank Jacques Stern for initiating our collaboration on this topic and the anonymous reviewers for their helpful comments. 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 P. Laud. A mechanized, cryptographically sound type inference checker. In *Workshop on Formal and Computational Cryptography (FCC'06)*, July 2006. To appear.
3. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *CSFW'04*. IEEE, June 2004.
4. M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *26th IEEE Symposium on Security and Privacy*, pages 171–182. IEEE, May 2005.
5. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *CCS'03*, pages 220–230. ACM, Oct. 2003.
6. 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.
7. 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.
8. M. Bellare. Practice-Oriented Provable Security. In *ISW '97*, LNCS 1396. Springer, 1997.
9. M. Bellare and P. Rogaway. Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols. In *CCS'93*, pages 62–73. ACM Press, 1993.

10. 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, 1996.
11. M. Bellare and P. Rogaway. The Game-Playing Technique and its Application to Triple Encryption, 2004. Cryptology ePrint Archive 2004/331.
12. B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, May 2004.
13. 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`.
14. B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, May 2006.
15. B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. Cryptology ePrint Archive, Report 2006/069, Feb. 2006. Available at `http://eprint.iacr.org/2006/069`.
16. 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`.
17. 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`.
18. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP'05*, LNCS 3444, pages 157–171. Springer, Apr. 2005.
19. 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.
20. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, November 1976.
21. D. Dolev and A. C. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
22. S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
23. 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.
24. S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, June 2005. Available at `http://eprint.iacr.org/2005/181`.
25. 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.
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. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1–3):118–164, Mar. 2006.
34. M. Naor and M. Yung. Universal One-Way Hash Functions and Their Cryptographic Applications. In *STOC'89*, pages 33–43. ACM Press, 1989.
35. 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, 1992.
36. 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.
37. V. Shoup. OAEP Reconsidered. *Journal of Cryptology*, 15(4):223–249, September 2002.
38. V. Shoup. Sequences of games: a tool for taming complexity in security proofs, 2004. Cryptology ePrint Archive 2004/332.
39. C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *CSFW'06*. IEEE, July 2006. To appear.

40. S. Tarento. Machine-checked security proofs of cryptographic signature schemes. In *ESORICS'05*, LNCS 3679, pages 140–158. Springer, Sept. 2005.

Appendix

A Syntax of the Process Calculus

$M, N ::=$	terms
$\quad i$	replication index
$\quad x[M_1, \ldots, M_m]$	variable access
$\quad f(M_1, \ldots, M_m)$	function application
$Q ::=$	oracle definitions
$\quad 0$	nil
$\quad Q \mid Q'$	parallel composition
$\quad$ **foreach** $i \leq n$ **do** Q	n copies of Q in parallel
$\quad$ **newOracle** $O; Q$	restriction for oracles
$\quad O[i_1, \ldots, i_m](x_1[i_1, \ldots, i_m] : T_1, \ldots, x_k[i_1, \ldots, i_m] : T_k) := P$	oracle definition
$P ::=$	oracle body
$\quad$ **return**$(N_1, \ldots, N_k); Q$	return
$\quad$ **end**	end
$\quad x[i_1, \ldots, i_m] \stackrel{R}{\leftarrow} T; P$	random number generation
$\quad x[i_1, \ldots, i_m] : T \leftarrow M; P$	assignment
$\quad (x_1[i_1, \ldots, i_m] : T_1, \ldots, x_{k'}[i_1, \ldots, i_m] : T_{k'}) \leftarrow O[M_1, \ldots, M_l](N_1, \ldots, N_k); P$ **else** P'	oracle call
$\quad$ **if defined**$(M_1, \ldots, M_l) \wedge M$ **then** P **else** P'	conditional
$\quad$ **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat** (**defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$) **then** P_j) **else** P	array lookup
$\quad$ **event** $e; P$	event
$C ::=$	contexts
$\quad [\,]$	hole
$\quad C \mid Q$	parallel composition
$\quad Q \mid C$	parallel composition
$\quad$ **newOracle** $O; C$	restriction for oracles

Fig. 2. Syntax of the process calculus

The syntax of our calculus is summarized in Figure 2. It distinguishes two categories of processes: oracle definitions Q consist of a set of definitions of oracles, while oracle bodies P describe the content of an oracle definition. Oracle bodies perform some computations and return a result. After returning the result, they may define new oracles. (An oracle definition Q follows the **return**$(N_1, \ldots, N_k)$ instruction.)

The nil oracle definition 0 defines no oracle. The construct **newOracle** $O; Q$ hides oracle O outside Q; oracle O can be called only inside Q. The other constructs for oracle definitions have been presented in Section 2.1.

The oracle call $(x_1[i_1, \ldots, i_m] : T_1, \ldots, x_{k'}[i_1, \ldots, i_m] : T_{k'}) \leftarrow O[M_1, \ldots, M_l](N_1, \ldots, N_k); P$ **else** P' calls oracle $O[M_1, \ldots, M_l]$ with arguments $N_1, \ldots, N_k$. When this oracle returns a result by **return**$(N'_1, \ldots, N'_{k'})$, this result is stored in $x_1[i_1, \ldots, i_m]$, …, $x_{k'}[i_1, \ldots, i_m]$ and the process executes P. When the oracle $O[M_1, \ldots, M_l]$ terminates by **end**, the process executes P'. (Returning a result by **return** corresponds to the normal termination of the oracle O, while terminating with **end** corresponds to abnormal termination.)

This paper	[13, 14]
foreach $i \leq n$ **do** Q	$!^{i \leq n} Q$
newOracle $O; Q$	**newChannel** $c; Q$
$O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k) := P$	$c[\widetilde{i}](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k); P$
$\mathbf{return}(M_1, \dots, M_k); Q$	$\overline{c[\widetilde{i}]}\langle M_1, \dots, M_k \rangle; Q$
end	$\overline{yield}\langle\rangle$
$x[\widetilde{i}] \xleftarrow{R} T; P$	**new** $x[\widetilde{i}] : T; P$
$x[\widetilde{i}] : T \leftarrow M; P$	**let** $x[\widetilde{i}] : T = M$ **in** P
In equivalences that define security assumptions	
$O(x_1 : T_1, \dots, x_k : T_k) := \mathbf{return}(M)$	$(x_1 : T_1, \dots, x_k : T_k) \rightarrow M$

Fig. 3. Differences of syntax with [13, 14]

The general syntax of an array lookup is **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat** $(\mathbf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j)$ **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 $\mathbf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ 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.)

