Formally Certifying the Security of Digital Signature Schemes

Santiago Zanella Béguelin   Benjamin Grégoire
INRIA Sophia Antipolis - Méditerranée
Microsoft Research - INRIA Joint Centre
$\{$*Santiago.Zanella,Benjamin.Gregoire*$\}$*@inria.fr*

Gilles Barthe   Federico Olmedo
IMDEA Software
$\{$*Gilles.Barthe,Federico.Olmedo*$\}$*@imdea.org*

Abstract

We present two machine-checked proofs of the existential unforgeability under adaptive chosen-message attacks of the Full Domain Hash signature scheme. These proofs formalize the original argument of Bellare and Rogaway, and an optimal reduction by Coron that provides a tighter bound on the probability of a forgery. Both proofs are developed using CertiCrypt*, a general framework to formalize exact security proofs of cryptographic systems in the computational model. Since* CertiCrypt *is implemented on top of the* Coq *proof assistant, the proofs are highly trustworthy and can be verified independently and fully automatically.*

1. Introduction

Designing secure cryptographic systems is notoriously difficult; many systems that had been thought secure for a long time were subsequently broken, generally by attackers acting in a manner that was unpredicted by its designers. Indeed, there cannot be any empirical proof of security, and the correctness of a cryptographic system can only be established through formal arguments showing that security holds against all feasible adversaries.

Provable security [1], [2] aims to establish the correctness of cryptographic systems using rigorous techniques based on complexity theory. In a typical provable security argument, security is proved to hold against any probabilistic polynomial time (PPT) adversary by showing that an efficient way to break a cryptographic property would imply a way to solve a difficult mathematical problem with relatively little extra effort.

The game-playing technique [3]–[5] is a popular method to develop provable security proofs that is extremely valuable for structuring and managing the complexity of cryptographic proofs and has been used extensively to prove exact security in the standard and random oracle models. The game-playing technique advocates describing the interaction between the cryptographic system and the adversary as a probabilistic program, and organizing the proof as a sequence of transitions $G, A \rightarrow G', A'$ between pairs of games and events, such that the probability of event A in game G is bounded by a function of the probability of event A' in game G'. Such sequences yield an upper bound on the success probability of an adversary, provided the sequence starts with the original attack game and the event of the adversary breaking the scheme, and that one can bound the probability of the event in the last game of the sequence.

Bellare and Rogaway [3] and Halevi [6] suggested that game-playing is also well-suited for mechanical verification. Their intuition has been corroborated by Blanchet's CryptoVerif [7], an automated prover that has been used to establish exact security of cryptographic systems in the computational model. In particular, Blanchet and Pointcheval [8] used CryptoVerif to prove that Full Domain Hash (FDH), an emblematic signature scheme, is secure against existential forgery under an adaptive chosen message attack (EF-CMA), the strongest notion of security for signature schemes. The work around CryptoVerif has stirred considerable interest and shown the benefits of machine-checked verification. However, it also exposed one major weakness of CryptoVerif: it deviates from the style that is natural to cryptographers since it is difficult to recover a reductionist argument from the proof trace that the prover outputs, and even if one manages to do so, most likely the reduction will not be optimal. Indeed, only the original, suboptimal bound of Bellare and Rogaway has been proved in CryptoVerif.

Our main contribution is to provide the first formalized proof of Coron's bound for the EF-CMA security of FDH [9]; in contrast to [8], the bound we prove is optimal. We use CertiCrypt [10], [11], a general framework for certifying game-based cryptographic proofs in the Coq proof assistant. Besides its intrinsic interest, our proof demonstrates that it is possible to build fully verified exact security proofs without sacrificing tightness, and furthermore, that it is feasible to produce "certificates" for cryptographic proofs, in the form of proof objects that can be verified independently and automatically by third parties. In summary, our proof provides further evidence to support formal verification as an effective tool to increase confidence in cryptographic proofs.

Synopsis. The paper consists of two main sections: Section 2 provides a user view on CertiCrypt, and intends to complement the theoretical account given by Barthe et al. in [10]; Section 3 describes the formalizations of Bellare and Rogaway's security bound for FDH, and of the optimal bound by Coron. Section 4 provides a broader analysis of the significance of our results and of the perspectives of our enterprise.

2. A quick guide to CertiCrypt

CertiCrypt is built on top of Coq [12], a general purpose proof assistant that has been used for over two decades to formalize results in mathematics and computer science. Coq provides an expressive specification language based on the Calculus of Inductive Constructions, a higher-order dependently-typed λ-calculus in which mathematical notions can be formalized conveniently. The Coq logic distinguishes between types, of type $\mathbf{Type}$, which represent sets, and propositions, of type $\mathbf{Prop}$, which represent formulae: thus, $a : A$ is interpreted as a is an element of type A if A is a set, and as a is a proof of A if A is a proposition. In the latter case, we say that a is a proof object. Types can either be introduced by standard constructions, e.g. (generalized) function space and products, or by inductive definitions. In the remainder, we will use the inductively defined types of homogeneous and heterogeneous lists. The latter is defined as

$$\begin{array}{l}\mathbf{Inductive}\ \mathbb{L}_{\text{het}} : \mathbb{L}\ A \to \mathbf{Type} := \\ \mid \mathsf{dnil} \quad : \mathbb{L}_{\text{het}}\ \mathsf{nil} \\ \mid \mathsf{dcons} : \forall\ a\ l,\ P\ a \to \mathbb{L}_{\text{het}}\ l \to \mathbb{L}_{\text{het}}\ (a :: l).\end{array}$$

where $A : \mathbf{Type}$, $P : A \to \mathbf{Type}$, and $\mathbb{L}$ is the usual type of (homogeneous) lists. We use $A^\star$ to denote the type of A-lists (i.e. $\mathbb{L}\ A$), and $P_l^\star$ to denote the type of heterogeneous P-lists over a list l.

In addition, Coq features tools to construct proofs, including a predefined set of tactics that support most common forms of reasoning, and a language for users to build their own tactics. CertiCrypt heavily relies on reflection-based tactics, which exploit the computing abilities of Coq to reduce reasoning to computation: given a property $P : A \to \mathbf{Prop}$ over a type A, one can program a checker $f : A \to \mathbb{B}$ and prove its correctness, i.e. prove that $\forall x, f\ x = \mathsf{true} \to P\ x$. Then, in order to prove that a value a of type A satisfies P, i.e. that $P\ a$ holds, it is sufficient to check that $f\ a = \mathsf{true}$, which can be done by evaluating $f\ a$.

2.1. Formalization of games

The lower layer of CertiCrypt is a formalization of a probabilistic extension of a simple imperative language with procedure calls. The formalization is carefully crafted to exploit key features of Coq underlying type theory: it uses modules to support an extensible expression language that can be adapted according to the cryptographic system being verified, dependent types to ensure that programs are well-typed and have a total semantics, and monads to give semantics to probabilistic programs and capture the cost of executing them.

The formalization of programs uses a deep embedding, i.e. the syntax of the language is encoded within the proof assistant. The semantics of programs is given by an interpretation function that takes a program p— an element of the type of programs— and an initial state s, and returns the result of executing p starting from s. In a deterministic case, this result will be a state, whereas in the case of CertiCrypt is a probabilistic measure over states. Deep embeddings offer one tremendous advantage over shallow embeddings, in which the language used to represent programs is the same as the underlying language of the proof assistant. Namely, a deep embedding allows a high level of automation through reflection-based tactics that implement syntactic program transformations. Additionally, deep embeddings allow to formalize complexity issues neatly and to reason about programs by induction on the structure of their code. Our language is a mild generalization of the language of Bellare and Rogaway [3], in that it allows while loops whereas they only consider bounded for loops.

2.1.1. Types. CertiCrypt formalizes a typed syntax, and uses the underlying type system of Coq to ensure for free that expressions and commands are legal. In our experience, the typed syntax provides particularly useful feedback when debugging proofs. The types and expressions of the language are defined on top of a module that contains the declaration of user-defined types and operators. Formally, the set $\mathcal{T}$ of types is defined as:

$$\begin{array}{l}\textbf{Inductive } \mathcal{T} : \textbf{Type} := \\ \mid \textsf{User} : \mathcal{T}_{\text{user}} \to \mathcal{T} \\ \mid \textsf{Nat} \ : \mathcal{T} \\ \mid \textsf{Bool} : \mathcal{T} \\ \mid \textsf{List} \ : \mathcal{T} \to \mathcal{T} \\ \mid \textsf{Pair} \ : \mathcal{T} \to \mathcal{T} \to \mathcal{T}.\end{array}$$

where $\mathcal{T}_{\text{user}}$ denotes the set of user-defined types. In the proofs in the next section we will use an extension of the semantics with used-defined types for bitstrings of arbitrary length and elements of a cyclic group, obtained by instantiating $\mathcal{T}_{\text{user}}$ as

$$\begin{array}{l}\textbf{Inductive } \mathcal{T}_{\text{user}} : \textbf{Type} := \\ \mid \ \textsf{Bitstring} \ : \ \mathcal{T}_{\text{user}} \\ \mid \ \textsf{Group} \ : \ \mathcal{T}_{\text{user}}.\end{array}$$

