Formal Certification of Code-Based Cryptographic Proofs

GILLES BARTHE and SANTIAGO ZANELLA BEGUELIN
IMDEA Software, Madrid, Spain
and
BENJAMIN GREGOIRE
INRIA Sophia Antipolis - Méditerranée, France

As cryptographic proofs have become essentially unverifiable, cryptographers have argued in favor of developing techniques that help tame the complexity of their proofs. The game-based approach is a popular method in which proofs are structured as sequences of games and in which proof steps establish the validity of transitions between successive games. Game-based proofs can be rigorously formalized by taking a code-centric view of games as probabilistic programs and relying on programming language theory to justify proof steps. While this code-based view contributes to formalize the security statements precisely and to carry out proofs systematically, typical proofs are so long and involved that formal verification is necessary to achieve a high degree of confidence. We present CertiCrypt, a framework that enables the machine-checked construction and verification of game-based proofs. CertiCrypt is built upon the general-purpose proof assistant Coq, and draws on many areas, including probability and complexity theory, algebra, and semantics of programming languages. The framework provides certified tools to reason about the equivalence of probabilistic programs, including a relational Hoare logic, a theory of observational equivalence, verified program transformations, and ad-hoc techniques such as reasoning about failure events. We demonstrate the usefulness of CertiCrypt through various examples, including proofs of the security of OAEP against adaptive chosen-ciphertext attacks (with a bound that improves upon previously published results) and of the existential unforgeability of FDH signatures. Our work constitutes a first yet significant step towards Halevi's ambitious program of providing tool support for cryptographic proofs.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics*; *Program analysis*

General Terms: Languages, Security, Verification

Additional Key Words and Phrases: Coq proof assistant, cryptographic proofs, observational equivalence, program transformations, relational Hoare logic

1. INTRODUCTION

Designing secure cryptographic systems is a notoriously difficult task. Indeed, the history of modern cryptography is fraught with examples of cryptographic systems that had been thought secure for a long time before being broken and with flawed

security proofs that stood unchallenged for years. Provable security [Goldwasser and Micali 1984; Stern 2003] is an approach that aims to establish the security of cryptographic systems through a rigorous analysis in the form of a mathematical proof, borrowing techniques from complexity theory. In a typical provable security argument, security is proved by reduction, showing that any attack against the security of the system would lead to an efficient way to solve some computationally hard problem.

Provable security holds the promise of delivering strong guarantees that cryptographic schemes meet their goals and is becoming unavoidable in the design and evaluation of new schemes. Yet provable security *per se* does not provide specific tools for managing the complexity of proofs and as a result several purported security arguments that followed the approach have been shown to be flawed. Consequently, the cryptographic community is increasingly aware of the necessity of developing methodologies that systematize the type of reasoning that pervade cryptographic proofs, and that guarantee that such reasoning is applied correctly. One prominent method for achieving a high degree of confidence in cryptographic proofs is to cast security as a program verification problem: this is achieved by formulating goals and hypotheses in terms of probabilistic programs, and defining the adversarial model in terms of complexity classes, e.g. probabilistic polynomial-time programs. This code-centric view leads to statements that are unambiguous and amenable to formalization. However, standard methods to verify programs (e.g. in terms of program logics) are ineffective to directly address the kind of verification goals that arise from cryptographic statements. The game-based approach [Shoup 2004; Halevi 2005; Bellare and Rogaway 2006] is an alternative to standard program verification methods that establishes the verification goal through successive program transformations. In a nutshell, a game-based proof is structured as a sequence of transformations of the form $G, A \rightarrow^h G', A'$, where G and G' are probabilistic programs, A and A' are events, and h is a monotonic function such that $\Pr[G : A] \leq h(\Pr[G' : A'])$. When the security of a scheme is expressed as an inequality $\Pr[G_0 : A_0] \leq p$, it can be proved by exhibiting a sequence of transformations

$$G_0, A_0 \rightarrow^{h_1} G_1, A_1 \rightarrow \cdots \rightarrow^{h_n} G_n, A_n$$

and proving that $h_1 \circ \cdots \circ h_n(\Pr[G_n : A_n]) \leq p$. Reductionist arguments can be naturally formulated in this manner by exhibiting a sequence of games where $\Pr[G_n : A_n]$ encodes the probability of success of some efficient algorithm in solving a problem believed to be hard. Under this code-centric view of games, game transformations become program transformations and can be justified rigorously by semantic means; in particular, many transformations can be viewed as common program optimizations.

Whereas Bellare and Rogaway [2006] already observed that code-based proofs could be more easily amenable to machine-checking, Halevi [2005] argued that formal verification techniques should be used to improve trust in cryptographic proofs, and set up a program for building a tool that could be used by the cryptographic community to mechanize their proofs. We take a first step towards Halevi's ambitious program by presenting CertiCrypt [Barthe et al. 2009b], a fully machine-checked framework for constructing and verifying game-based cryptographic proofs.

CertiCrypt builds on top of the Coq proof assistant [The Coq development team 2009] a broad set of reasoning principles used by cryptographers, drawing on program verification, algebraic reasoning, and probability and complexity theory. The framework features:

. *Faithful and rigorous encoding of games.* In order to be readily accessible to cryptographers, we adopt a formalism that is commonly used to describe games. Concretely, the lowest layer of CertiCrypt is an imperative programming language with probabilistic assignments, structured datatypes, and procedure calls. We formalize the syntax and semantics of programs; the latter uses the measure monad of Audebaud and Paulin-Mohring [2009]. (For the connoisseur, we provide a deep and dependently-typed embedding of the syntax; thanks to dependent types, the typeability of programs is obtained for free.) The semantics is instrumented to calculate the cost of running programs; this offers the means to define complexity classes, and in particular to define formally the notion of effective (probabilistic polynomial-time) adversary. We provide in addition a precise formalization of the adversarial model that captures many assumptions left informal in proofs, notably including policies on memory access.