The process **if** $\mathbf{defined}(M_1, \dots, M_l) \wedge M$ **then** P **else** P' is syntactic sugar for **find suchthat** $(\mathbf{defined}(M_1, \dots, M_l) \wedge M)$ **then** P **else** P'.

To lighten notations, $\wedge\,\mathsf{true}$ and $\mathbf{defined}() \wedge$ may be omitted in conditions of **if** and **find**. Moreover, **else end**, a trailing 0, or a trailing **end** may be omitted. When oracle O returns nothing, the oracle call to oracle O can be abbreviated $O[M_1, \dots, M_l](N_1, \dots, N_k); P$ **else** P'. Types may be omitted when they can be inferred. Some array indexes can be omitted: when x is defined under **foreach** $i_1 \leq n_1 \dots$ **foreach** $i_m \leq n_m$, x is always an array with indexes $i_1, \dots, i_m$, so we abbreviate all occurrences of $x[i_1, \dots, i_m]$ by x, and more generally if x is defined under **foreach** $i_1 \leq n_1 \dots$ **foreach** $i_m \leq n_m$ and used under **foreach** $i_1 \leq n_1 \dots$ **foreach** $i_k \leq n_k$ $(k \leq m)$, we abbreviate $x[i_1, \dots, i_k, u_{k+1}, \dots, u_m]$ by $x[u_{k+1}, \dots, u_m]$. Similarly, an oracle definition $O[i_1, \dots, i_m](\dots) := P$ under **foreach** $i_1 \leq n_1 \dots$ **foreach** $i_m \leq n_m$ is abbreviated $O(\dots) := P$.

In the equivalences that serve as security assumptions in the prover, we also write **foreach** $i \leq n$ **do** $x_1 \xleftarrow{R} T_1; \dots x_m \xleftarrow{R} T_m; Q$ as an abbreviation for **foreach** $i \leq n$ **do** $O() := x_1 \xleftarrow{R} T_1; \dots x_m \xleftarrow{R} T_m; \mathbf{return}; Q$, where O is a fresh oracle name. (The same oracle names are used in both sides of the equivalences.)

A context is a process with a hole. In this paper, we consider only evaluation contexts, generated by the grammar given at the bottom of Figure 2 and that do not contain events.

The syntax used in this paper differs from the syntax used in previous papers [13, 14], to make it closer to the standard syntax of cryptographic games. The differences are summarized in Figure 3. Previous papers [13, 14] used *channels* c instead of oracles O. An oracle call then corresponds to two communications on channels: in the first communication, the caller sends the arguments of the oracle to the callee, which receives them in an input process; in the second communication, the callee sends the result of the oracle to the caller (or replies on a special channel *yield* when the oracle terminates with **end**). The oracle calls never occur in processes given to the prover, so we just give in Figure 3 the correspondence between oracle definitions and inputs on the one hand, and between returns and outputs on the other hand. Accordingly, [13, 14] also used **newChannel** instead of **newOracle**, the "input processes" of [13, 14] correspond to oracle definitions, and the "output processes" of [13, 14] correspond to oracle bodies. As shown in Figure 3, [13, 14] also used a different syntax for copies of oracles, random number generation, and assignments, with exactly the same meaning. In equivalences that define security assumptions, [13, 14] used functions $(x_1 : T_1, \dots, x_k : T_k) \to M$, while this paper uses oracle definitions $O(x_1 : T_1, \dots, x_k : T_k) := \mathbf{return}(M)$.

B Manual Proofs

Before proving the correctness of our formalizations of one-wayness and of unforgeability, let us present a few simple lemmas, which will be used in these proofs.

B.1 Preliminary Lemmas

The next lemma is Shoup's lemma [37], where we consider a process Q which contains an event e (e.g. a "bad" event). In this lemma, the notation $Q\{P'/P\}$ means that, in the process Q, we substitute any occurrence of the subprocess P by the process P'.

Lemma 2. *If Q executes event e with probability at most p, then we have $Q\{P'/\mathbf{event}\ e.P\} \approx_p Q\{P''/\mathbf{event}\ e.P\}$.*

We denote by $n \times Q$ the process obtained by adding **foreach** $i \leq n$ **do** 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 oracles 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 event e, and simulating all other copies of Q.

Lemma 3. *If Q executes event e with probability at most p and Q runs in time t_Q, then $n \times Q$ executes event e with probability at most p' where $p'(t) = n \times p(t + (n-1)t_Q)$.*

B.2 Proof of the Definition of One-wayness as an Equivalence

Proof (of (3)). We expand the abbreviations **foreach** $i_{\mathsf{k}} \leq n_{\mathsf{k}}$ **do** $r \stackrel{R}{\leftarrow} \mathit{seed}$ into **foreach** $i_{\mathsf{k}} \leq n_{\mathsf{k}}$ **do** $\mathit{Ogr}() := r \stackrel{R}{\leftarrow} \mathit{seed}; \mathbf{return}$ and **foreach** $i_{\mathsf{f}} \leq n_{\mathsf{f}}$ **do** $x \stackrel{R}{\leftarrow} D$ into **foreach** $i_{\mathsf{f}} \leq n_{\mathsf{f}}$ **do** $\mathit{Ogx}() := x \stackrel{R}{\leftarrow} D; \mathbf{return}$, and rename the oracle Oeq of (3) into Oeq', in order to avoid confusion with the oracle Oeq of the process LR defined in Section 3.1. Then, in order to prove (3), we show that $L \approx_{p^{\mathsf{ow}}} R$, where