2.1.2. Expressions. Expressions are built from a set of $\mathcal{T}$-indexed variable names $\mathcal{V}$, using operators from the core language, such as constructors for pairs and lists, and user-defined operators. All operators are declared with typing information, as specified by the functions targs and tres, that return for each operator the list of types of its arguments, and the type of its result, respectively. The $\mathcal{T}$-indexed family $\mathcal{E}$ of expressions is then defined as:

$$\begin{array}{lll}\textbf{Inductive } \mathcal{E} : \mathcal{T} \to \textbf{Type} := & & \\ \mid \textsf{Enat} & :> & \mathbb{N} \to \mathcal{E}_{\textsf{Nat}} \\ \mid \textsf{Ebool} & :> & \mathbb{B} \to \mathcal{E}_{\textsf{Bool}} \\ \mid \textsf{Evar} & :> & \forall t, \mathcal{V}_t \to \mathcal{E}_t \\ \mid \textsf{Eop} & : & \forall op, \mathcal{E}^{\star}_{(\textsf{targs}\ op)} \to \mathcal{E}_{(\textsf{tres}\ op)} \\ \mid \textsf{Eexists} & : & \forall t, \mathcal{V}_t \to \mathcal{E}_{\textsf{Bool}} \to \mathcal{E}_{(\textsf{List}\ t)} \to \mathcal{E}_{\textsf{Bool}} \\ \mid \textsf{Eforall} & : & \forall t, \mathcal{V}_t \to \mathcal{E}_{\textsf{Bool}} \to \mathcal{E}_{(\textsf{List}\ t)} \to \mathcal{E}_{\textsf{Bool}} \\ \mid \textsf{Efind} & : & \forall t, \mathcal{V}_t \to \mathcal{E}_{\textsf{Bool}} \to \mathcal{E}_{(\textsf{List}\ t)} \to \mathcal{E}_t.\end{array}$$

The first three clauses declare constructors as coercions; thanks to this mechanism it is possible to view an element of their codomain as an element of their domain, e.g. a natural number as an expression of type Nat and a variable of type t as an expression of type t. The fourth clause corresponds to the standard rule for operators; the rule requires that the types of the arguments are compatible with the declaration of the operator, as enforced by the type $\mathcal{E}^{\star}_{(\textsf{targs}\ op)}$ of heterogeneous lists of expressions. Note that op is universally quantified over a sum type $\mathcal{O}+\mathcal{O}_{\text{user}}$ where $\mathcal{O}$ is a fixed set of operators for base types, and $\mathcal{O}_{\text{user}}$ is a set of user-defined operators. In addition, expressions include useful operators on lists: Eexists, Eforall and Efind take as parameters a variable x of type t, a Boolean valued expression e that may depend on x, and an expression l of type List t and respectively:

- check whether some element a of l verifies e, when interpreting x by a;
- check whether all elements a of l verify e, when interpreting x by a;
- return the first element of the list l that verifies e, when interpreting x by a, or a default element of type t if no element is found.

Both Eexists and Efind are often used in the code of oracles, while Eforall is essential to specify invariants. Thanks to dependent types, the semantics of expressions is total, which considerably simplifies reasoning about programs.

It is worth noting that dependent types allow for rich specifications of operators. For instance, one can define a type for bitstrings of fixed length $\{0,1\}^k$, and a concatenation operator of type

$$\forall m\ n,\ \{0,1\}^m \to \{0,1\}^n \to \{0,1\}^{m+n}$$

2.1.3. Commands. Commands are built from a set of procedure names $\mathcal{P}$ indexed by the type of their arguments and return value. Formally, the sets $\mathcal{I}$ of instructions and $\mathcal{C}$ of commands are defined as:

$$\begin{array}{lll}\textbf{Inductive } \mathcal{I} : \textbf{Type} := & & \\ \mid \textsf{Assign} & : & \forall t,\ \mathcal{V}_t \to \mathcal{E}_t \to \mathcal{I} \\ \mid \textsf{Rand} & : & \forall t,\ \mathcal{V}_t \to \mathcal{D}_t \to \mathcal{I} \\ \mid \textsf{Cond} & : & \mathcal{E}_{\textsf{Bool}} \to \mathcal{C} \to \mathcal{C} \to \mathcal{I} \\ \mid \textsf{While} & : & \mathcal{E}_{\textsf{Bool}} \to \mathcal{C} \to \mathcal{I} \\ \mid \textsf{Call} & : & \forall l\ t, \mathcal{P}_{(l,t)} \to\ \mathcal{V}_t \to \mathcal{E}^{\star}_l \to \mathcal{I} \\ \textbf{where } \mathcal{C} := \mathcal{I}^{\star}. & & \end{array}$$

where $\mathcal{D}_t$ is a set of expressions that evaluate to probabilistic distributions over values of type t. For instance, $\{0,1\}$ is a constant expression in $\mathcal{D}_{\textsf{Bool}}$ that evaluates to the uniform distribution on bits, and $b \xleftarrow{\$} \{0,1\}$ (a shorthand for Rand b $\{0,1\}$) is an instruction that samples a random bit with uniform probability and assigns it to variable b. As for operators, expressions in $\mathcal{D}$ are either predefined or user-defined. Note that the above syntax lacks a construct for sequential composition; instead, we use lists of instructions.

Finally, a program consists of a main command and an environment, which maps a procedure $p : \mathcal{P}_{(l,t)}$ to a declaration that specifies its formal parameters, its body, and a return expression (procedures are single-exit):

$$\textbf{Record } \textsf{decl} := \{\textsf{params} : \mathcal{V}^{\star}_l; \textsf{body} : \mathcal{C}; \textsf{re} : \mathcal{E}_t\}.$$

We denote games either by G, or by (E, c) when we need to make explicit the environment and the main command. In the rest of the paper we revert to the usual notation to specify games, i.e. we use an explicit return and rely on standard notation as in [10], [11]. For instance, the code of a random oracle is written as:

```
$\mathcal{O}(x) \stackrel{\text{def}}{=}$
  if $x \notin \mathsf{dom}(\boldsymbol{L})$ then $y \xleftarrow{\$} \{0,1\}^k; \boldsymbol{L} \leftarrow (x,y) :: \boldsymbol{L}$;
  return $\boldsymbol{L}(x)$
```

where the global variable $\boldsymbol{L}$ is an association list used to record the calls to the oracle (we follow the convention of typesetting global variables in bold).

2.1.4. Semantics of programs. The semantics of programs is defined in a continuation-passing style, using the measure monad:

$$\textbf{Definition } M(X : \textbf{Type}) := (X \to [0,1]) \to [0,1].$$

where $[0,1]$ is an axiomatization of the unit interval. The definition of the semantics is intricate, but without surprise. However, the semantics has two nonstandard features worth noting: first, it is indexed by a security parameter k (the semantics of types and operators depend on this parameter, so a program should be really seen as a family of programs indexed by the security parameter); second, the semantics is instrumented with a cost monad to compute the computational cost of evaluating a command. The instrumented semantics and the security parameter are used conjointly to formulate the notion of PPT program (cf. [10]). This paper does not elaborate on complexity issues, and views the semantics of programs as a function $[\![G]\!] : \mathcal{S} \to M\ \mathcal{S}$, where $\mathcal{S}$ is the type of deterministic states (memories) which map variables to values.

Let $\overline{A}^{\mu}$ be the characteristic function of the evaluation of the predicate A in memory μ. The measure monad allows to easily define the probability of an event A in a game G and an initial memory μ:

$$\Pr_{G,\mu}[A] \stackrel{\text{def}}{=} [\![G]\!]_{\mu}\ (\lambda \mu'.\overline{A}^{\mu'})$$

In the remainder, we sometimes omit the initial memory μ. In this case one may safely suppose that the memory initially maps variables to default values of their type.

2.1.5. From programs to games. Probabilistic programs capture only partially the interaction between the challenger, the adversary, and the oracles. In addition to their code, the specification of games must also prescribe the expected behavior of the adversary. Consider the definition of EF-CMA security for signature schemes, that could be modeled by the game:

$$(m,\sigma) \leftarrow \mathcal{A}(); b \leftarrow \mathcal{V}(m,\sigma)$$

where $\mathcal{A}$ is the adversary, $\mathcal{V}$ the signature verification algorithm, and the fixed public and secret keys are modeled as global variables. The definition implicitly assumes that the adversary has oracle access to the signing oracle, has read access to the public key, does not have access to the secret key, and that the message m the adversary submits is not one of the messages whose signature it requested before. To make this assumptions explicit, games must specify write and read policies for the adversary. Constraints on the adversary are captured by an inductive well-formedness predicate WFAdv; the adversary is modeled as a procedure whose body is a variable of type $\mathcal{C}$, and that satisfies WFAdv for a given policy. It is then possible to reason by induction on well-formed adversaries. The notion of well-formed adversary, however, is not sufficient to guarantee that the message m is fresh. This may be specified by recording calls to the signing oracle in a list and stating as postcondition of the above game that m does not appear in this list.

2.2. Game-based proofs

CertiCrypt provides tactics that support common patterns of reasoning in game-based proofs. A first set of tactics implement methods to reason about the observational equivalence of programs, including certified program transformations in the style of certified compilers [13]. A second set of tactics extend these methods to the more general setting of a relational Hoare logic, which is typically used to establish invariants or to prove that two programs are observationally equivalent in a given context specified by means of pre- and postconditions. A third batch of specialized tactics implement techniques commonly used in game-based proofs, including lazy sampling, and reasoning about failure events.

2.2.1. Probabilistic information flow. Observational equivalence is captured as a form of probabilistic non-interference, and is defined relative to a set of input variables I and a set of output variables O. Formally, two measurable functions f and g are equivalent w.r.t. a set of variables O, written $f =_O g$, iff $\forall \mu_1\ \mu_2,\ \mu_1 =_O \mu_2 \Rightarrow f\ \mu_1 = g\ \mu_2$. Then, two programs G_1 and G_2 are observationally equivalent w.r.t. input variables I and output variables O, written $\models G_1 \simeq^I_O G_2$, iff $[\![G_1]\!]_{\mu_1} f = [\![G_2]\!]_{\mu_2} g$ for all memories μ_1 and μ_2 and measurable functions f and g such that $\mu_1 =_I \mu_2$ and $f =_O g$.