. *Exact security.* Many security proofs only show that the advantage of any effective adversary against the security of a cryptographic system is asymptotically negligible w.r.t. a security parameter (which typically determines the length of keys or messages). However, the cryptographic community is increasingly focusing on exact security, a more useful goal since it gives hints as to how to choose system parameters in practice to satisfy a security requirement. The goal of exact security is to provide a concrete upper bound for the advantage of an adversary executing in a given amount of time. This is in general done by reduction, constructing an algorithm that solves a problem believed to be hard and giving a lower bound for its success probability and an upper bound for its execution time in terms of the advantage and execution time of the original adversary. We focus on bounding the success probability (and only provide automation to bound the execution time asymptotically) since it is arguably where lies most of the difficulty of a cryptographic proof.

. *Full and independently verifiable proofs.* We adopt a formal semanticist perspective and go beyond Halevi's vision in two respects. First, we provide a unified framework to carry out full proofs; all intermediate steps of reasoning can be justified formally, including complex side conditions that justify the correctness of transformations (about probabilities, algebra, complexity, etc.). Second, one notable feature of Coq, and thus CertiCrypt, is to deliver independently verifiable proofs, an important motivation behind the game-based approach. More concretely, every proof yields a proof object that can be checked automatically by a (small and trustworthy) proof checking engine. In order to trust a cryptographic proof, one only needs to check its statement and not its details.

. *Powerful and automated reasoning methods.* We formalize a relational Hoare logic and a theory of observational equivalence, and use them as stepping stones to support the main tools of code-based reasoning. In particular, we prove that many transformations used in code-based proofs, including common optimizations, are semantics-preserving. In addition, we mechanize reasoning patterns used ubiqui-

tously in cryptographic proofs, such as reasoning about failure events (the so-called fundamental lemma of game-playing), and a logic for interprocedural code-motion (used to justify the eager/lazy sampling of random values).

Organization of the Article. This article is a revised and extended version of [Barthe et al. 2009b], and builds on [Zanella Béguelin et al. 2009; Barthe et al. 2009a; 2010a; 2010b]. Its purpose is to provide a high-level description of the CertiCrypt framework, overview the case studies formalized so far, and stir further interest in machine-checked cryptographic proofs. The rest of the article is organized as follows: we begin in Section 2 with two introductory examples of game-based proofs, namely the semantic security of the ElGamal and Hashed ElGamal encryption schemes; Section 3 describes the mathematical tools that we use in our formalization of games; in Section 4 we introduce the language used to represent games and its semantics and discuss the notions of complexity and termination; Section 5 presents the probabilistic relational Hoare logic that forms the core of the framework; in Sections 6 and 7 we overview the formulation and automation of game transformations; Section 8 reports on some significant case studies formalized in CertiCrypt. We conclude with a survey of related work and a discussion of lessons learned and perspectives to further this line of research.

2. BASIC EXAMPLES

This section illustrates the principles of the CertiCrypt framework on two elementary examples of game-based proofs: the semantic security of ElGamal encryption under the Decision Diffie-Hellman assumption, and the semantic security of Hashed ElGamal encryption in the Random Oracle Model under the Computational Diffie-Hellman assumption. The language used to represent games will be formally introduced in the next sections; an intuitive understanding should suffice to grasp the meaning of the games appearing here.

We begin with some background on encryption schemes and their security. An asymmetric encryption scheme is composed of a triple of algorithms:

Key generation: Given a security parameter η, the key generation algorithm $\mathcal{KG}(\eta)$ returns a public/secret key pair (pk, sk);

Encryption: Given a public key pk and a plaintext m, the encryption algorithm $\mathcal{E}(pk, m)$ computes a ciphertext corresponding to the encryption of m under pk;

Decryption: Given a secret key sk and a ciphertext c, the decryption algorithm $\mathcal{D}(sk, c)$ returns either the plaintext corresponding to the decryption of c, if it is a valid ciphertext, or a distinguished value $\perp$ otherwise.

Key generation and encryption may be probabilistic, while decryption is deterministic. We require that decryption undo encryption: for every pair of keys (pk, sk) that can be output by the key generation algorithm, and every plaintext m, it must be the case that $\mathcal{D}(sk, \mathcal{E}(pk, m)) = m$.

An asymmetric encryption scheme is said to be semantically secure if it is unfeasible to gain significant information about a plaintext given only a corresponding

ciphertext and the public key. Goldwasser and Micali [1984] showed that semantic security is equivalent to the property of ciphertext indistinguishability under chosen-plaintext attacks (IND-CPA, for short). This property can be formally defined in terms of a game played between a challenger and an adversary $\mathcal{A}$, represented as a pair of procedures $(\mathcal{A}_1, \mathcal{A}_2)$ that may share state:

Game IND-CPA :
$(pk, sk) \leftarrow \mathcal{KG}(\eta)$;
$(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
$b \xleftarrow{\$} \{0,1\}$; $c \leftarrow \mathcal{E}(pk, m_b)$;
$\bar{b} \leftarrow \mathcal{A}_2(c)$

In this game, the challenger first generates a new key pair and gives the public key pk to the adversary, who returns two plaintexts m_0, m_1 of his choice. The challenger then tosses a fair coin b and gives the encryption of m_b back to the adversary, whose goal is to guess which message has been encrypted. The advantage of an adversary $\mathcal{A}$ in the above experiment is defined as

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}} = \left| \Pr\left[\mathsf{IND\text{-}CPA} : b = \bar{b}\right] - \frac{1}{2} \right|$$