$$
\begin{aligned}
L = {} & \mathbf{foreach}\ i_{\mathsf{k}} \leq n_{\mathsf{k}}\ \mathbf{do}\ \mathit{Ogr}() := r \stackrel{R}{\leftarrow} \mathit{seed}; \mathbf{return}; \\
& \quad (\mathit{Opk}() := \mathbf{return}(\mathsf{pkgen}(r)) \\
& \quad \mid \mathbf{foreach}\ i_{\mathsf{f}} \leq n_{\mathsf{f}}\ \mathbf{do}\ \mathit{Ogx}() := x \stackrel{R}{\leftarrow} D; \mathbf{return}; \\
& \quad\quad (\mathit{Oy}() := \mathbf{return}(\mathsf{f}(\mathsf{pkgen}(r), x)) \\
& \quad\quad \mid \mathbf{foreach}\ i_1 \leq n_1\ \mathbf{do}\ \mathit{Oeq}'(x' : D) := \mathbf{return}(x' = x) \\
& \quad\quad \mid \mathit{Ox}() := \mathbf{return}(x))) \\
R = {} & \mathbf{foreach}\ i_{\mathsf{k}} \leq n_{\mathsf{k}}\ \mathbf{do}\ \mathit{Ogr}() := r \stackrel{R}{\leftarrow} \mathit{seed}; \mathbf{return}; \\
& \quad (\mathit{Opk}() := \mathbf{return}(\mathsf{pkgen}'(r)) \\
& \quad \mid \mathbf{foreach}\ i_{\mathsf{f}} \leq n_{\mathsf{f}}\ \mathbf{do}\ \mathit{Ogx}() := x \stackrel{R}{\leftarrow} D; \mathbf{return}; \\
& \quad\quad (\mathit{Oy}() := \mathbf{return}(\mathsf{f}'(\mathsf{pkgen}'(r), x)) \\
& \quad\quad \mid \mathbf{foreach}\ i_1 \leq n_1\ \mathbf{do}\ \mathit{Oeq}'(x' : D) := \\
& \quad\quad\quad\quad \mathbf{if}\ \mathbf{defined}(k)\ \mathbf{then}\ \mathbf{return}(x' = x)\ \mathbf{else}\ \mathbf{return}(\mathsf{false}) \\
& \quad\quad \mid \mathit{Ox}() := k \leftarrow \mathsf{mark}; \mathbf{return}(x)))
\end{aligned}
$$

Let

$$\begin{aligned}
LR' = {} & Ogr() := r \stackrel{R}{\leftarrow} seed; pk \leftarrow \mathsf{pkgen}(r); \mathbf{return}; (Opk() := \mathbf{return}(pk) \\
& \quad \mid \mathbf{foreach}\ i_{\mathsf{f}} \le n_{\mathsf{f}}\ \mathbf{do}\ Ogx() := x \stackrel{R}{\leftarrow} D; y \leftarrow \mathsf{f}(pk, x); \mathbf{return}; \\
& \qquad (Oy() := \mathbf{return}(y) \\
& \qquad \mid \mathbf{foreach}\ i_1 \le n_1\ \mathbf{do}\ Oeq'(x' : D) := \\
& \qquad\quad \mathbf{if}\ \mathbf{defined}(k)\ \mathbf{then}\ \mathbf{return}(x' = x)\ \mathbf{else} \\
& \qquad\quad \mathbf{if}\ x' = x\ \mathbf{then}\ \mathbf{event}\ \mathsf{invert}\ \mathbf{else}\ \mathbf{return}(\mathsf{false}) \\
& \qquad \mid Ox() := k \leftarrow \mathsf{mark}; \mathbf{return}(x)))
\end{aligned}$$

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

$$\begin{aligned}
C_a = {} & \mathbf{newOracle}\ Ogen; \mathbf{newOracle}\ Oeq; ([\,] \\
& \mid Ogr() := (pk, y) \leftarrow Ogen(); \mathbf{return}; \\
& \quad (Opk() := \mathbf{return}(pk) \\
& \quad \mid \mathbf{foreach}\ i_{\mathsf{f}} \le n_{\mathsf{f}}\ \mathbf{do}\ Ogx() := \\
& \qquad \mathbf{if}\ i_{\mathsf{f}} = a\ \mathbf{then}\ \mathbf{return}; \\
& \qquad\quad (Oy() := \mathbf{return}(y) \\
& \qquad\quad \mid \mathbf{foreach}\ i_1 \le n_1\ \mathbf{do}\ Oeq'(x' : D) := Oeq(x')\ \mathbf{else}\ \mathbf{return}(\mathsf{false})) \\
& \qquad \mathbf{else}\ x \stackrel{R}{\leftarrow} D; \mathbf{return}; \\
& \qquad\quad (Oy() := \mathbf{return}(\mathsf{f}(pk, x)) \\
& \qquad\quad \mid \mathbf{foreach}\ i_1 \le n_1\ \mathbf{do}\ Oeq'(x' : D) := \mathbf{return}(x' = x) \\
& \qquad\quad \mid Ox() := \mathbf{return}(x))))
\end{aligned}$$

Next, we show that LR' executes **event** invert with probability at most $n_{\mathsf{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$ in the presence of a context that runs in time t, where C_a runs in time $t_{C_a} = (n_{\mathsf{f}} - 1)t_{\mathsf{f}}$.[1] By definition of one-wayness, the process LR defined in Section 3.1 executes **event** invert with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t)$. By Lemma 1, Property 5, $C_a[LR]$ executes **event** invert with probability at most $\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t_{C_a})$. Let C be a context acceptable for LR' that runs in time t. Consider a trace of $C[LR']$ that executes **event** invert. Let $a \in [1, n_{\mathsf{f}}]$ such that the first time **event** invert is executed in this trace, $i_{\mathsf{f}} = a$. Then the prefix of this trace up to the point at which it executes **event** invert 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 oracle Ogr is called, LR' picks a new seed r. Correspondingly, $C_a[LR]$ also picks a seed r_0 by calling oracle $Ogen$ of LR. It also chooses a random value x_0, so a single configuration of LR' of 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 $C_a[LR]$ return an empty message.
- When oracle Opk is called, LR' returns $\mathsf{pkgen}(r)$. Correspondingly, $C_a[LR]$ returns the public key $pk = \mathsf{pkgen}(r_0)$.
- When oracle $Ogx[i_{\mathsf{f}}]$ is called, LR' picks a random value $x[i_{\mathsf{f}}]$. Correspondingly, $C_a[LR]$ either picks a random value $x[i_{\mathsf{f}}]$ if $i_{\mathsf{f}} \neq a$, or reuses the value of x_0 previously chosen by LR when $i_{\mathsf{f}} = a$. 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 $C_a[LR]$ return an empty message.

[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.