One tactic widely used to reason about probabilistic information flow is `eqobs_in`, which takes as input a main command c, two environments E and E', a set of output variables O and either fails or produces a set of variables I such that $\models E, c \simeq_O^I E', c$. The tactic `eqobs_in` performs a dependency analysis; to handle procedure calls, it relies on information flow specifications for procedures. For each procedure p with body c_p in E and body c'_p in E', its specification gives sets I_p and O_p such that $\models E, c_p \simeq_{O_p}^{I_p} E', c'_p$. In order to guarantee soundness, users must provide the specification of procedures. However, CertiCrypt provides mechanisms to generate such specifications under the assumption that the calling relation between procedures is a well-founded order. The main difficulty resides in generating the specification of adversaries, since their bodies are unknown; CertiCrypt can infer the specification of a well-formed adversary based on its read/write policy and on the specification of the oracles it can call.

Variants of `eqobs_in` include `eqobs_tl`, that does not require the two main commands to coincide, but instead acts only on the greatest common suffix of the two commands. Formally, `eqobs_tl` takes as arguments two games E_1, c_1 and E_2, c_2 and sets of variables I and O, and returns c'_1, c'_2, c and O' such that the rule

$$\models E_1, c'_1 \simeq_{O'}^I E_2, c'_2 \quad \Rightarrow \quad \models E_1, c_1 \simeq_O^I E_2, c_2$$

is valid. It does so by first computing the longest common suffix c of c_1 and c_2 ($c_1 = c'_1; c$, $c_2 = c'_2; c$), and then using the tactic `eqobs_in` to compute O'. In addition, CertiCrypt provides the converse tactic `eqobs_hd` that strips off the longest common prefix of two programs, and a tactic `eqobs_ctxt` that combines both.

Some transitions in game-based proofs consist in applying a standard program transformation T, which transforms goals of the form $\models E_1, c_1 \simeq_O^I E_2, c_2$ into goals of the form $\models E_1, T(c_1) \simeq_O^I E_2, T(c_2)$. In order to automate these transitions, CertiCrypt provides certified implementations that, in addition to the two-sided version above, yield one-sided versions that apply the transformation only to the left (suffix `l`) or right-hand side (suffix `r`) program.

Dead code elimination is one of such transformations and is automated by the tactic `deadcode`; it removes from both programs the instructions that have no influence on the set of ouput variables O. Formally, `deadcode` transforms a goal of the form $\models E_1, c_1 \simeq_O^I E_2, c_2$ into one of the form $\models E_1, c'_1 \simeq_O^I E_2, c'_2$, where c'_1 and c'_2 are simplified versions of the original commands computed by performing and aggressive form of program slicing; in particular, it removes self or unused assignments and trivial branching statements, and performs branch coalescing.

The tactic `ep` implements expression propagation; it relies on a generic function that performs dataflow analysis on programs, transforming the code by performing partial evaluation using the result of the analysis. In order to account for the extensibility of the language, the user can also add simplification rules to the analyzer. For instance, if the language of expressions is extended with a random permutation operator f and its inverse f^{-1}, the user can extend the partial evaluator with the simplification rule $f(f^{-1}(e)) = e$. The tactic `ep` starts the analysis with an empty domain, whereas the variant `ep_eq` $e\ v$ starts the analysis assuming that $e = v$ initially holds and generates the corresponding proof obligation.

Local code motion is automated by the tactic `swap`, that reorders instructions in programs to generate a largest common suffix while preserving observational equivalence. Formally, `swap` transforms a goal of the form $\models E_1, c_1 \simeq_O^I E_2, c_2$ into one of the form $\models E_1, c'_1; c \simeq_O^I E_2, c'_2; c$. Typically, an application of `swap` is immediately followed by an application of `eqobs_tl` to remove the computed suffix. CertiCrypt automates non-local code motion by providing support to the lazy sampling technique. This technique allows to define a random value, which would otherwise be sampled inside a procedure, at the beginning of a game.

Finally, CertiCrypt implements a tactic `alloc` $x\ y$ to introduce a copy of variable x in variable y and consistently replace all its uses, and a tactic `inline` to inline procedure calls in programs. The tactic `sinline` combines `inline`, `alloc`, `ep`, and `deadcode` in one powerful tactic to further simplify the goal after inlining a procedure call.

2.2.2. Dealing with properties.

Program transformations are seldom justified on their own; rather their validity relies on program invariants. Consider for example these two oracles

$$\mathcal{O}_1(x) \overset{\text{def}}{=} \textsf{return } x \times \boldsymbol{y} \qquad \mathcal{O}_2(x) \overset{\text{def}}{=} \textsf{return } x \times 2$$

Are they observationally equivalent in the context of a game $\boldsymbol{y} \leftarrow 2; d \leftarrow \mathcal{A}()$? The answer depends on whether variable $\boldsymbol{y}$ can be written by the adversary $\mathcal{A}$ and whether other procedures in the environment preserve its value. In order to justify such transformations, one is forced to consider a generalization of observational equivalence to a full-fledged probabilistic relational Hoare logic. This logic deals with judgments of the form $\models G_1 \sim G_2 : P \Rightarrow Q$ where P and Q are relations over states. The formal meaning of such a judgment is that for all memories μ_1 and μ_2 satisfying $P\ \mu_1\ \mu_2$, we have $Q^\#\ [\![G_1]\!]_{\mu_1}\ [\![G_2]\!]_{\mu_2}$, where $Q^\#$ is a lifting of relation Q to distributions [10]. CertiCrypt implements a logic that supports reasoning about such quadruples, and that justifies substituting oracle $\mathcal{O}_2$ for oracle $\mathcal{O}_1$. Intuitively, the proof proceeds by showing that the invariant $\boldsymbol{y} = 2$ is established after the first assignment to $\boldsymbol{y}$, and that it is preserved throughout execution. This can be achieved, assuming $\mathcal{A}$ honors a read-only policy on y, by proving that the invariant is preserved by all oracles that can be called by $\mathcal{A}$. The probabilistic relational Hoare logic is described in detail in [10], and is supported by proof rules and by a sound but incomplete weakest precondition calculus.

Since transformations are required to preserve invariants and postconditions (as these properties may be used subsequently in the proof), all the tactics that have been described in the previous subsection have been extended to handle them. Concretely, all tactics have been extended to deal with postconditions of the form $(=_O \wedge Q)$, where O is a set of output variables on which the two games are required to coincide, and Q is an arbitrary relation over states. As for observational equivalence the tactics need additional information about procedures, namely that they preserve the invariants. CertiCrypt provides mechanisms to automatically build the information about adversaries based on the information about the oracles. The tactics stop when a variable on which the invariant depends is directly modified (it poses no problem if it is modified inside the body of a procedure, since its information ensures the invariant is preserved). At this point, the user can use the tactic `wp` which simultaneously performs a weakest precondition calculation and a backwards dependency analysis in the style of `eqobs_in`. It is worth noting that it is not possible to deal separately with the two conjuncts of a postcondition $(=_O \wedge Q)$ and invoke the tactics for observational equivalence to handle the first, as the rule:

$$\frac{\models G_1 \sim G_2 : \Psi \Rightarrow \Phi \qquad \models G_1 \sim G_2 : \Psi \Rightarrow \Phi'}{\models G_1 \sim G_2 : \Psi \Rightarrow \Phi \wedge \Phi'}$$

is unsound, as illustrated by the following example:

$$c_1 \overset{\text{def}}{=} x \overset{\$}{\leftarrow} \{0,1\}; y \leftarrow x \quad c_2 \overset{\text{def}}{=} x, y \overset{\$}{\leftarrow} \{0,1\}$$

Indeed, we have $\models c_1 \simeq^{\emptyset}_{\{x\}} c_2$ and $\models c_1 \simeq^{\emptyset}_{\{y\}} c_2$ but not $\models c_1 \simeq^{\emptyset}_{\{x,y\}} c_2$, since an observer with access to both x and y can trivially distinguish the joint distribution generated by one program from the one generated by the other.

The tactic `cp_test` allows to perform a case analysis over the guard e of a conditional statement. The tactic uses the logical rule:

$$\frac{\models G_1 \sim G_2 : A \wedge P \Rightarrow Q \qquad \models G_1 \sim G_2 : \neg A \wedge P \Rightarrow Q}{\models G_1 \sim G_2 : P \Rightarrow Q}$$

and propagates the precondition strengthened with, respectively, $e = \mathsf{true}$ or $e = \mathsf{false}$ through the branches using the tactic `ep_eq`. Variants of this tactic perform the propagation on one of G_1 and G_2 or on both, provided e evaluates in the same way in both games.

It is important to note that neither the observational equivalence nor the relational Hoare logic appear in the final statements of security. Instead, both relations are used to reason about the probability of events. For example, if the predicate A depends only on a set of variables O, then to show that $\Pr_{G_1,\mu_1}[A] = \Pr_{G_2,\mu_2}[A]$ it is sufficient to show $\models G_1 \simeq^I_O G_2$ and $\mu_1 =_I \mu_2$. Likewise, if $\models G_1 \sim G_2 : =_I \Rightarrow =_O \wedge P\langle 2\rangle$ and $\mu_1 =_I \mu_2$, then $\Pr_{G_1,\mu_1}[A] = \Pr_{G_2,\mu_2}[A \wedge P]$, where, given a predicate P over memories we denote by $P\langle 1\rangle$ (respectively $P\langle 2\rangle$) the relation over memories which is true if the left (respectively right) memory satisfies the predicate P.

2.2.3. Failure events. Game-based proofs often rely on failure events to justify a transition. Informally, such transitions involve two games that only differ after some flag bad has been set to true, signaling a failure. Therefore for every event A, the probability of the event $A \wedge \neg\mathsf{bad}$ is the same in both games. A corollary of this result is sometimes referred to as the Fundamental Lemma of game-playing. It is particularly amenable to automation since it admits a completely syntactical formulation; it is completely automated in CertiCrypt.

2.2.4. Complexity. Assumptions about hard mathematical problems appear all the time in cryptographic proofs, but only hold for probabilistic polynomial-time (PPT) adversaries. For programs without loops or recursive calls, the tactic `PPT_tac` generates automatically a proof of membership to the class of PPT programs. This is used in next section to obtain an asymptotic security result from a proof by reduction.