The scheme is said to be IND-CPA secure if the advantage of any effective adversary is a negligible function of the security parameter η, i.e. the adversary cannot do much better than a blind guess. Recall that a function $\nu : \mathbb{N} \to \mathbb{R}$ is said to be negligible if it decreases faster than the inverse of any polynomial:

$$\forall c \in \mathbb{N}.\ \exists n_c \in \mathbb{N}.\ \forall n \in \mathbb{N}.\ n \geq n_c \Longrightarrow |\nu(n)| \leq n^{-c}$$

2.1 The ElGamal Encryption Scheme

Let $\{G_\eta\}$ be a family of cyclic prime-order groups indexed by a security parameter $\eta \in \mathbb{N}$. For a specific value of the security parameter, which we leave implicit, let q denote the order of the corresponding group in the family and let g be a generator. ElGamal encryption is defined by the following triple of algorithms:

$$\begin{array}{lcl} \mathcal{KG}(\eta) & \stackrel{\text{def}}{=} & x \xleftarrow{\$} \mathbb{Z}_q;\ \mathsf{return}\ (g^x, x) \\ \mathcal{E}(\alpha, m) & \stackrel{\text{def}}{=} & y \xleftarrow{\$} \mathbb{Z}_q;\ \mathsf{return}\ (g^y, \alpha^y \times m) \\ \mathcal{D}(x, (\beta, \zeta)) & \stackrel{\text{def}}{=} & \mathsf{return}\ (\zeta \times \beta^{-x}) \end{array}$$

We prove the IND-CPA security of ElGamal encryption under the assumption that the Decision Diffie-Hellman (DDH) problem is hard. Intuitively, for a family of finite cyclic groups, the DDH problem consists in distinguishing between triples of the form (g^x, g^y, g^{xy}) and triples of the form (g^x, g^y, g^z), where the exponents x, y, z are uniformly sampled from $\mathbb{Z}_q$. One characteristic of game-based proofs is to formulate computational assumptions using games; the assumption that the DDH problem is hard can be formulated as follows:

Definition 2.1 *Decision Diffie-Hellman assumption.* Consider the games

$$\begin{array}{ll} \textbf{Game}\ \mathsf{DDH}_0 : & x, y \xleftarrow{\$} \mathbb{Z}_q; \quad d \leftarrow \mathcal{B}(g^x, g^y, g^{xy}) \\ \textbf{Game}\ \mathsf{DDH}_1 : & x, y, z \xleftarrow{\$} \mathbb{Z}_q;\ d \leftarrow \mathcal{B}(g^x, g^y, g^z) \end{array}$$

and define the DDH-advantage of an adversary $\mathcal{B}$ as

$$\mathbf{Adv}_{\mathcal{B}}^{\mathsf{DDH}} \stackrel{\text{def}}{=} \left| \Pr\left[\mathsf{DDH}_0 : d = 1\right] - \Pr\left[\mathsf{DDH}_1 : d = 1\right] \right|$$

We say that the DDH assumption holds for the family of groups $\{G_\eta\}$ when the advantage of any effective adversary $\mathcal{B}$ in the above experiment is a negligible function of the security parameter. Note that the semantics of the games (and in particular the order q of the group) depends on the security parameter η.

ElGamal is an emblematic example of game-based proofs. The proof of its security, which follows the proof by Shoup [2004], embodies many of the techniques described in subsequent sections. The proof is done by reduction and shows that every adversary $\mathcal{A}$ against the chosen-plaintext security of ElGamal that achieves a given advantage can be used to construct a distinguisher $\mathcal{B}$ that solves DDH with the same advantage and in roughly the same amount of time. We exhibit a concrete construction of this distinguisher:

Adversary $\mathcal{B}(\alpha, \beta, \gamma)$:
$(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha)$;
$b \xleftarrow{\$} \{0,1\}$;
$\bar{b} \leftarrow \mathcal{A}_2(\beta, \gamma \times m_b)$;
return $b = \bar{b}$

and we prove that $\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{B}} = \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A}}$ for any given adversary $\mathcal{A}$. To conclude the proof (i.e. to show that the advantage of any efficient adversary $\mathcal{A}$ is negligible), we show that the reduction is efficient: the adversary $\mathcal{B}$ executes in probabilistic polynomial-time provided the IND-CPA adversary $\mathcal{A}$ does—we do not show a concrete bound for the execution time of $\mathcal{B}$, although it is evident that it incurs only a constant overhead.

Figure 1 gives a high-level view of the reduction: games appear inside white background boxes, whereas gray background boxes contain the actual proof scripts used to prove observational equivalence between consecutive games. A proof script is simply a sequence of tactics, each intermediate tactic transforms the current goal into a simpler one, whereas the last tactic in the script ultimately solves the goal. The tactics that appear in the figure hopefully have self-explanatory names, but are explained cursorily below and in more detail in Section 6. The proof proceeds by constructing an adversary $\mathcal{B}$ against DDH such that the distribution of $b = \bar{b}$ (equivalently, d) after running the IND-CPA game for ElGamal is exactly the same as the distribution obtained by running game DDH_0. In addition, we show that the probability of d being true in DDH_1 is exactly $1/2$ for the same adversary $\mathcal{B}$. The remaining gap between DDH_0 and DDH_1 is the DDH-advantage of $\mathcal{B}$. The reduction is summarized by the following equations:

$$\begin{aligned}
\left|\Pr\left[\mathsf{IND\text{-}CPA} : b = \bar{b}\right] - 1/2\right| &= \left|\Pr\left[\mathsf{G}_1 : d\right] - 1/2\right| && (1)\\
&= \left|\Pr\left[\mathsf{DDH}_0 : d\right] - 1/2\right| && (2)\\
&= \left|\Pr\left[\mathsf{DDH}_0 : d\right] - \Pr\left[\mathsf{G}_3 : d\right]\right| && (3)\\
&= \left|\Pr\left[\mathsf{DDH}_0 : d\right] - \Pr\left[\mathsf{G}_2 : d\right]\right| && (4)\\
&= \left|\Pr\left[\mathsf{DDH}_0 : d\right] - \Pr\left[\mathsf{DDH}_1 : d\right]\right| && (5)
\end{aligned}$$