– When oracle $Oy[i_{\mathrm{f}}]$ is called, LR' returns $\mathsf{f}(\mathsf{pkgen}(r), x[i_{\mathrm{f}}])$. Correspondingly, $C_a[LR]$ returns either $\mathsf{f}(pk, x[i_{\mathrm{f}}])$ when $i_{\mathrm{f}} \neq a$, or $y = \mathsf{f}(pk, x_0)$ when $i_{\mathrm{f}} = a$.
– When $Oeq'[i_{\mathrm{f}}, i_1](x')$ is called, LR' returns $x' = x[i_{\mathrm{f}}]$ when $i_{\mathrm{f}} \neq a$ (since it never executes **event** invert with $i_{\mathrm{f}} \neq a$ in the considered trace prefix), executes **event** invert when $x' = x[i_{\mathrm{f}}]$ and $i_{\mathrm{f}} = a$, and returns false when $x' \neq x[i_{\mathrm{f}}]$ and $i_{\mathrm{f}} = a$. (Since the considered trace prefix executes **event** invert with $i_{\mathrm{f}} = a$ at the end of this prefix, $k[a]$ is not defined at the end of this prefix, so $k[a]$ is not defined at any point in this trace prefix.) Correspondingly, $C_a[LR]$ returns $x' = x[i_{\mathrm{f}}]$ when $i_{\mathrm{f}} \neq a$ and calls LR when $i_{\mathrm{f}} = a$ in order to execute **event** invert when $x' = x_0$; when LR ends, it returns false.
– When oracle $Ox[i_{\mathrm{f}}]$ is called, we have $i_{\mathrm{f}} \neq a$ since $k[a]$ is not defined at any point in the considered trace prefix, as mentioned above, and LR' returns $x[i_{\mathrm{f}}]$. Correspondingly, $C_a[LR]$ returns $x[i_{\mathrm{f}}]$ in this case.

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

By Lemma 3, $n_{\mathrm{k}} \times LR'$ executes **event** invert 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 $t_{LR'} = t_{\mathsf{pkgen}} + n_{\mathrm{f}} t_{\mathsf{f}}$. 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 2, we have the equivalence $(n_{\mathrm{k}} \times LR')\{\mathbf{return}(\mathsf{true})/\mathbf{event}\ \mathsf{invert}\} \approx_{p^{\mathsf{ow}}} (n_{\mathrm{k}} \times LR')\{\mathbf{return}(\mathsf{false})/\mathbf{event}\ \mathsf{invert}\}$.

The process **if** $x' = x$ **then return**(true) **else return**(false) can be replaced with **return**$(x' = x)$, the process **find**... **then return**$(x' = x)$ **else return**$(x' = x)$ can be replaced with **return**$(x' = x)$, and the assignments to pk and y can be expanded without changing the behavior of the process, so $(n_{\mathrm{k}} \times LR')\{\mathbf{return}(\mathsf{true})/\mathbf{event}\ \mathsf{invert}\} \approx_0 L$. The test **if** $x' = x$ **then return**(false) **else return**(false) can be replaced with **return**(false), and the assignments to pk and y can be expanded, so $(n_{\mathrm{k}} \times LR')\{\mathbf{return}(\mathsf{false})/\mathbf{event}\ \mathsf{invert}\} \approx_0 R$. Hence $L \approx_{p^{\mathsf{ow}}} R$. □

B.3 Definition of Security of Signatures

Lemma 4. *Let* $\mathsf{skgen}' : seed \rightarrow skey$, $\mathsf{pkgen}' : seed \rightarrow pkey$, $\mathsf{sign}' : bitstring \times skey \rightarrow signature$, *and* $\mathsf{verify}' : bitstring \times pkey \times signature \rightarrow bool$ *such that the functions associated to the primed symbols* skgen', pkgen', sign', verify' *are equal to the functions associated to their corresponding unprimed symbol* skgen, pkgen, sign, verify. *We have the following equivalence:*

$$
\begin{array}{l}
\mathbf{foreach}\ i_{\mathrm{k}} \leq n_{\mathrm{k}}\ \mathbf{do}\ r \stackrel{R}{\leftarrow} seed;\\
\quad (Opk() := \mathbf{return}(\mathsf{pkgen}(r))\\
\quad \mid \mathbf{foreach}\ i_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{do}\ OS'(m : bitstring) := \mathbf{return}(\mathsf{sign}(m, \mathsf{skgen}(r))))\\
\mid \mathbf{foreach}\ i_{\mathrm{v}} \leq n_{\mathrm{v}}\ \mathbf{do}\ OV(m' : bitstring, y : pkey, s : signature) := \mathbf{return}(\mathsf{verify}(m', y, s))\ [all]\\
\approx_{p^{\mathsf{uf-cma}}}\\
\mathbf{foreach}\ i_{\mathrm{k}} \leq n_{\mathrm{k}}\ \mathbf{do}\ r \stackrel{R}{\leftarrow} seed;\\
\quad (Opk() := \mathbf{return}(\mathsf{pkgen}'(r))\\
\quad \mid \mathbf{foreach}\ i_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{do}\ OS'(m : bitstring) := \mathbf{return}(\mathsf{sign}'(m, \mathsf{skgen}'(r))))\\
\mid \mathbf{foreach}\ i_{\mathrm{v}} \leq n_{\mathrm{v}}\ \mathbf{do}\ OV(m' : bitstring, y : pkey, s : signature) :=\\
\quad \mathbf{find}\ u_{\mathrm{k}} \leq n_{\mathrm{k}}, u_{\mathrm{s}} \leq n_{\mathrm{s}}\ \mathbf{suchthat}\ (\mathbf{defined}(r[u_{\mathrm{k}}], m[u_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge\\
\qquad\quad y = \mathsf{pkgen}'(r[u_{\mathrm{k}}]) \wedge m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}] \wedge \mathsf{verify}'(m', y, s))\ \mathbf{then}\ \mathbf{return}(\mathsf{true})\ \mathbf{else}\\
\quad \mathbf{find}\ u_{\mathrm{k}} \leq n_{\mathrm{k}}\ \mathbf{suchthat}\ (\mathbf{defined}(r[u_{\mathrm{k}}]) \wedge y = \mathsf{pkgen}'(r[u_{\mathrm{k}}]))\ \mathbf{then}\ \mathbf{return}(\mathsf{false})\ \mathbf{else}\\
\quad \mathbf{return}(\mathsf{verify}(m', y, s))
\end{array}
$$

where $p^{\mathsf{uf-cma}}(t) = n_{\mathrm{k}} \times \mathsf{Succ}^{\mathsf{uf-cma}}_{\mathsf{S}}(n_{\mathrm{s}}, \max(\ell_{\mathrm{s}}, \ell_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)(t_{\mathsf{pkgen}} + t_{\mathsf{skgen}} + n_{\mathrm{s}} t_{\mathsf{sign}}(\ell_{\mathrm{s}})) + (n_{\mathrm{k}} + n_{\mathrm{v}} - 1)(t_{\mathsf{verify}}(\ell_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{s}}, \ell_{\mathrm{v}})) + n_{\mathrm{v}} t_{\mathbf{find}}(n_{\mathrm{k}}, \ell_{pkey}))$*;* t_{pkgen}*,* t_{skgen}*,* $t_{\mathsf{sign}}(\ell)$*,* $t_{\mathsf{verify}}(\ell)$ *are the times for one evaluation of* pkgen, skgen, sign, verify *respectively, with a message of length at most* ℓ*;* $t_{\mathbf{find}}(n, \ell)$ *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; $\ell_{\mathrm{s}} = \max_{i_{\mathrm{k}}\in[1,n_{\mathrm{k}}],i_{\mathrm{s}}\in[1,n_{\mathrm{s}}]} \mathrm{length}(m[i_{\mathrm{k}}, i_{\mathrm{s}}])$; and $\ell_{\mathrm{v}} = \max_{i_{\mathrm{v}}\in[1,n_{\mathrm{v}}]} \mathrm{length}(m'[i_{\mathrm{v}}])$.