3. Certifying the Full Domain Hash signature scheme

In this section we will go through the formalization in CertiCrypt of two different proofs of security of the Full Domain Hash (FDH) signature scheme. The FDH scheme was first proposed by Bellare and Rogaway [14] as an efficient RSA-based signature scheme, but is in fact an instance of an earlier construction described by the same authors in [15]. Here, we will consider this latter, more general construction, which is based on a family of trapdoor one-way permutations f_k on a cyclic group $\mathcal{G}_k$, and a hash function $H : \{0,1\}^* \to \mathcal{G}_k$ whose range is the full domain of f_k. The RSA-based scheme is obtained by instantiating f_k with the RSA function, and the hash function with some cryptographic hash function, such as SHA-1 with the length of its output extended to match that of the RSA modulus. For a given value k of the security parameter, f_k is the public key of the signature scheme, and the private key is the *trapdoor* information allowing to easily invert f_k. The signature of a message $m \in \{0,1\}^*$ is $f_k^{-1}(H(m))$, the preimage of its digest under f_k. To verify a purported signature σ on message m, it suffices to check whether $H(m)$ and $f_k(\sigma)$ coincide.

The FDH scheme can be proved secure in the random oracle model against existential forgery under adaptive chosen-message attacks. This means that if we regard the hash function H as a truly random function, then any computationally feasible adversary with access to the public key and that can ask for the signature of messages of its choice, succeeds in forging a signature for a fresh message only with a negligible probability. This asymptotic security statement is desirable, but of limited practical utility because it does not give any hint as to how to choose the scheme parameters to attain a certain degree of security. What we are really looking for is an exact security statement, a bound that quantifies the gap between the security of the scheme and the intractability of inverting the one-way permutation.

Suppose there exists an adversary against the existential unforgeability of FDH that makes at most $q_{\mathsf{H}}(k)$ and $q_{\mathsf{S}}(k)$ queries to the hash and signing oracles respectively, and succeeds in forging a FDH signature for a fresh message with probability $\epsilon(k)$ within time $t(k)$. In a code-based setting, such an adversary is regarded as black-box procedure $\mathcal{A}$ run in the context of the following attack game:

```
Game $\mathsf{G_{EF}}$ :                $H(m) \overset{\text{def}}{=}$
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;            if $m \notin$ dom($\boldsymbol{L}$) then
  $(m, \sigma) \leftarrow \mathcal{A}()$;            $h \overset{\$}{\leftarrow} \mathcal{G}; \boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$
  $h \leftarrow H(m)$                 return $\boldsymbol{L}(m)$
                              Sign($m$) $\overset{\text{def}}{=}$
                                $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
                                return $f^{-1}(h)$
```

Its success probability is $\Pr_{\mathsf{G_{EF}}}[h = f(\sigma)]$. Note that in the above game the signing oracle makes a hash query each time the adversary asks for the signature of a message, and an additional hash query is made at the end as part of the verification. Thus the number of effective hash queries is at most $q_{\mathsf{H}} + q_{\mathsf{S}} + 1$. All this is captured by the following postcondition,

$$\Phi_{\mathsf{EF}} \overset{\text{def}}{=} |\boldsymbol{L}| \leq q_{\mathsf{H}} + q_{\mathsf{S}} + 1 \ \wedge \ |\boldsymbol{S}| \leq q_{\mathsf{S}} \ \wedge \ m \notin \boldsymbol{S}$$

So, $\Pr_{\mathsf{G_{EF}}}[A] = \Pr_{\mathsf{G_{EF}}}[A \wedge \Phi_{\mathsf{EF}}]$ for any event A.

In the proofs presented in the remainder of this section we will show two different ways of constructing an inverter $\mathcal{I}$ that uses the forger $\mathcal{A}$ to invert f_k. These constructions effectively reduce the security of the signature scheme to the intractability of inverting the underlying one-way permutation.

3.1. The original proof

The proof appearing in [15] provides a security bound that depends on the number of queries the adversary makes to both, the hash and the signing oracle.

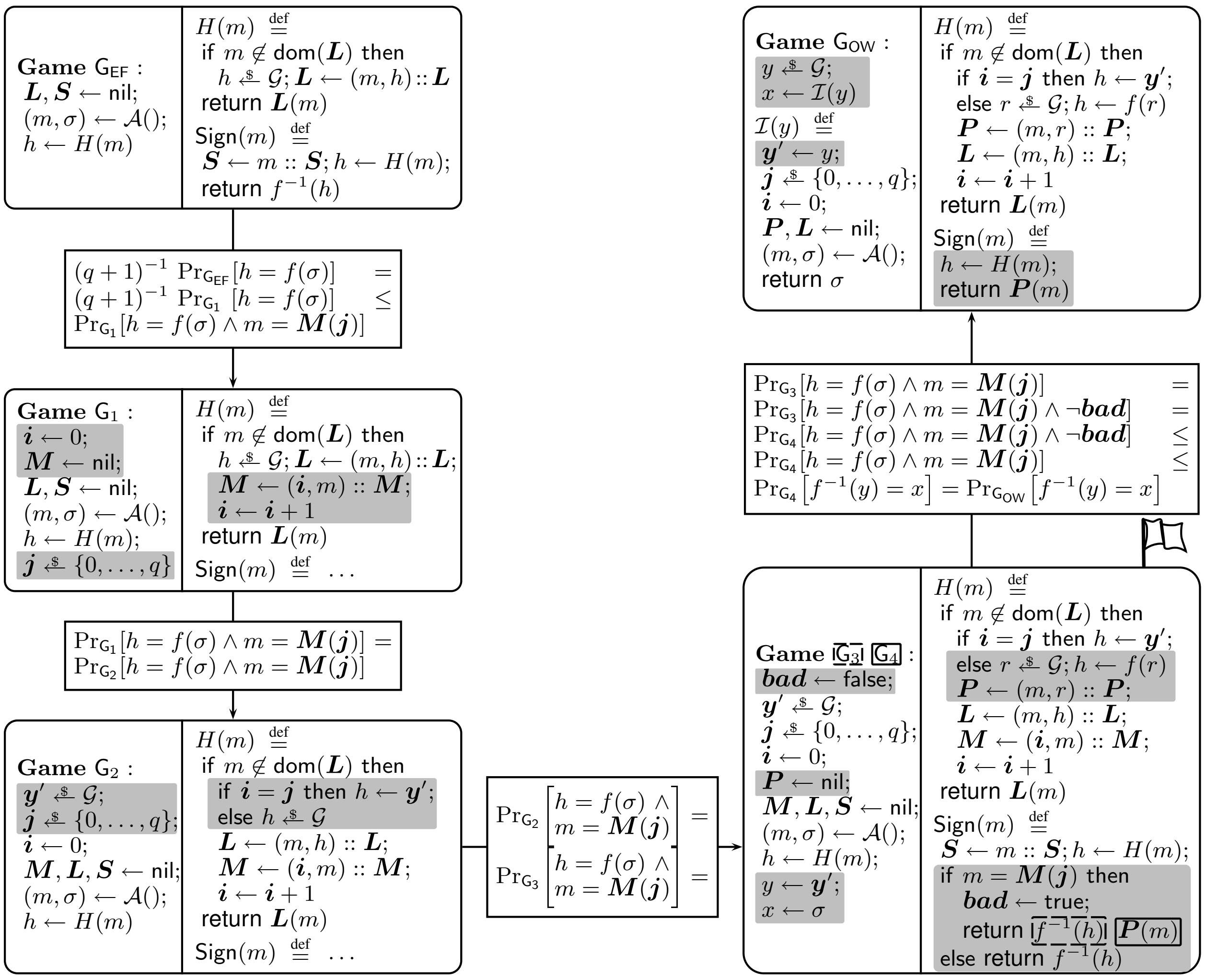
Theorem 1 (Original bound): There exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f_k with probability $\epsilon'(k)$ within time $t'(k)$, where

$$\epsilon'(k) \geq (q_{\mathsf{H}}(k) + q_{\mathsf{S}}(k) + 1)^{-1} \ \epsilon(k)$$
$$t'(k) \leq t(k) + (q_{\mathsf{H}}(k) + q_{\mathsf{S}}(k)) \ \Theta(T_f)$$

and $T_f(k)$ is a bound on the cost of evaluating f_k.

The inverter $\mathcal{I}$ shown in game $\mathsf{G_{OW}}$ in Fig. 1 does the job. It simulates an environment for $\mathcal{A}$ where it replaces the hash and signing oracles with versions of its own. The figure shows the sequence of games that we use to relate the success of the inverter in $\mathsf{G_{OW}}$ to the success of the forger in the attack game $\mathsf{G_{EF}}$. For each game the main experiment is shown alongside the code of procedures in the environment; code pieces that change with respect to the previous game in the sequence appear on a grey background.