Equation (1) holds because games IND-CPA and G_1 induce the same distribution on d. We specify this as an observational equivalence judgment as $\mathsf{IND\text{-}CPA} \simeq_{\{d\}} \mathsf{G}_1$, and prove it using certified program transformations and decision procedures. We

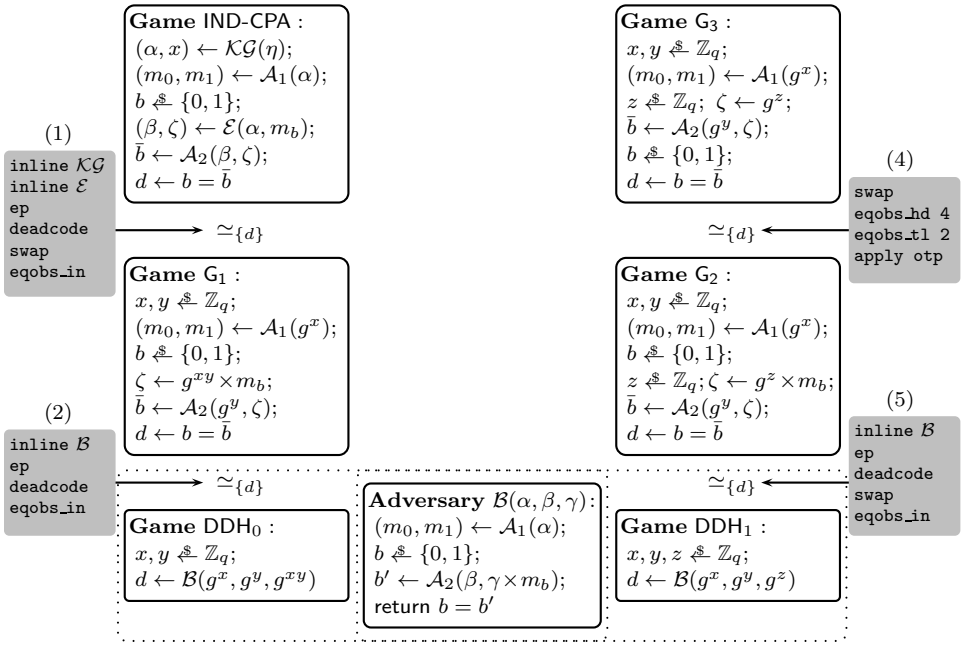

Fig. 1. Code-based proof of ElGamal semantic security.

first inline the procedure calls to $\mathcal{KG}$ and $\mathcal{E}$ in the IND-CPA game and simplify the resulting games by propagating assignments and eliminating dead code (`ep`, `deadcode`). At this point we are left with two games almost identical, except that y is sampled later in one game than in the other. The tactic `swap` hoists instructions in one game whenever is possible in order to obtain a maximal common prefix with another game, and allows us to hoist the sampling of y in the program on the left hand side. A graphical representation of this sequence of program transformations is given in Figure 2. We conclude the proof by applying the tactic `eqobs_in` that decides observational equivalence of a program with itself.

Equations (2) and (5) are obtained similarly, while (3) is established by simple probabilistic reasoning because in game G_3 the bit $\bar{b}$ is independent from b. Finally, to prove Equation (4) we begin by removing the part the two games have in common with the exception of the instruction $z \xleftarrow{\$} \mathbb{Z}_q$ (`swap`, `eqobs_hd`, `eqobs_tl`) and then apply an algebraic property of cyclic groups that we have proved as a lemma (`otp`): if one applies the group operation to an uniformly distributed element of the group and some other constant element, the result is uniformly distributed—a random element acts as a one-time pad. This allows to prove that $z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z \times m_b$ and $z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z$ induce the same distribution on ζ, and thus remove the dependence of $\bar{b}$ on b.

The proof concludes by applying the DDH assumption to show that the IND-CPA advantage of $\mathcal{A}$ is negligible. For this, and in view that $\mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A}} = \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{B}}$, it suffices to prove that the adversary $\mathcal{B}$ is probabilistic polynomial-time (under the assumption that the procedures $\mathcal{A}_1$ and $\mathcal{A}_2$ are so); the proof of this latter fact is

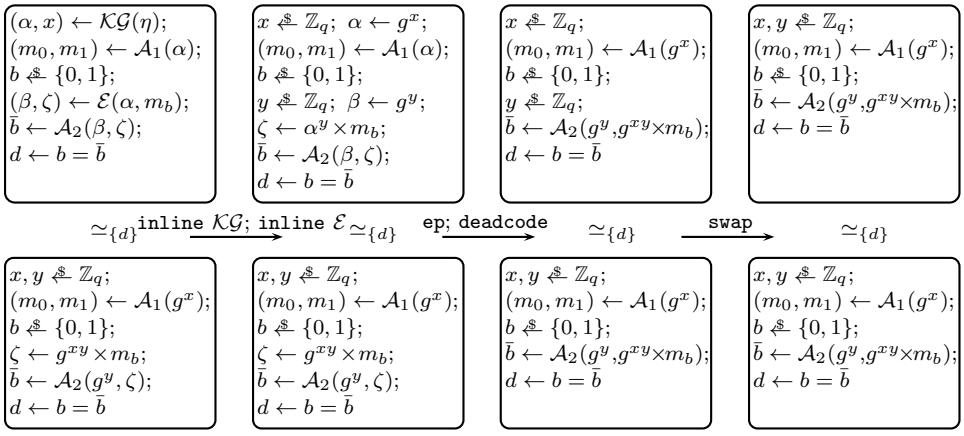

Fig. 2. Sequence of transformations in the proof of IND-CPA $\simeq_{\{d\}}$ G_1.

entirely automated in CertiCrypt.