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, after expanding the abbreviation **foreach** $i_{\mathrm{k}} \leq n_{\mathrm{k}}$ **do** $r \xleftarrow{R} seed$ into **foreach** $i_{\mathrm{k}} \leq n_{\mathrm{k}}$ **do** $Ogr() := r \xleftarrow{R} seed;$ **return**, and show that $L \approx_{p^{\mathsf{uf-cma}}} R$.

By definition of UF-CMA, the process LR defined in Section 3.2 executes **event** forge with probability at most $\mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \ell, t)$ in the presence of a context that runs in time t where ℓ is the maximum length of m and m'. By Lemma 3, $n_{\mathrm{k}} \times LR$ executes **event** forge with probability at most $n_{\mathrm{k}} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_{\mathrm{s}}, \max(\ell'_{\mathrm{s}}, \ell'_{\mathrm{v}}), t + (n_{\mathrm{k}} - 1)t_{LR})$, where $t_{LR} = t_{\mathsf{pkgen}} + t_{\mathsf{skgen}} + n_{\mathrm{s}} t_{\mathsf{sign}}(\ell'_{\mathrm{s}}) + t_{\mathsf{verify}}(\ell'_{\mathrm{v}}) + t_{\mathbf{find}}(n_{\mathrm{s}}, \ell'_{\mathrm{v}})$, $\ell'_{\mathrm{s}} = \max_{i_{\mathrm{k}}\in[1,n_{\mathrm{k}}],i_{\mathrm{s}}\in[1,n_{\mathrm{s}}]} \mathrm{length}(m[i_{\mathrm{k}}, i_{\mathrm{s}}])$, and $\ell'_{\mathrm{v}} = \max_{i_{\mathrm{k}}\in[1,n_{\mathrm{k}}]} \mathrm{length}(m'[i_{\mathrm{k}}])$.

$n_{\mathrm{k}} \times LR =$ **foreach** $i_{\mathrm{k}} \leq n_{\mathrm{k}}$ **do**
 $Ogen() := r \xleftarrow{R} seed; pk \leftarrow \mathsf{pkgen}(r); sk \leftarrow \mathsf{skgen}(r);$ **return**(pk);
  (**foreach** $i_{\mathrm{s}} \leq n_{\mathrm{s}}$ **do** $OS(m : bitstring) :=$ **return**$(\mathsf{sign}(m, sk))$
  | $OT(m' : bitstring, s : signature) :=$ **if** $\mathsf{verify}(m', pk, s)$ **then**
   find $u_{\mathrm{s}} \leq n_{\mathrm{s}}$ **suchthat** (**defined**$(m[i_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge m' = m[i_{\mathrm{k}}, u_{\mathrm{s}}]$) **then end else event** forge)

Let

$C =$ **newOracle** $Ogen$; **newOracle** OS; **newOracle** OT; ([ ]
 | **foreach** $i_{\mathrm{k}} \leq n_{\mathrm{k}}$ **do** $Ogr() := pk \leftarrow Ogen[i_{\mathrm{k}}]();$ **return**;
  ($Opk() :=$ **return**(pk)
  | **foreach** $i_{\mathrm{s}} \leq n_{\mathrm{s}}$ **do** $OS'(m : bitstring) := s \leftarrow OS[i_{\mathrm{k}}, i_{\mathrm{s}}](m);$ **return**(s))
 | **foreach** $i_{\mathrm{v}} \leq n_{\mathrm{v}}$ **do** $OV(m' : bitstring, y : pkey, s : signature) :=$
  find $u_{\mathrm{k}} \leq n_{\mathrm{k}}$ **suchthat** (**defined**$(pk[u_{\mathrm{k}}]) \wedge y = pk[u_{\mathrm{k}}]$) **then**
   if $\mathsf{verify}(m', y, s)$ **then**
    find $u_{\mathrm{s}} \leq n_{\mathrm{s}}$ **suchthat** (**defined**$(m[u_{\mathrm{k}}, u_{\mathrm{s}}]) \wedge m' = m[u_{\mathrm{k}}, u_{\mathrm{s}}]$) **then**
     return(true) **else** $OT[u_{\mathrm{k}}](m', s)$
   else return(false)
  else return$(\mathsf{verify}(m', y, s))$)

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

$n_\mathrm{v} t_{\mathbf{find}}(n_\mathrm{k}, \ell_{pkey})$, since $\ell'_\mathrm{s} \leq \ell_\mathrm{s}$ and $\ell'_\mathrm{v} \leq \ell_\mathrm{v}$. Let $p^{\mathsf{uf-cma}}(t) = n_\mathrm{k} \times \mathsf{Succ}_{\mathsf{S}}^{\mathsf{uf-cma}}(n_\mathrm{s}, \max(\ell_\mathrm{s}, \ell_\mathrm{v}), t + t')$. Let

$$
\begin{aligned}
LR'' = &\mathbf{foreach}\ i_\mathrm{k} \leq n_\mathrm{k}\ \mathbf{do}\ Ogr() := r \stackrel{R}{\leftarrow} seed; \mathbf{return};\\
&\quad (Opk() := \mathbf{return}(\mathsf{pkgen}(r))\\
&\quad \mid \mathbf{foreach}\ i_\mathrm{s} \leq n_\mathrm{s}\ \mathbf{do}\ OS'(m : bitstring) := \mathbf{return}(\mathsf{sign}(m, \mathsf{skgen}(r))))\\
&\mid \mathbf{foreach}\ i_\mathrm{v} \leq n_\mathrm{v}\ \mathbf{do}\ OV(m' : bitstring, y : pkey, s : signature) :=\\
&\quad \mathbf{find}\ u_\mathrm{k} \leq n_\mathrm{k}\ \mathbf{suchthat}\ (\mathbf{defined}(pk[u_\mathrm{k}]) \wedge y = pk[u_\mathrm{k}])\ \mathbf{then}\\
&\quad\quad \mathbf{if}\ \mathsf{verify}(m', y, s)\ \mathbf{then}\\
&\quad\quad\quad \mathbf{find}\ u_\mathrm{s} \leq n_\mathrm{s}\ \mathbf{suchthat}\ (\mathbf{defined}(m[u_\mathrm{k}, u_\mathrm{s}]) \wedge m' = m[u_\mathrm{k}, u_\mathrm{s}])\ \mathbf{then}\\
&\quad\quad\quad\quad \mathbf{return}(\mathsf{true})\ \mathbf{else\ event}\ \mathsf{forge}\\
&\quad\quad \mathbf{else\ return}(\mathsf{false})\\
&\quad \mathbf{else\ return}(\mathsf{verify}(m', y, s))
\end{aligned}
$$