Let $q \overset{\text{def}}{=} q_{\mathsf{S}} + q_{\mathsf{H}}$. The inverter first randomly chooses an index $\boldsymbol{j}$ in $\{0, \ldots, q\}$ and then runs the forger intercepting its oracle queries. It answers to the $\boldsymbol{j}$-th hash query (we index the queries from 0) with his challenge y, and to the remaining hash queries with a random element in the range of f with a known preimage; it stores this preimage in a list $\boldsymbol{P}$. It answers a sign query by first making the corresponding hash query itself and then obtaining the preimage of the hash value under f from the list $\boldsymbol{P}$. The simulation is perfect provided the forger never asks the signature of the message corresponding to the $\boldsymbol{j}$-th hash query, because in this case its preimage will not be in the list. A sufficient condition for the simulation to be correct is that $m = \boldsymbol{M}(\boldsymbol{j})$, since m cannot appear in a sign



```
Game $\mathsf{G}_{\mathsf{EF}}$ :
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$

$H(m) \stackrel{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} \mathcal{G}; \boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$
  return $\boldsymbol{L}(m)$
Sign$(m) \stackrel{\text{def}}{=}$
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  return $f^{-1}(h)$
```

$$\begin{aligned}(q+1)^{-1}\ \Pr_{\mathsf{G}_{\mathsf{EF}}}[h = f(\sigma)] &= \\ (q+1)^{-1}\ \Pr_{\mathsf{G}_1}[h = f(\sigma)] &\le \\ \Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})]\end{aligned}$$

```
Game $\mathsf{G}_1$ :
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{M} \leftarrow$ nil;
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$;
  $\boldsymbol{j} \xleftarrow{\$} \{0,\dots,q\}$

$H(m) \stackrel{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} \mathcal{G}; \boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{M} \leftarrow (\boldsymbol{i}, m) :: \boldsymbol{M}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \stackrel{\text{def}}{=}$ ...
```

$$\begin{aligned}\Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] &= \\ \Pr_{\mathsf{G}_2}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})]\end{aligned}$$

```
Game $\mathsf{G}_2$ :
  $\boldsymbol{y}' \xleftarrow{\$} \mathcal{G}$;
  $\boldsymbol{j} \xleftarrow{\$} \{0,\dots,q\}$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$

$H(m) \stackrel{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    if $\boldsymbol{i} = \boldsymbol{j}$ then $h \leftarrow \boldsymbol{y}'$;
    else $h \xleftarrow{\$} \mathcal{G}$
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{M} \leftarrow (\boldsymbol{i}, m) :: \boldsymbol{M}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \stackrel{\text{def}}{=}$ ...
```

$$\Pr_{\mathsf{G}_2}\begin{bmatrix} h = f(\sigma)\ \wedge \\ m = \boldsymbol{M}(\boldsymbol{j})\end{bmatrix} = \qquad \Pr_{\mathsf{G}_3}\begin{bmatrix} h = f(\sigma)\ \wedge \\ m = \boldsymbol{M}(\boldsymbol{j})\end{bmatrix} =$$

```
Game $\boxed{\mathsf{G}_3}$ $\boxed{\mathsf{G}_4}$ :
  $\boldsymbol{bad} \leftarrow$ false;
  $\boldsymbol{y}' \xleftarrow{\$} \mathcal{G}$;
  $\boldsymbol{j} \xleftarrow{\$} \{0,\dots,q\}$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{P} \leftarrow$ nil;
  $\boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$;
  $y \leftarrow \boldsymbol{y}'$;
  $x \leftarrow \sigma$

$H(m) \stackrel{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    if $\boldsymbol{i} = \boldsymbol{j}$ then $h \leftarrow \boldsymbol{y}'$;
    else $r \xleftarrow{\$} \mathcal{G}; h \leftarrow f(r)$
    $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{M} \leftarrow (\boldsymbol{i}, m) :: \boldsymbol{M}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \stackrel{\text{def}}{=}$
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  if $m = \boldsymbol{M}(\boldsymbol{j})$ then
    $\boldsymbol{bad} \leftarrow$ true;
    return $\boxed{f^{-1}(h)}$ $\boxed{\boldsymbol{P}(m)}$
  else return $f^{-1}(h)$
```

$$\begin{aligned}\Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] &= \\ \Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j}) \wedge \neg\boldsymbol{bad}] &= \\ \Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j}) \wedge \neg\boldsymbol{bad}] &\le \\ \Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] &\le \\ \Pr_{\mathsf{G}_4}\left[f^{-1}(y) = x\right] = \Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right]\end{aligned}$$

```
Game $\mathsf{G}_{\mathsf{OW}}$ :
  $y \xleftarrow{\$} \mathcal{G}$;
  $x \leftarrow \mathcal{I}(y)$
$\mathcal{I}(y) \stackrel{\text{def}}{=}$
  $\boldsymbol{y}' \leftarrow y$;
  $\boldsymbol{j} \xleftarrow{\$} \{0,\dots,q\}$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{P}, \boldsymbol{L} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  return $\sigma$

$H(m) \stackrel{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    if $\boldsymbol{i} = \boldsymbol{j}$ then $h \leftarrow \boldsymbol{y}'$;
    else $r \xleftarrow{\$} \mathcal{G}; h \leftarrow f(r)$
    $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \stackrel{\text{def}}{=}$
  $h \leftarrow H(m)$;
  return $\boldsymbol{P}(m)$
```

Figure 1. Sequence of games in the original proof of FDH

query (it must be fresh). We will show how to stepwise transform the attack game into the simulation.

In game G_1 we instrument the hash oracle to keep track of the indices of queries, and we introduce the guess $\boldsymbol{j}$ at the end of the game. Consider the invariant

$$I_1 \stackrel{\text{def}}{=} \begin{array}{l}|\boldsymbol{L}| = |\boldsymbol{M}| = \boldsymbol{i}\ \wedge (\forall i' \in \mathsf{dom}(\boldsymbol{M}), i' < \boldsymbol{i})\ \wedge \\ (\forall m \in \mathsf{dom}(\boldsymbol{L}), \exists i' \in \mathsf{dom}(\boldsymbol{M}), m = \boldsymbol{M}(i'))\end{array}$$

We prove that

$$\models \mathsf{G}_{\mathsf{EF}} \sim \mathsf{G}_1 : \mathsf{true} \Rightarrow =_{\{\boldsymbol{L},\boldsymbol{S},m,\sigma,h\}} \wedge\, I_1\langle 2\rangle$$

The tactics `wp` and `eqobs_in` are enough to construct the information for the oracles in the environments in games G_1 and G_2. The resulting procedure information ι_{12} is then extended automatically to the adversary. The proof script for the above lemma is just

```
deadcode $\iota_{12}$; eqobs_tl $\iota_{12}$; wp; ...
```

The tactic `deadcode` removes the random assignment to $\boldsymbol{j}$ in G_1, `eqobs_tl` removes the common suffix, with the exception of the instruction $\boldsymbol{L} \leftarrow$ nil because it affects the invariant I_1. The resulting goal is

$$\models \boldsymbol{L} \leftarrow \mathsf{nil} \sim \boldsymbol{i} \leftarrow 0; \boldsymbol{M}, \boldsymbol{L} \leftarrow \mathsf{nil} : \\ \mathsf{true} \Rightarrow =_{\{\boldsymbol{L}\}} \wedge\, I_1\langle 2\rangle$$

The tactic `wp` computes the weakest precondition of $=_{\{\boldsymbol{L}\}} \wedge I_1\langle 2\rangle$; the ellipsis stands for a straightforward script to prove that this weakest precondition holds. Games G_{EF} and G_1 are thus equivalent on h and σ, which implies

$$\Pr_{\mathsf{G}_{\mathsf{EF}}}[h = f(\sigma)] = \Pr_{\mathsf{G}_1}[h = f(\sigma)] \qquad (1)$$

Games G_{EF} and G_1 are also equivalent on all the free variables appearing in Φ_{EF}, so Φ_{EF} is a postcondition of G_1 as well. Furthermore, since the game makes a last hash call for m, $m \in \mathsf{dom}(\boldsymbol{L})$. Now we have that

$$\Phi_{\mathsf{EF}} \wedge I_1 \wedge m \in \mathsf{dom}(\boldsymbol{L}) \Rightarrow \exists i', i' \leq q \wedge m = \boldsymbol{M}(i')$$

So, there exists at least an index i' in $\{0, \dots, q\}$ such that $m = \boldsymbol{M}(i')$. (In fact, there exists exactly one, but we do not need to prove it.) The probability of $\boldsymbol{j}$ being one of such indices is at least $(q+1)^{-1}$ and is independent of the success of the forgery, thus

$$\frac{\Pr_{\mathsf{G}_1}[h = f(\sigma)]}{q+1} \leq \Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \quad (2)$$

Game G_2 eagerly samples the value $\boldsymbol{y'}$ that is given as answer to the $\boldsymbol{j}$-th hash query, and that will later become the challenge to the inverter. By the lazy sampling lemma we obtain, $\models \mathsf{G}_1 \simeq^{\emptyset}_{\{\boldsymbol{j}, \boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S}, m, \sigma, h\}} \mathsf{G}_2$. Therefore,

$$\begin{aligned} &\Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \\ &\qquad = \Pr_{\mathsf{G}_2}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \end{aligned} \quad (3)$$

In game G_3 we modify the way hash queries are answered. For all but the $\boldsymbol{j}$-th query we return the image under f of a uniformly sampled element in its domain, and we store this element in a list $\boldsymbol{P}$. This is a local change that does not change the distribution of the answers. Indeed, since f is a permutation we have

$$\models E, h \xleftarrow{\$} \mathcal{G} \simeq^{\emptyset}_{\{h\}} E', r \xleftarrow{\$} \mathcal{G}; h \leftarrow f(r)$$

We also introduce a flag $\boldsymbol{bad}$ to signal whether the simulation failed, i.e. whether the adversary asked the signature of $\boldsymbol{M}(\boldsymbol{j})$. We prove the equivalence

$$\models \mathsf{G}_2 \sim \mathsf{G}_3 : \mathsf{true} \Rightarrow \begin{cases} =_{\{\boldsymbol{j}, \boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S}, m, \sigma, h\}} \wedge \\ (\boldsymbol{M}(\boldsymbol{j}) \notin \boldsymbol{S} \Rightarrow \neg \boldsymbol{bad})\langle 2 \rangle \end{cases}$$

Hence

$$\begin{aligned} &\Pr_{\mathsf{G}_2}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \\ &\qquad = \Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \end{aligned} \quad (4)$$

Games G_3 and G_4 differ only in a portion of code that appears after $\boldsymbol{bad}$ is set, therefore they are syntactically equal up to the failure event $\boldsymbol{bad}$. The Fundamental Lemma gives us

$$\begin{aligned} &\Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j}) \wedge \neg \boldsymbol{bad}] \\ &\qquad = \Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j}) \wedge \neg \boldsymbol{bad}] \end{aligned} \quad (5)$$