2.2 The Hashed ElGamal Encryption Scheme

Hashed ElGamal is a variant of the ElGamal public-key encryption scheme that does not require plaintexts to be members of the underlying group G. Instead, plaintexts in Hashed ElGamal are just bitstrings of certain length ℓ and group elements are mapped into bitstrings using a hash function $H : G \to \{0,1\}^\ell$. Formally, the scheme is defined by the following triple of algorithms:

$$\begin{array}{lcl} \mathcal{KG}(\eta) & \stackrel{\text{def}}{=} & x \xleftarrow{\$} \mathbb{Z}_q;\ \mathsf{return}\ (g^x, x) \\ \mathcal{E}(\alpha, m) & \stackrel{\text{def}}{=} & y \xleftarrow{\$} \mathbb{Z}_q;\ h \leftarrow H(\alpha^y);\ \mathsf{return}\ (g^y, h \oplus m) \\ \mathcal{D}(x, (\beta, \zeta)) & \stackrel{\text{def}}{=} & h \leftarrow H(\beta^x);\ \mathsf{return}\ (\zeta \oplus h) \end{array}$$

Hashed ElGamal encryption is semantically secure in the random oracle model under the Computational Diffie-Hellman (CDH) assumption on the underlying group family $\{G_\eta\}$. This is the assumption that it is hard to compute g^{xy} given only g^x and g^y where x and y are uniformly sampled from $\mathbb{Z}_q$. Clearly, the DDH assumption implies the CDH assumption, but the converse need not necessarily hold.[1]

Definition 2.2 *Computational Diffie-Hellman assumption.* Consider the game

$$\textbf{Game}\ \mathsf{CDH}:\ x, y \xleftarrow{\$} \mathbb{Z}_q;\ \gamma \leftarrow \mathcal{B}(g^x, g^y)$$

and define the CDH-advantage of an adversary $\mathcal{B}$ as

$$\mathbf{Adv}^{\mathsf{CDH}}_{\mathcal{B}} \stackrel{\text{def}}{=} \Pr[\mathsf{CDH} : \gamma = g^{xy}]$$

We say that the CDH assumption holds for the family of groups $\{G_\eta\}$ when the advantage of any probabilistic polynomial-time adversary $\mathcal{B}$ is a negligible function of the security parameter.

[1] Groups where DDH is easy and CDH is believed to be hard are of practical importance in cryptography and are called Diffie-Hellman *gap* groups [Okamoto and Pointcheval 2001].

We show that any adversary $\mathcal{A}$ against the IND-CPA security of Hashed ElGamal that makes at most q_H queries to the hash oracle H can be used to construct an adversary $\mathcal{B}$ that achieves an advantage $q_H^{-1}\ \mathbf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}$ in solving the CDH problem. The reduction is done in the random oracle model, where hash functions are modeled as truly random functions. We represent random oracles using stateful procedures; queries are answered consistently: if some value is queried twice, the same response is given. For instance, we code the hash function H as follows:

Oracle $H(\lambda)$:
 if $\lambda \notin \mathsf{dom}(\boldsymbol{L})$ then
  $h \xleftarrow{\$} \{0,1\}^{\ell}$;
  $\boldsymbol{L} \leftarrow (\lambda, h) :: \boldsymbol{L}$
 else $h \leftarrow \boldsymbol{L}[\lambda]$
 return h

The proof is sketched in Figure 3. We follow the convention of typesetting global variables in boldface. The figure shows the sequence of games used to relate the success of the IND-CPA adversary in the original attack game to the success of the CDH adversary $\mathcal{B}$; the definition of the hash oracle is shown alongside each game. As in the proof of the semantic security of ElGamal, we begin by inlining the calls to $\mathcal{KG}$ and $\mathcal{E}$ in the IND-CPA game to obtain an observationally equivalent game G_1 such that

$$\Pr\left[\mathsf{IND\text{-}CPA} : b = \overline{b}\right] = \Pr\left[\mathsf{G}_1 : b = \overline{b}\right] \tag{6}$$

We then fix the value $\hat{\boldsymbol{h}}$ that the hash oracle gives in response to g^{xy}. This is an instance of the *lazy sampling* transformation: any value that is randomly sampled at some point in a program can be sampled in advance, somewhere earlier in the program. This transformation is automated in CertiCrypt and is described in greater detail in Section 6. We get

$$\Pr\left[\mathsf{G}_1 : b = \overline{b}\right] = \Pr\left[\mathsf{G}_2 : b = \overline{b}\right] \tag{7}$$

We can then modify the hash oracle so that it does not store in $\boldsymbol{L}$ the response given to a g^{xy} query; this will later let us remove $\hat{\boldsymbol{h}}$ altogether from the code of the hash oracle. We prove that games G_2 and G_3 are equivalent by considering the following relational invariant:

$$\phi_{23} \stackrel{\text{def}}{=} (\boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L}) \Longrightarrow \boldsymbol{L}[\boldsymbol{\Lambda}] = \hat{\boldsymbol{h}})\langle 1\rangle \ \wedge\ \forall \lambda. \lambda \neq \boldsymbol{\Lambda}\langle 1\rangle \Longrightarrow \boldsymbol{L}[\lambda]\langle 1\rangle = \boldsymbol{L}[\lambda]\langle 2\rangle$$

where $e\langle 1\rangle$ (resp. $e\langle 2\rangle$) denotes the value that expression e takes in the left hand side (resp. right hand side) program. Intuitively, this invariant shows that the association list $\boldsymbol{L}$, which represents the memory of the hash oracle, coincides in both programs, except perhaps on an element $\boldsymbol{\Lambda}$, which the list in the program on the left hand side (G_2) necessarily maps to $\hat{\boldsymbol{h}}$. It is easy to prove that the implementations of oracle H in games G_2 and G_3 are semantically equivalent under this invariant and preserve it. Since ϕ_{23} is established just before calling $\mathcal{A}$ and is preserved throughout the games, we can prove by inlining the call to H in game G_2 that

$$\Pr\left[\mathsf{G}_2 : b = \overline{b}\right] = \Pr\left[\mathsf{G}_3 : b = \overline{b}\right] \tag{8}$$