By inlining the calls to oracles $Ogen$, OS, and OT, we can easily see that each prefix of a trace $C[n_\mathrm{k} \times LR]$ until the first execution of event forge can be simulated by LR'', and conversely. Indeed, when C calls $OT[u_\mathrm{k}](m', s)$, C has checked that s is a forged signature of m' under the key $pk[u_\mathrm{k}]$, so $n_\mathrm{k} \times LR$ always executes **event** forge when receiving this call. Therefore LR'' executes **event** forge with probability at most $p^{\mathsf{uf-cma}}$. By Lemma 2,

$$LR''\{\mathbf{return}(\mathsf{true})/\mathbf{event}\ \mathsf{forge}\} \approx_{p^{\mathsf{uf-cma}}} LR''\{\mathbf{return}(\mathsf{false})/\mathbf{event}\ \mathsf{forge}\}.$$

The process **find** ... **then return**(true) **else return**(true) can be replaced with **return**(true), the process **if** $\mathsf{verify}(m', y, s)$ **then return**(true) **else return**(false) can be replaced with **return**($\mathsf{verify}(m', y, s)$), and the process **find** ... **then return**($\mathsf{verify}(m', y, s)$) **else return**($\mathsf{verify}(m', y, s)$) can be replaced with **return**($\mathsf{verify}(m', y, s)$) without changing the behavior of the process, so we have the equivalence $LR''\{\mathbf{return}(\mathsf{true})/\mathbf{event}\ \mathsf{forge}\} \approx_0 L$. By reorganizing **find**s, we can prove the equivalence $LR''\{\mathbf{return}(\mathsf{false})/\mathbf{event}\ \mathsf{forge}\} \approx_0 R$. Hence $L \approx_{p^{\mathsf{uf-cma}}} R$. □

C Automatic Proof of the FDH Signature Example

In this appendix, we give detailed explanations on the proof automatically generated by our prover for the FDH 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 (8). Then each argument of a call to hash is first stored in an intermediate variable, x_{19} for m', x_{21} for m, and x_{23} 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_{19}, x_{21}, and x_{23}. 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. (In this game, the identifiers $@i_k$ are generated by the prover. We use the special character @ just to distinguish them from identifiers chosen by the user.)

$$
\begin{aligned}
&(\\
&\quad \mathbf{foreach}\ iH_{13} \leq qH\ \mathbf{do}\\
&\quad OH(x : bitstring) :=\\
&\quad x_{23} : bitstring \leftarrow x;\\
&\quad \mathbf{find}\ \ \mathbf{suchthat\ defined}(x_{19}, r_{18}) \wedge (x_{23} = x_{19})\ \mathbf{then}\\
&\quad\quad \mathbf{return}(r_{18})\\
&\quad \oplus\ @i_{29} \leq qS\ \mathbf{suchthat\ defined}(x_{21}[@i_{29}], r_{20}[@i_{29}]) \wedge (x_{23} = x_{21}[@i_{29}])\ \mathbf{then}\\
&\quad\quad \mathbf{return}(r_{20}[@i_{29}])\\
&\quad \oplus\ @i_{28} \leq qH\ \mathbf{suchthat\ defined}(x_{23}[@i_{28}], r_{22}[@i_{28}]) \wedge (x_{23} = x_{23}[@i_{28}])\ \mathbf{then}\\
&\quad\quad \mathbf{return}(r_{22}[@i_{28}])\\
&\quad \mathbf{else}
\end{aligned}
$$

$r_{22} \stackrel{R}{\leftarrow} D;$
return(r_{22})
$|$
$Ogen() :=$
$r \stackrel{R}{\leftarrow} seed;$
$pk : pkey \leftarrow \mathsf{pkgen}(r);$
$sk : skey \leftarrow \mathsf{skgen}(r);$
return$(pk);$
$($
foreach $iS_{14} \leq qS$ **do**
$OS(m : bitstring) :=$
$x_{21} : bitstring \leftarrow m;$
find **suchthat defined**$(x_{19}, r_{18}) \wedge (x_{21} = x_{19})$ **then**
return$(\mathsf{invf}(sk, r_{18}))$
$\oplus$ $@i_{27} \leq qS$ **suchthat defined**$(x_{21}[@i_{27}], r_{20}[@i_{27}]) \wedge (x_{21} = x_{21}[@i_{27}])$ **then**
return$(\mathsf{invf}(sk, r_{20}[@i_{27}]))$
$\oplus$ $@i_{26} \leq qH$ **suchthat defined**$(x_{23}[@i_{26}], r_{22}[@i_{26}]) \wedge (x_{21} = x_{23}[@i_{26}])$ **then**
return$(\mathsf{invf}(sk, r_{22}[@i_{26}]))$
else
$r_{20} \stackrel{R}{\leftarrow} D;$
return$(\mathsf{invf}(sk, r_{20}))$
$|$
$OT(m' : bitstring, s : D) :=$
$x_{19} : bitstring \leftarrow m';$
1 **find** **suchthat defined**$(x_{19}, r_{18}) \wedge (x_{19} = x_{19})$ **then**
if $(\mathsf{f}(pk, s) = r_{18})$ **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
2 $\oplus$ $@i_{25} \leq qS$ **suchthat defined**$(x_{21}[@i_{25}], r_{20}[@i_{25}]) \wedge (x_{19} = x_{21}[@i_{25}])$ **then**
3 **if** $(\mathsf{f}(pk, s) = r_{20}[@i_{25}])$ **then**
4 **find** $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
$\oplus$ $@i_{24} \leq qH$ **suchthat defined**$(x_{23}[@i_{24}], r_{22}[@i_{24}]) \wedge (x_{19} = x_{23}[@i_{24}])$ **then**
if $(\mathsf{f}(pk, s) = r_{22}[@i_{24}])$ **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
else
$r_{18} \stackrel{R}{\leftarrow} D;$
if $(\mathsf{f}(pk, s) = r_{18})$ **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
$)$
$)$

This game is automatically simplified as follows: The test at line 1 always fails since r_{18} is not defined at this point. (It is defined only in the **else** branch of the **find**.) The variables x_{19}, x_{21}, and x_{23} are substituted with their value, respectively m', m, and x. After this substitution, the values assigned to x_{19}, x_{21}, and x_{23} are no longer important, so they are replaced with constants $\mathsf{cst\_bitstring}$. (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 $u = @i_{25}$, due to the **find** are line 2. So the **else** branch of the **find** at line 4 can be removed, hence the **find** at line 4 and the three following lines can be replaced with **end**, and therefore lines 3, 4, and the three following lines can be replaced with **end**.