The postcondition Φ_{EF} can be propagated to G_3 using the transitivity property of observational equivalence. Then, since $\boldsymbol{M}(\boldsymbol{j}) \notin \boldsymbol{S} \Rightarrow \neg \boldsymbol{bad}$ is an invariant of G_3 and $m \notin \boldsymbol{S}$ a postcondition,

$$\begin{aligned} &\Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \\ &\qquad = \Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j}) \wedge \neg \boldsymbol{bad}] \end{aligned} \quad (6)$$

Define I_4 as

$$\begin{aligned} &\forall m \in \mathsf{dom}(\boldsymbol{L}), m \neq \boldsymbol{M}(\boldsymbol{j}) \Rightarrow \boldsymbol{P}(m) = f^{-1}(\boldsymbol{L}(m)) \wedge \\ &(\boldsymbol{j} \in \mathsf{dom}(\boldsymbol{M}) \Rightarrow \boldsymbol{L}(\boldsymbol{M}(\boldsymbol{j})) = \boldsymbol{y'}) \end{aligned}$$

When answering a sign query for a message $m \neq \boldsymbol{M}(\boldsymbol{j})$, we may obtain the preimage of its hash value from $\boldsymbol{P}$ rather than using f^{-1}. We use I_4 to prove that the signing oracles in games G_4 and G_{OW} are equivalent. To this end we introduce a hybrid game $\mathsf{G}_{4'}$ identical to G_4 but with the signature oracle defined as in G_{OW}, and we show that

$$\models \mathsf{G}_4 \sim \mathsf{G}_{4'} : \mathsf{true} \Rightarrow =_{\{x, y, \boldsymbol{j}, \boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S}, m, \sigma, h\}} \wedge I_4\langle 1 \rangle$$

The invariant I_4 implies that $h = \boldsymbol{L}(\boldsymbol{M}(\boldsymbol{j})) = y$ is a postcondition of G_4, which gives

$$\Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge m = \boldsymbol{M}(\boldsymbol{j})] \leq \Pr_{\mathsf{G}_{4'}}\left[f^{-1}(y) = x\right] \quad (7)$$

Then we prove $\models \mathsf{G}_{4'} \simeq^{\emptyset}_{\{x,y\}} \mathsf{G}_{\mathsf{OW}}$. Its proof script is straightforward,

```
alloc_l y' y; sinline_r ι I.
eqobs_tl ι; swap; deadcode; eqobs_in.
```

Using this lemma we derive

$$\Pr_{\mathsf{G}_{4'}}\left[f^{-1}(y) = x\right] = \Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right] \quad (8)$$

Putting all the above results together, we conclude

$$\frac{\Pr_{\mathsf{G}_{\mathsf{EF}}}[h = f(\sigma)]}{q+1} \leq \Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right] \quad (9)$$

For the time being, CertiCrypt does not provide automation to prove exact bounds on the running time of programs, so we do not certify the bound for $t'(k)$. A glance at the code of the simulation in G_{OW} should suffice to convince oneself that the bound holds. However, a complete proof of asymptotic security follows trivially from (9). Since f_k is a one-way permutation family, $\Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right]$ is negligible in the security parameter provided $\mathcal{I}$ runs in PPT. This is indeed the case, and is proved automatically in CertiCrypt using the tactic `PPT_tac`.

3.2. Improved bound

A tighter security bound for FDH appears in [9]; this bound is independent of the number of hash queries. This is of much practical significance since the number of hash values a real-world forger can compute is only limited by the time and computational resources it invests, whereas the number of signatures it gets could be limited by the owner of the private key.

Theorem 2 (Improved bound): If the permutation family f_k is homomorphic with respect to the group operation in $\mathcal{G}_k$, i.e. $\forall k\ x\ y, f_k(x \times y) = f_k(x) \times f_k(y)$, then there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f_k with probability $\epsilon'(k)$ within time $t'(k)$, where

$$\begin{aligned}\epsilon'(k) &\geq \frac{1}{q_{\mathsf{S}}(k)+1}\left(1-\frac{1}{q_{\mathsf{S}}(k)+1}\right)^{q_{\mathsf{S}}(k)} \epsilon(k)\\ t'(k) &\leq t(k) + (q_{\mathsf{H}}(k)+q_{\mathsf{S}}(k))\ \Theta(T_f)\end{aligned}$$

These bounds hold for the inverter shown in Fig. 2. The inverter first samples $q+1$ bits at random, choosing true with probability p and false with probability $(1-p)$, and stores them in a list $\boldsymbol{T}$. It answers to the $\boldsymbol{i}$-th hash query as follows: it picks uniformly a value r from the domain of f and stores it in a list $\boldsymbol{P}$, then replies according to the $\boldsymbol{i}$-th entry in $\boldsymbol{T}$: if it is true, answers with $y \times f(r)$ where y is its challenge, if it is false answers with simply $f(r)$. In both cases the answers are indistinguishable from those of a random function. When the adversary asks for the signature of a message m, the inverter makes the corresponding hash query itself and then answers with the m entry in the list $\boldsymbol{P}$. The simulation is correct provided the entries in $\boldsymbol{T}$ corresponding to messages appearing in a sign query are false, because in this case the corresponding entries in $\boldsymbol{P}$ give the preimage of their hash value. The aim of the inverter is to make the challenge appear in as much hash queries as possible, while at the same time maximizing the probability of the simulation being correct. The parameter p is left unspecified through the proof and will be chosen later to find the best compromise between these two competing goals.

In game G_1 in Fig. 2 we instrument the hash oracle to keep track of the indices of queries, and we initialize $\boldsymbol{T}$ at the end of the game. Recall that $\overline{e}^{\mu}$ stands for the evaluation of e in memory μ; define

$$\begin{aligned}F_\mu(i,n) &\stackrel{\text{def}}{=} \mathsf{if}\ n \leq i\ \mathsf{then}\ \overline{\boldsymbol{T}(i-n)}^{\mu}\ \mathsf{else}\ p\\ G_\mu(l,n) &\stackrel{\text{def}}{=} \prod_{i\in l} \mathsf{if}\ n \leq i\ \mathsf{then}\ \overline{\neg\boldsymbol{T}(i-n)}^{\mu}\ \mathsf{else}\ 1-p\end{aligned}$$

$F_\mu(i, q+1-|\boldsymbol{T}|)$ equals the probability of $\boldsymbol{T}(i)$ being true after executing Init_T in μ. $G_\mu(i, q+1-|\boldsymbol{T}|)$ is a lower bound on the probability of $\forall i \in l, \neg\boldsymbol{T}(i)$. We prove the following invariant about the while loop Init_T: for every index i and list of indices l such that $i \notin l$,

$$\begin{aligned}&F_\mu(i,q+1-|\boldsymbol{T}|)\ G_\mu(l,q+1-|\boldsymbol{T}|)\\ &\qquad \leq \Pr_{\mathsf{init}_T,\mu}[\boldsymbol{T}(i) \wedge \forall i \in l, \neg\boldsymbol{T}(i)] \qquad (10)\end{aligned}$$

Consider game G_1 up to the point where $\boldsymbol{T}$ is initialized. Postcondition Φ_{EF} holds for game G_{EF}, and we can propagate it to G_1 up to this point. We prove, in addition, that $\mathsf{ran}(\boldsymbol{I}) = [|\boldsymbol{L}|-1\,..\,0]$ is an invariant of this piece of code. So, before initializing $\boldsymbol{T}$ in game G_1 the following precondition holds,

$$|\boldsymbol{L}| \leq q{+}1 \wedge |\boldsymbol{S}| \leq q_{\mathsf{S}} \wedge m \notin \boldsymbol{S} \wedge \mathsf{ran}(\boldsymbol{I}) = [|\boldsymbol{L}|{-}1\,..\,0]$$

which allows us to infer that $\boldsymbol{I}(m) \notin \boldsymbol{I}[\boldsymbol{S}]$, and to apply (10) with $i = \boldsymbol{I}(m)$ and $l = \boldsymbol{I}[\boldsymbol{S}]$, obtaining

$$p\ (1{-}p)^{q_S} \leq \Pr_{\mathsf{init}_T}[\boldsymbol{T}(\boldsymbol{I}(m)) \wedge \forall m' \in \boldsymbol{S}, \neg\boldsymbol{T}(\boldsymbol{I}(m'))]$$

And therefore

$$\begin{aligned}&p\ (1-p)^{q_S}\ \Pr_{\mathsf{G}_1}[h = f(\sigma)]\\ &\qquad \leq \Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge \mathsf{Success}] \qquad (11)\end{aligned}$$

Observe that the transformation of G_1 into G_2 can be justified by locally reasoning on the code of the hash oracle, without need to apply the lazy sampling lemma as before, thanks to the fact that f is a permutation and $f(r)$ acts as a one-time pad. The proof continues in a similar fashion as the one presented before. For the sake of brevity, we only describe the last transition. We prove the following invariant of G_4,

$$\begin{aligned}\forall (m,h) \in \boldsymbol{L},\quad &\boldsymbol{T}(\boldsymbol{I}(m)) \Rightarrow h = \boldsymbol{y'} \times f(\boldsymbol{P}(m)) \wedge\\ &\neg\boldsymbol{T}(\boldsymbol{I}(m)) \Rightarrow h = f(\boldsymbol{P}(m))\end{aligned}$$

This permits to show that the signing oracles in G_4 and G_{OW} are equivalent and, using the homomorphic property of f, to show

$$\begin{aligned}\Pr_{\mathsf{G}_4}[h{=}f(\sigma) \wedge \mathsf{Success}] &\leq \Pr_{\mathsf{G}_4}[y \times f(\boldsymbol{P}(m)) = f(\sigma)]\\ &= \Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right]\end{aligned}$$