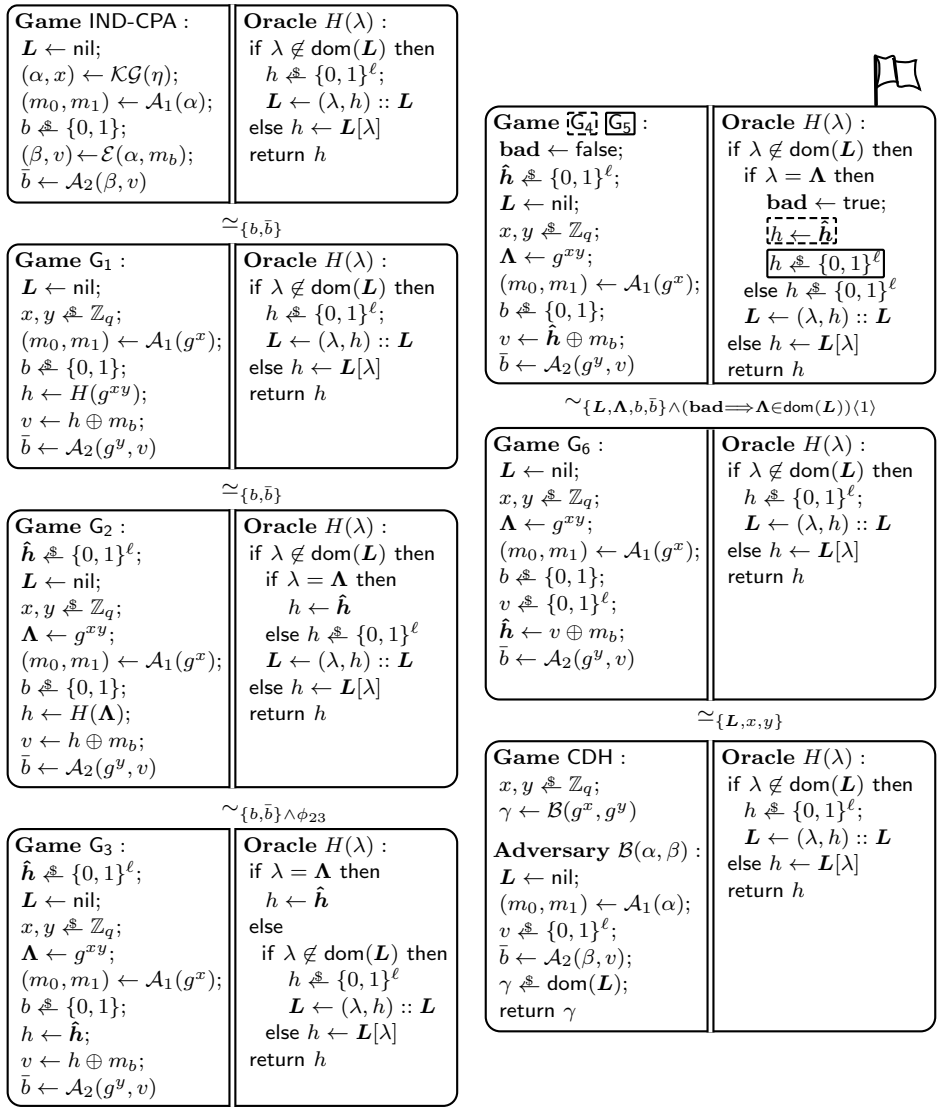

Fig. 3. Game-based proof of semantic security of Hashed ElGamal encryption in the Random Oracle Model.

We then *undo* the previous modification to revert to the previous implementation of the hash oracle and prove that games G_3 and G_4 are observationally equivalent, from which we obtain

$$\Pr\left[\mathsf{G}_3 : b = \bar{b}\right] = \Pr\left[\mathsf{G}_4 : b = \bar{b}\right] \tag{9}$$

Let us now introduce a Boolean flag **bad** that is set at program points where the code of G_4 and G_5 differ. We argue that the difference in the probability of any event in those games is bounded by the probability of **bad** being set in G_5, and

therefore

$$\left|\Pr\left[\mathsf{G}_4 : b = \bar{b}\right] - \Pr\left[\mathsf{G}_5 : b = \bar{b}\right]\right| \leq \Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] \tag{10}$$

This form of reasoning is pervasive in game-based cryptographic proofs and is an instance of the so-called Fundamental Lemma that we discuss in detail in Section 7. In addition, we establish that $\mathbf{bad} \Longrightarrow \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})$ is a post-condition of game G_5 and thus

$$\Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_5 : \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] \tag{11}$$

Since now both branches in the innermost conditional of the hash oracle are equivalent, we coalesce them to recover the original random oracle implementation of H in G_6. We can now use the `swap` tactic to defer the sampling of $\hat{\boldsymbol{h}}$ to the point just before computing v, and substitute

$$v \xleftarrow{\$} \{0,1\}^{\ell};\ \hat{\boldsymbol{h}} \leftarrow v \oplus m_b \qquad \text{for} \qquad \hat{\boldsymbol{h}} \xleftarrow{\$} \{0,1\}^{\ell};\ v \leftarrow \hat{\boldsymbol{h}} \oplus m_b$$