Then, the prover tries to apply a cryptographic transformation. All transformations fail, but when applying (6), the game contains $\mathsf{invf}(sk, y)$ while (6) expects $\mathsf{invf}(\mathsf{skgen}(r), y)$, 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 $sk : skey \leftarrow \mathsf{skgen}(r)$. The transformation (6) is then retried. It now succeeds, which leads to replacing r_j with $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ and $\mathsf{invf}(\mathsf{skgen}(r), r_j)$ with r_j. We obtain the following game:

$$
\begin{array}{l}
(\\
\quad \textbf{foreach } iH_{13} \leq qH \textbf{ do}\\
\quad OH(x : \mathit{bitstring}) :=\\
\quad x_{23} : \mathit{bitstring} \leftarrow \mathsf{cst\_bitstring};\\
\quad \textbf{find } \ \textbf{suchthat defined}(m', x_{19}, r_{18}) \wedge (x = m') \textbf{ then}\\
\quad\quad \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), r_{18}))\\
\quad \oplus\ @i_{29} \leq qS \textbf{ suchthat defined}(m[@i_{29}], x_{21}[@i_{29}], r_{20}[@i_{29}]) \wedge (x = m[@i_{29}]) \textbf{ then}\\
\quad\quad \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), r_{20}[@i_{29}]))\\
\quad \oplus\ @i_{28} \leq qH \textbf{ suchthat defined}(x[@i_{28}], x_{23}[@i_{28}], r_{22}[@i_{28}]) \wedge (x = x[@i_{28}]) \textbf{ then}\\
\quad\quad \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), r_{22}[@i_{28}]))\\
\quad \textbf{else}\\
\quad\quad r_{22} \stackrel{R}{\leftarrow} D;\\
\quad\quad \textbf{return}(\mathsf{f}(\mathsf{pkgen}(r), r_{22}))\\
|\\
\quad Ogen() :=\\
\quad r \stackrel{R}{\leftarrow} \mathit{seed};\\
\quad pk : \mathit{pkey} \leftarrow \mathsf{pkgen}(r);\\
\quad \textbf{return}(pk);\\
\quad (\\
\quad\quad \textbf{foreach } iS_{14} \leq qS \textbf{ do}\\
\quad\quad OS(m : \mathit{bitstring}) :=\\
\quad\quad x_{21} : \mathit{bitstring} \leftarrow \mathsf{cst\_bitstring};\\
\quad\quad \textbf{find } \ \textbf{suchthat defined}(m', x_{19}, r_{18}) \wedge (m = m') \textbf{ then}\\
\quad\quad\quad \textbf{return}(r_{18})\\
\quad\quad \oplus\ @i_{27} \leq qS \textbf{ suchthat defined}(m[@i_{27}], x_{21}[@i_{27}], r_{20}[@i_{27}]) \wedge (m = m[@i_{27}]) \textbf{ then}\\
\quad\quad\quad \textbf{return}(r_{20}[@i_{27}])\\
\quad\quad \oplus\ @i_{26} \leq qH \textbf{ suchthat defined}(x[@i_{26}], x_{23}[@i_{26}], r_{22}[@i_{26}]) \wedge (m = x[@i_{26}]) \textbf{ then}\\
\quad\quad\quad \textbf{return}(r_{22}[@i_{26}])\\
\quad\quad \textbf{else}\\
\quad\quad\quad r_{20} \stackrel{R}{\leftarrow} D;\\
\quad\quad\quad \textbf{return}(r_{20})\\
\quad |\\
\quad\quad OT(m' : \mathit{bitstring}, s : D) :=\\
\quad\quad x_{19} : \mathit{bitstring} \leftarrow \mathsf{cst\_bitstring};\\
\quad\quad \textbf{find } @i_{25} \leq qS \textbf{ suchthat defined}(m[@i_{25}], x_{21}[@i_{25}], r_{20}[@i_{25}]) \wedge (m' = m[@i_{25}]) \textbf{ then}\\
\quad\quad\quad \textbf{end}\\
\quad\quad \oplus\ @i_{24} \leq qH \textbf{ suchthat defined}(x[@i_{24}], x_{23}[@i_{24}], r_{22}[@i_{24}]) \wedge (m' = x[@i_{24}]) \textbf{ then}\\
\quad\quad\quad \textbf{if } (\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_{22}[@i_{24}])) \textbf{ then}\\
\quad\quad\quad \textbf{find } u \leq qS \textbf{ suchthat defined}(m[u]) \wedge (m' = m[u]) \textbf{ then}\\
\quad\quad\quad\quad \textbf{end}
\end{array}
$$

else
event forge
else
$r_{18} \stackrel{R}{\leftarrow} D;$
if $(\mathsf{f}(pk, s) = \mathsf{f}(\mathsf{pkgen}(r), r_{18}))$ **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
)
)

This game is automatically simplified as follows. In this game, it is useless to test whether $x_{23}[i]$ is defined, since when we require that $x_{23}[i]$ is defined, we also require that $r_{22}[i]$ is defined, and if $r_{22}[i]$ is defined, then $x_{23}[i]$ has been defined before. So the prover removes $x_{23}[i]$ from **defined** tests, and removes the assignments to x_{23}, which is no longer used. The situation is similar for x_{19} and x_{21}.