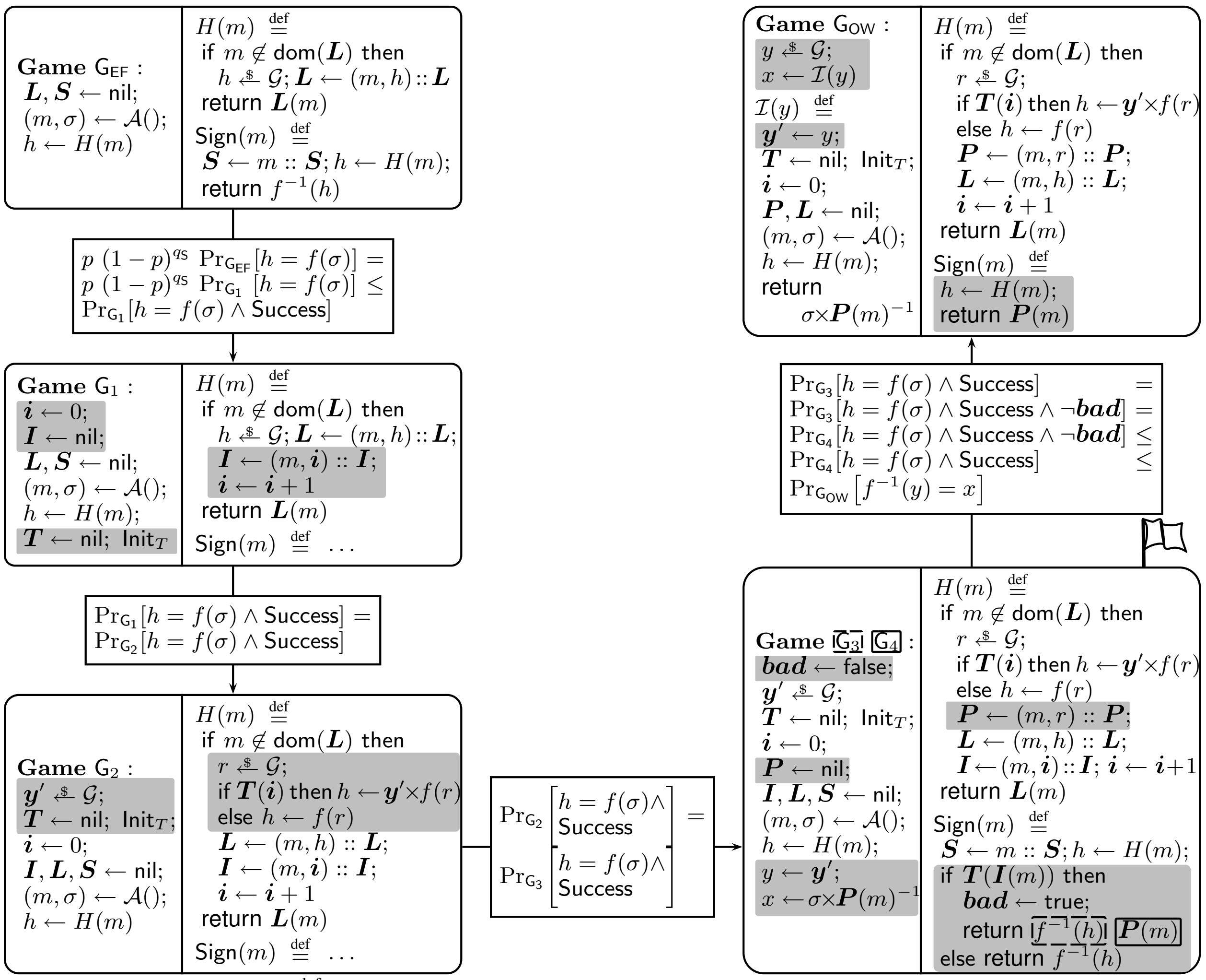
Therefore, we can conclude

$$p\ (1-p)^{q_S}\ \Pr_{\mathsf{G}_{\mathsf{EF}}}[h = f(\sigma)] \leq \Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right]$$

We get the bound in the statement of the theorem by choosing $p = (q_{\mathsf{S}}+1)^{-1}$, which maximizes the factor $p\ (1{-}p)^{q_S}$. For this value of p, the factor approximates $\exp(-1)\ q_{\mathsf{S}}^{-1}$ for large values of q_{S}.

3.3. Practical interpretation

If one accepts that it is reasonable to draw practical conclusions from a security proof in the random oracle model, then the results above may be used to choose the scheme parameters based on an estimate of the time needed to invert the underlying one-way permutation. For instance, the best known method to invert the RSA function is to factorize its modulus; the fastest factorization algorithm (NFS) could factorize a 1024-bit number in around 2^{80} operations, and a 2048-bit number in around 2^{111} operations. Assume some safe bounds for q_{H} and q_{S}, $q_{\mathsf{H}} \leq 2^{60}$, $q_{\mathsf{S}} \leq 2^{20}$. To ensure no forger within these bounds could forge a RSA-FDH signature within $t = 2^{80}$ operations, one should pick



```
Game $\mathsf{G}_{\mathsf{EF}}$ :
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$

$H(m) \overset{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} \mathcal{G}; \boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$
  return $\boldsymbol{L}(m)$
Sign$(m) \overset{\text{def}}{=}$
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  return $f^{-1}(h)$
```

$$p\ (1-p)^{q_{\mathsf{S}}}\ \Pr_{\mathsf{G}_{\mathsf{EF}}}[h = f(\sigma)] = p\ (1-p)^{q_{\mathsf{S}}}\ \Pr_{\mathsf{G}_1}[h = f(\sigma)] \leq \Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge \mathsf{Success}]$$

```
Game $\mathsf{G}_1$ :
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{I} \leftarrow$ nil;
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$;
  $\boldsymbol{T} \leftarrow$ nil; Init$_T$

$H(m) \overset{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} \mathcal{G}; \boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{I} \leftarrow (m,\boldsymbol{i}) :: \boldsymbol{I}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \overset{\text{def}}{=}$ ...
```

$$\Pr_{\mathsf{G}_1}[h = f(\sigma) \wedge \mathsf{Success}] = \Pr_{\mathsf{G}_2}[h = f(\sigma) \wedge \mathsf{Success}]$$

```
Game $\mathsf{G}_2$ :
  $\boldsymbol{y}' \xleftarrow{\$} \mathcal{G}$;
  $\boldsymbol{T} \leftarrow$ nil; Init$_T$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{I}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$

$H(m) \overset{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $r \xleftarrow{\$} \mathcal{G}$;
    if $\boldsymbol{T}(\boldsymbol{i})$ then $h \leftarrow \boldsymbol{y}' \times f(r)$
    else $h \leftarrow f(r)$
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{I} \leftarrow (m,\boldsymbol{i}) :: \boldsymbol{I}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \overset{\text{def}}{=}$ ...
```

$$\Pr_{\mathsf{G}_2}\left[\begin{matrix} h = f(\sigma) \wedge \\ \mathsf{Success} \end{matrix}\right] = \Pr_{\mathsf{G}_3}\left[\begin{matrix} h = f(\sigma) \wedge \\ \mathsf{Success} \end{matrix}\right]$$

```
Game $\boxed{\mathsf{G}_3}$ $\boxed{\mathsf{G}_4}$ :
  $\boldsymbol{bad} \leftarrow$ false;
  $\boldsymbol{y}' \xleftarrow{\$} \mathcal{G}$;
  $\boldsymbol{T} \leftarrow$ nil; Init$_T$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{P} \leftarrow$ nil;
  $\boldsymbol{I}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$;
  $y \leftarrow \boldsymbol{y}'$;
  $x \leftarrow \sigma \times \boldsymbol{P}(m)^{-1}$

$H(m) \overset{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $r \xleftarrow{\$} \mathcal{G}$;
    if $\boldsymbol{T}(\boldsymbol{i})$ then $h \leftarrow \boldsymbol{y}' \times f(r)$
    else $h \leftarrow f(r)$
    $\boldsymbol{P} \leftarrow (m,r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{I} \leftarrow (m,\boldsymbol{i}) :: \boldsymbol{I}$; $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \overset{\text{def}}{=}$
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  if $\boldsymbol{T}(\boldsymbol{I}(m))$ then
    $\boldsymbol{bad} \leftarrow$ true;
    return $\boxed{f^{-1}(h)}$ $\boxed{\boldsymbol{P}(m)}$
  else return $f^{-1}(h)$
```

$$\begin{aligned}
&\Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge \mathsf{Success}] &=\\
&\Pr_{\mathsf{G}_3}[h = f(\sigma) \wedge \mathsf{Success} \wedge \neg \boldsymbol{bad}] &=\\
&\Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge \mathsf{Success} \wedge \neg \boldsymbol{bad}] &\leq\\
&\Pr_{\mathsf{G}_4}[h = f(\sigma) \wedge \mathsf{Success}] &\leq\\
&\Pr_{\mathsf{G}_{\mathsf{OW}}}\left[f^{-1}(y) = x\right]
\end{aligned}$$

```
Game $\mathsf{G}_{\mathsf{OW}}$ :
  $y \xleftarrow{\$} \mathcal{G}$;
  $x \leftarrow \mathcal{I}(y)$
$\mathcal{I}(y) \overset{\text{def}}{=}$
  $\boldsymbol{y}' \leftarrow y$;
  $\boldsymbol{T} \leftarrow$ nil; Init$_T$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{P}, \boldsymbol{L} \leftarrow$ nil;
  $(m,\sigma) \leftarrow \mathcal{A}()$;
  $h \leftarrow H(m)$;
  return
      $\sigma \times \boldsymbol{P}(m)^{-1}$

$H(m) \overset{\text{def}}{=}$
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $r \xleftarrow{\$} \mathcal{G}$;
    if $\boldsymbol{T}(\boldsymbol{i})$ then $h \leftarrow \boldsymbol{y}' \times f(r)$
    else $h \leftarrow f(r)$
    $\boldsymbol{P} \leftarrow (m,r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m,h) :: \boldsymbol{L}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}(m)$
Sign$(m) \overset{\text{def}}{=}$
  $h \leftarrow H(m)$;
  return $\boldsymbol{P}(m)$
```

$$\begin{aligned}
\mathsf{Init}_T &\overset{\text{def}}{=} \text{while } |\boldsymbol{T}| \leq q \text{ do } (b \xleftarrow{\$} \langle \mathsf{true} \mapsto p, \mathsf{false} \mapsto 1-p\rangle;\ \boldsymbol{T} \leftarrow b :: \boldsymbol{T})\\
\mathsf{Success} &\overset{\text{def}}{=} \boldsymbol{T}(\boldsymbol{I}(m)) \wedge \forall m' \in \boldsymbol{S}, \neg \boldsymbol{T}(\boldsymbol{I}(m'))
\end{aligned}$$

Figure 2. Sequence of games in the proof of the improved security bound of FDH

the RSA modulus such that factoring it takes at least roughly $q_{\mathsf{S}}(t + (q_{\mathsf{H}} + q_{\mathsf{S}})\Theta(T_f)) \approx 2^{100}$ operations, otherwise one can iterate the construction in Theorem 2 q_{S} times to invert the RSA function in less time than using the NFS algorithm. A 1024-bit modulus would not be enough, but a 2048-bit modulus would do. In contrast, if one were to choose the modulus according to the original security bound, even a 2048-bit modulus would not be enough. Unfortunately, for FDH there is no tighter reduction than the one in Theorem 2 as showed by Coron [16].