The semantic equivalence of these two program fragments can be proved using the probabilistic relational Hoare logic presented in Section 5.1—a proof is given in Section 6. Hence,

$$\Pr\left[\mathsf{G}_5 : b = \bar{b}\right] = \Pr\left[\mathsf{G}_6 : b = \bar{b}\right] \tag{12}$$

and

$$\Pr\left[\mathsf{G}_5 : \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] = \Pr\left[\mathsf{G}_6 : \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] \tag{13}$$

Observe that $\bar{b}$ does not depend anymore on b in G_6 ($\hat{\boldsymbol{h}} \leftarrow v \oplus m_b$ is dead code), so

$$\Pr\left[\mathsf{G}_6 : b = \bar{b}\right] = \frac{1}{2} \tag{14}$$

We finally construct an adversary $\mathcal{B}$ against CDH that interacts with the adversary $\mathcal{A}$ playing the role of an $\mathsf{IND\text{-}CPA}$ challenger. It returns a random element sampled from the list of queries that adversary $\mathcal{A}$ made to the hash oracle. Observe that $\mathcal{B}$ does not need to know x or y because it gets g^x and g^y as parameters. If the correct answer $\mathbf{\Lambda} = g^{xy}$ to the CDH challenge appears in the list of queries $\boldsymbol{L}$ when the experiment terminates, adversary $\mathcal{B}$ has probability $|\boldsymbol{L}|^{-1}$ of returning it as an answer. Since we know that $\mathcal{A}$ does not make more than q_H queries to the hash oracle, we finally have that

$$\Pr\left[\mathsf{G}_6 : \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] = \Pr\left[\mathsf{G}_6 : g^{xy} \in \mathsf{dom}(\boldsymbol{L})\right] \leq q_H \Pr\left[\mathsf{CDH} : \gamma = g^{xy}\right] \tag{15}$$

To summarize, from Equations (6)—(15) we obtain

$$\begin{aligned}
\left|\Pr\left[\mathsf{IND\text{-}CPA} : b = \bar{b}\right] - 1/2\right| &= \left|\Pr\left[\mathsf{G}_4 : b = \bar{b}\right] - 1/2\right| \\
&= \left|\Pr\left[\mathsf{G}_4 : b = \bar{b}\right] - \Pr\left[\mathsf{G}_6 : b = \bar{b}\right]\right| \\
&= \left|\Pr\left[\mathsf{G}_4 : b = \bar{b}\right] - \Pr\left[\mathsf{G}_5 : b = \bar{b}\right]\right| \\
&\leq \Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] \\
&\leq \Pr\left[\mathsf{G}_6 : \mathbf{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] \\
&\leq q_H \Pr\left[\mathsf{CDH} : \gamma = g^{xy}\right]
\end{aligned}$$

For any adversary $\mathcal{A}$ that executes in polynomial time, we can assume that the bound q_H on the number of queries is polynomial on the security parameter. Under the CDH assumption, the $\mathsf{IND\text{-}CPA}$ advantage of adversary $\mathcal{A}$ must then be

negligible. Otherwise, the adversary $\mathcal{B}$ that we constructed would solve CDH with non-negligible probability, contradicting our computational assumption. To see this, we need to verify that adversary $\mathcal{B}$ runs in probabilistic polynomial time, but this is the case because procedures $\mathcal{A}_1, \mathcal{A}_2$ do, and $\mathcal{B}$ does not perform any additional costly computations. As in the previous example, the proof of this latter fact is completely automated in CertiCrypt.

Hashed ElGamal can also be proved semantically secure in the standard model, but under the stronger DDH assumption. The security reduction can be made under the assumption that the family of hash functions H is *entropy smoothing*—such a family of hash functions can be built without additional assumptions using the Leftover Hash Lemma [Håstad et al. 1999].

3. MATHEMATICAL PRELIMINARIES

3.1 The Unit Interval

The starting point of our formalization is the ALEA Coq library, developed by Paulin-Mohring and described in [Audebaud and Paulin-Mohring 2009]. It provides an axiomatization of the unit interval $[0, 1]$, with the following operations:

Addition:	$(x, y) \mapsto \min(x + y, 1)$, where $+$ denotes addition over reals;
Inversion:	$x \mapsto 1 - x$, where $-$ denotes subtraction over reals;
Multiplication:	$(x, y) \mapsto x \times y$, where $\times$ denotes multiplication over reals;
Division:	$(x, y \neq 0) \mapsto \min(x/y, 1)$, where $/$ denotes division over reals; moreover, if $y = 0$, for convenience division is defined to be 0.

Other useful operations can be derived from these basic operations; for instance the absolute value of the difference of two values $x, y \in [0, 1]$ can be obtained by computing $(x - y) + (y - x)$ and their maximum by computing $(x - y) + y$.

The unit interval can be given an ω-complete partial order (cpo) structure. Recall that an ω-cpo consists of a partially ordered set such that any monotonic sequence has a least upper bound. The unit interval $[0, 1]$ can be given the structure of a ω-cpo by taking as order the usual $\leq$ relation and by defining an operator sup that computes the least upper bound of a monotonic sequence $f : \mathbb{N} \to [0, 1]$ as follows:

$$\sup\ f = \max_{n \in \mathbb{N}} f(n)$$

3.2 Distributions

Programs are interpreted as functions from initial memories to sub-distributions over final memories. To give semantics to most programs used in cryptographic proofs, it would be sufficient to consider sub-distributions with a countable support, which admit a very direct formalization as functions of the form

$$\mu : A \to [0, 1] \qquad \text{such that} \qquad \sum_{x \in A} \mu(x) \leq 1$$

However, it is convenient to take a more general approach and represent instead a distribution over a set A as a probability measure, by giving a function that maps a $[0, 1]$-valued random variable (a function in $A \to [0, 1]$) to its expected value, i.e. the integral of the random variable with respect to the probability measure.