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 (3). This transformation leads to replacing $\mathsf{f}(\mathsf{pkgen}(r), r_j)$ with $\mathsf{f}'(\mathsf{pkgen}'(r), r_j)$, r_j with $k_j : \mathit{bitstring} \leftarrow \mathsf{mark}; r_j$, and $s = r_j$ with **if defined**(k_j) **then** $s = r_j$ **else** false. Actually, the replacement is however a bit more complicated: $k_{47}[i]$ is defined for some i when r_{18} is used, so we replace $s = r_{18}$ with a lookup that returns $s = r_{18}$ when $k_{47}[i]$ is defined for some i and false otherwise: **find** $@i_{53} \leq N$ **suchthat defined**$(k_{47}[@i_{53}])$ **then** $s = r_{18}$ **else** false. Similarly, $k_{50}[i]$ is defined when $r_{22}[@i_{26}[i]]$ is used, so we replace $s = r_{22}[@i_{24}]$ with a lookup that returns $s = r_{22}[@i_{24}]$ when $r_{22}[@i_{24}]$ is used, that is, when $k_{50}[i]$ is defined for some i such that $@i_{24} = @i_{26}[i]$, and false otherwise: **find** $@i_{56} \leq N$ **suchthat defined**$(k_{50}[@i_{56}]) \wedge (@i_{24} = @i_{26}[@i_{56}])$ **then** $s = r_{22}[@i_{24}]$ **else** false. The difference of probability is $p^{\mathsf{ow}}(t + t') = n_{\mathrm{k}} \times n_{\mathsf{f}} \times \mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (n_{\mathrm{k}} n_{\mathsf{f}} - 1)t_{\mathsf{f}} + (n_{\mathrm{k}} - 1)t_{\mathsf{pkgen}}) = (qH + qS + 1)\mathsf{Succ}^{\mathsf{ow}}_{\mathcal{P}}(t + t' + (qH + qS)t_{\mathsf{f}})$ where $n_{\mathrm{k}} = 1$ is the number of key pairs considered, $n_{\mathsf{f}} = qH + qS + 1$ is the number of antecedents of f, and $t' = (3qS + 2qH + qS^2 + 2qSqH + qH^2)t_{\mathsf{eq}}(\ell)$ is the runtime of the context put around the equivalence (3). After this transformation, we obtain the following game:

(
foreach $iH_{13} \leq qH$ **do**
$OH(x : \mathit{bitstring}) :=$
find **suchthat defined**$(m', r, r_{18}) \wedge (x = m')$ **then**
return$(\mathsf{f}'(\mathsf{pkgen}'(r), r_{18}))$
$\oplus$ $@i_{29} \leq qS$ **suchthat defined**$(m[@i_{29}], r, r_{20}[@i_{29}]) \wedge (x = m[@i_{29}])$ **then**
return$(\mathsf{f}'(\mathsf{pkgen}'(r), r_{20}[@i_{29}]))$
$\oplus$ $@i_{28} \leq qH$ **suchthat defined**$(x[@i_{28}], r_{22}[@i_{28}]) \wedge (x = x[@i_{28}])$ **then**
return$(\mathsf{f}'(\mathsf{pkgen}'(r), r_{22}[@i_{28}]))$
else
$r_{22} \stackrel{R}{\leftarrow} D;$
return$(\mathsf{f}'(\mathsf{pkgen}'(r), r_{22}))$
|
$Ogen() :=$
$r \stackrel{R}{\leftarrow} \mathit{seed};$
$pk : \mathit{pkey} \leftarrow \mathsf{pkgen}'(r);$
return$(pk);$
(
foreach $iS_{14} \leq qS$ **do**
$OS(m : \mathit{bitstring}) :=$
find **suchthat defined**$(m', r_{18}) \wedge (m = m')$ **then**

3 $k_{47} : \mathit{bitstring} \leftarrow \mathsf{mark};$
$\mathbf{return}(r_{18})$
$\oplus\ @i_{27} \leq qS$ **suchthat defined**$(m[@i_{27}], r_{20}[@i_{27}]) \wedge (m = m[@i_{27}])$ **then**
$k_{48} : \mathit{bitstring} \leftarrow \mathsf{mark};$
$\mathbf{return}(r_{20}[@i_{27}])$
4 $\oplus\ @i_{26} \leq qH$ **suchthat defined**$(x[@i_{26}], r_{22}[@i_{26}]) \wedge (m = x[@i_{26}])$ **then**
5 $k_{50} : \mathit{bitstring} \leftarrow \mathsf{mark};$
$\mathbf{return}(r_{22}[@i_{26}])$
else
$r_{20} \stackrel{R}{\leftarrow} D;$
$k_{45} : \mathit{bitstring} \leftarrow \mathsf{mark};$
$\mathbf{return}(r_{20})$
|
$OT(m' : \mathit{bitstring}, s : D) :=$
find $@i_{25} \leq qS$ **suchthat defined**$(r_{20}[@i_{25}], m[@i_{25}]) \wedge (m' = m[@i_{25}])$ **then**
end
6 $\oplus\ @i_{24} \leq qH$ **suchthat defined**$(x[@i_{24}], r_{22}[@i_{24}]) \wedge (m' = x[@i_{24}])$ **then**
7 **find** $@i_{56} \leq qS$ **suchthat defined**$(k_{50}[@i_{56}]) \wedge (@i_{24} = @i_{26}[@i_{56}])$ **then**
if $(s = r_{22}[@i_{24}])$ **then**
8 **find** $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
else
9 **if** false **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
else
$r_{18} \stackrel{R}{\leftarrow} D;$
10 **find** $@i_{53} \leq qS$ **suchthat defined**$(k_{47}[@i_{53}])$ **then**
if $(s = r_{18})$ **then**
11 **find** $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
else
12 **if** false **then**
find $u \leq qS$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$ **then**
end
else
event forge
)
)

The prover then simplifies the obtained game automatically. The tests **if** false **then**... at lines 9 and 12 are obviously simplified out. The **find**s at lines 8 and 11 always succeed:

- At line 11, $k_{47}[@i_{53}]$ is defined according to the condition of the **find** at line 10. Since k_{47} is defined only at line 3, $m[@i_{53}]$ is then defined (line 1) and $m[@i_{53}] = m'$ by the condition of the **find** at line 2. So the **find** at line 11 succeeds with $u = @i_{53}$.
- At line 8, $k_{50}[@i_{56}]$ is defined by the condition of the **find** at line 7. Since k_{50} is defined only at line 5, $m[@i_{56}]$ is defined (line 1), and $m[@i_{56}] = x[@i_{26}[@i_{56}]]$ by the condition of the **find** at line 4, $@i_{26}[@i_{56}] = @i_{24}$ by the condition of the **find** at line 7, and $m' = x[@i_{24}]$ by the condition

of the **find** at line 6, so $m[@i_{56}] = x[@i_{26}[@i_{56}]] = x[@i_{24}] = m'$. So the **find** at line 8 succeeds with $u = @i_{56}$.

Therefore, the **else** branches of the **find**s at lines 8 and 11 can be removed, hence these **find**s can themselves be replaced with **end**, and therefore the test that precedes these **find**s can also be replaced with **end**.

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