4. Discussion

Our machine-checked proofs follow quite closely the pen-and-paper game-based proofs of FDH (cf. [4]). There is however one important difference: in order to justify local transformations, machine-checked proofs must make invariants explicit and establish formally their validity. Proving that invariants hold constitutes a fair amount of work. More generally, machine-checked proofs must justify all reasoning, including reasoning about side conditions (e.g. PPT complexity) and about elementary mathematics (groups, probabilities) in terms of basic definitions. In contrast to game transformations, for which suitable tactics have been designed, this form of reasoning is not always amenable to automation, and thus accounts for a substantial amount of the effort and of the size of the proofs. Indeed, we estimate that about a third of the proof scripts are devoted to basic facts about probabilities. In spite of this, the size of machine-checked proofs remains reasonable: the formalizations of Bellare-Rogaway and Coron proofs are about 3,000 lines each. While the length of our proofs might look prohibitive in comparison to published proofs, we expect that machine-checked proofs will shrink substantially as CertiCrypt (and its underlying libraries) mature. However, it must be noted that much of the proof lies outside of the trusted base: in order to trust the proofs of FDH, it is sufficient to trust our formalization of the scheme and of the security statement, the formalization of probabilistic programs provided by CertiCrypt, and the proof checker of Coq. In particular, trusting the proofs of FDH does not require trusting the sequence of games, nor the proofs of transitions, nor the proofs of invariants. In this respect, CertiCrypt provides the highest possible level of assurance for the security of a cryptographic scheme, and breaks the symmetry between the effort of writing and checking a cryptographic proof. Currently, both require a lot of expertise in cryptography, a lot of time, and a good understanding of the proof; in contrast, it is rather immediate and simple for a third party to check a proof in CertiCrypt.

4.1. Related work

There have been several efforts to verify formally the correctness of cryptographic schemes using general purpose proof assistants or domain specific tools. We briefly examine related work in view of three general principles which, in our opinion, should be met by an ideal tool for certifying security proofs[1]:

- *Generality*: the tool should be sufficiently general not to limit the conduct of cryptographic proofs: its underlying language should be sufficiently expressive to capture all notions and assumptions used by cryptographers, and its underlying logic should be sufficiently powerful so that all forms of mathematical judgments can be formalized and checked;
- *High assurance*: the tool should produce proof objects that are verifiable independently by small and trustworthy proof checkers. In particular, the proof objects should justify all steps in the proof, and should guarantee that side conditions which arise in the application of a generic transformation are verified;
- *Automation*: the tool should assist the user in proving transitions between games, or even to find the sequence of games. Ideally the tool should provide sufficient support to handle automatically all routine steps so that users can focus on the creative part of the proof.

1. One related issue, not developed here, is the naturalness of the tool, i.e. how the underlying language of the tool differs from the language used by cryptographers to write games.

There are several works that favor automation. In particular, CryptoVerif excels in this respect: being able to generate not only the proofs between transitions, but also the sequence of games itself, it requires minimal interaction from users. However, CryptoVerif does not achieve generality, and does not provide high assurance.

More recently, Courant et al. [17] have developed a prototype tool to establish security of asymmetric encryption schemes in the Random Oracle Model. The tool performs symbolic execution of programs, and yields proofs of IND-CPA and IND-CCA2 security; the tool has been applied successfully to several schemes. As with CryptoVerif, their tool excels with automation, but is not general and does not provide high assurance. One further limitation of their tool is that it neither gives reductionist arguments nor allows to compute exact security bounds.

In contrast to the above works, several authors have focused on developing proofs that provide high assurance, and formalized game-based proofs in proof assistants. The main results to date are a proof of ElGamal semantic security by Nowak [18], and of the PRP/PRF switching lemma by Affeldt et al. [19]. Although both proofs are conducted in the Coq proof assistant, they are incomplete (e.g. they do not deal with complexity, or deal with a weak, non-adaptive, adversary model), and appear like preliminary experiments that are unlikely to yield a reasonable framework for cryptographic proofs. Very recently, Backes et al. [20] have started developing a framework for game-based proofs in Isabelle. In contrast to our work, they rely on a λ-calculus with references to express games. Although the authors aim for generality, their work is very preliminary: as far as we can judge, only the semantics of the language has been modeled, and no proofs have been conducted. In particular, it remains difficult to assess whether using a higher-order language yields any benefits, or whether on the contrary it complicates the development and usage of the framework significantly.

4.2. Perspectives

In an inspiring paper, Halevi [6] advocates building and using dedicated formal verification tools to help cryptographers increase confidence in their proofs. Our work provides evidence that such tools can indeed be built. On the other hand, conducting proofs in CertiCrypt is currently time consuming and requires a significant amount of expertise. Further work is required to make the tool more accessible, so that its use can become routine and not limited to researchers with a background in formal methods. One essential element to increase the usability of CertiCrypt is to increase automation by interfacing with an external tool, along the paradigm of proof-producing decision procedures. In the envisioned setting, the external tool would rely on (untrusted) heuristics and provide an interface, and would be used to produce a proof sketch, in the form of incomplete proof objects, that would need to be completed interactively using Coq. Typically, the remaining proof obligations would establish that transitions were performed respecting side conditions.

Acknowledgments. This work was partially supported by the French ANR project SCALP. The authors would like to thank the anonymous reviewers for their suggestions and comments.

References

[1] S. Goldwasser and S. Micali, "Probabilistic encryption." *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984.

[2] J. Stern, "Why provable security matters?" in *Advances in Cryptology – EUROCRYPT'03*, ser. Lecture Notes in Computer Science, vol. 2656. Springer-Verlag, 2003, pp. 449–461.

[3] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology – EUROCRYPT'06*, ser. Lecture Notes in Computer Science, vol. 4004. Springer-Verlag, 2006, pp. 409–426.

[4] D. Catalano, R. Cramer, I. Damgard, G. D. Crescenzo, D. Pointcheval, and T. Takagi, *Contemporary Cryptology (Advanced Courses in Mathematics - CRM Barcelona)*. Birkhauser, 2005.

[5] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004.

[6] S. Halevi, "A plausible approach to computer-aided cryptographic proofs," Cryptology ePrint Archive, Report 2005/181, 2005.

[7] B. Blanchet, "A computationally sound mechanized prover for security protocols," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 140–154.

[8] B. Blanchet and D. Pointcheval, "Automated security proofs with sequences of games," in *Advances in Cryptology – CRYPTO'06*, ser. Lecture Notes in Computer Science, vol. 4117. Springer-Verlag, 2006, pp. 537–554.

[9] J.-S. Coron, "On the exact security of Full Domain Hash," in *Advances in Cryptology*, ser. Lecture Notes in Computer Science, vol. 1880. Springer-Verlag, 2000, pp. 229–235.

[10] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th ACM Symposium on Principles of Programming Languages*. ACM Press, 2009, pp. 90–101.

[11] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt," in *5th International Workshop on Formal Aspects in Security and Trust, FAST 2008*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2008, in press.

[12] The Coq development team, "The Coq Proof Assistant Reference Manual v8.2," 2008, [Online]. Available: http://coq.inria.fr.

[13] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *Proceedings of the 33rd ACM Symposium Principles of Programming Languages*. ACM Press, 2006, pp. 42–54.

[14] M. Bellare and P. Rogaway, "The exact security of digital signatures – How to sign with RSA and Rabin," in *Advances in Cryptology – EUROCRYPT'96*, ser. Lecture Notes in Computer Science, vol. 1070. Springer-Verlag, 1996, pp. 399–416.

[15] ——, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*. ACM Press, 1993, pp. 62–73.

[16] J.-S. Coron, "Optimal security proofs for PSS and other signature schemes," in *Advances in Cryptology – EUROCRYPT'02*, ser. Lecture Notes in Computer Science, vol. 2332. Springer-Verlag, 2002, pp. 272–287.

[17] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech, "Towards automated proofs for asymmetric encryption schemes in the random oracle model," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM Press, 2008, pp. 371–380.

[18] D. Nowak, "A framework for game-based security proofs," in *Information and Communications Security*, vol. 4861. Springer-Verlag, 2007, pp. 319–333.

[19] R. Affeldt, M. Tanaka, and N. Marti, "Formal proof of provable security by game-playing in a proof assistant," in *Proceedings of International Conference on Provable Security*, ser. Lecture Notes in Computer Science, vol. 4784. Springer-Verlag, 2007, pp. 151–168.

[20] M. Backes, M. Berg, and D. Unruh, "A formal language for cryptographic pseudocode," in *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*. Springer-Verlag, 2008, pp. 353–376.