This view of distributions eliminates the need of cluttered definitions and proofs involving summations, and allows us to give a continuation-passing style semantics to programs by defining a suitable monadic structure on distributions. Formally, we represent a distribution on A as a function μ of type

$$\mathcal{D}(A) \stackrel{\text{def}}{=} (A \to [0,1]) \to [0,1]$$

satisfying the following properties:

Monotonicity:	$f \leq g \Longrightarrow \mu\ f \leq \mu\ g$;
Compatibility with inverse:	$\mu\ (\mathbb{1} - f) \leq 1 - \mu\ f$;
Additive linearity:	$f \leq \mathbb{1} - g \Longrightarrow \mu\ (f + g) = \mu\ f + \mu\ g$;
Multiplicative linearity:	$\mu\ (k \times f) = k \times \mu\ f$;
Continuity:	if $f : \mathbb{N} \to (A \to [0,1])$ is monotonic, then $\mu\ (\sup\ f) \leq \sup\ (\mu \circ f)$

Distributions can be interpreted as a monad whose unit and bind operators are defined as follows:

$$\begin{array}{lll} \mathsf{unit} : A \to \mathcal{D}(A) & \stackrel{\text{def}}{=} & \lambda x.\ \lambda f.\ f\ x \\ \mathsf{bind} : \mathcal{D}(A) \to (A \to \mathcal{D}(B)) \to \mathcal{D}(B) & \stackrel{\text{def}}{=} & \lambda \mu.\ \lambda F.\ \lambda f.\ \mu\ (\lambda x.\ (F\ x)\ f) \end{array}$$

These operators satisfy the usual monadic laws

$$\begin{array}{lcl} \mathsf{bind}\ (\mathsf{unit}\ x)\ F & = & F\ x \\ \mathsf{bind}\ \mu\ \mathsf{unit} & = & \mu \\ \mathsf{bind}\ (\mathsf{bind}\ \mu\ F)\ G & = & \mathsf{bind}\ \mu\ (\lambda x.\ \mathsf{bind}\ (F\ x)\ G) \end{array}$$

The monad $\mathcal{D}$ was proposed by Audebaud and Paulin-Mohring [2009] as a variant of the expectation monad used by Ramsey and Pfeffer [2002], and builds on earlier work by Kozen [1981]. It is, in turn, a specialization of the continuation monad $(A \to B) \to B$, with result type $B = [0,1]$.

3.3 Lifting Predicates and Relations to Distributions

For a distribution $\mu : \mathcal{D}(A)$ over a countable set A, we let $\mathsf{support}(\mu)$ denote the set of values in A with positive probability, i.e. its support:

$$\mathsf{support}(\mu) \stackrel{\text{def}}{=} \left\{ x \in A \mid 0 < \mu\ \mathbb{I}_{\{x\}} \right\}$$

where $\mathbb{I}_X$ denotes the indicator function of set X,

$$\mathbb{I}_X \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases}$$

To lift relations to probability distributions we follow the early work of Jonsson et al. [2001] on probabilistic bisimulations.

Definition 3.1 *Lifting predicates to distributions.* Let μ be a distribution on a set A, and P be a predicate on A. We define the lifting of P to μ as follows:

$$\mathsf{range}\ P\ \mu \stackrel{\text{def}}{=} \forall f.\ (\forall x.\ P\ x \Longrightarrow f\ x = 0) \Longrightarrow \mu\ f = 0$$

Definition 3.2 *Lifting relations to distributions.* Let μ_1 be a probability distribution on a set A and μ_2 a probability distribution on a set B. We define the lifting of a relation $R \subseteq A \times B$ to μ_1 and μ_2 as follows:

$$\mu_1 \, \mathcal{L}(R) \, \mu_2 \stackrel{\text{def}}{=} \exists \mu : \mathcal{D}(A \times B). \; \pi_1(\mu) = \mu_1 \; \wedge \; \pi_2(\mu) = \mu_2 \; \wedge \; \mathsf{range} \; R \; \mu \qquad (16)$$

where $\mathsf{range} \; R \; \mu$ stands for the lifting of R, seen as a predicate on pairs in $A \times B$, to distribution μ, and the projections $\pi_1(\mu)$, $\pi_2(\mu)$ of μ are given by

$$\pi_1(\mu) \stackrel{\text{def}}{=} \mathsf{bind} \; \mu \; (\mathsf{unit} \circ \mathsf{fst}) \qquad \pi_2(\mu) \stackrel{\text{def}}{=} \mathsf{bind} \; \mu \; (\mathsf{unit} \circ \mathsf{snd})$$

In contrast to the definition given by Jonsson et al. [2001], the definition above makes sense even when the distributions do not have a countable support. When they do, both definitions coincide; in this case, $\mu_1 \, \mathcal{L}(R) \, \mu_2$ amounts to saying that the probability of each element a in the support of μ_1 can be divided among the elements related to it in such a way that when summing up over these probabilities for an element $b \in B$, one obtains $\mu_2 \; \mathbb{I}_{\{b\}}$.

Let us give an example that conveys a better intuition; suppose one wants to prove $\mathcal{U}_A \, \mathcal{L}(R) \mathcal{U}_B$, where $\mathcal{U}_X$ stands for the uniform probability distribution on a finite set X. When A and B have the same size, proving this is equivalent to exhibiting a bijection $f : A \to B$ such that for every $a \in A$, $R(a, f(a))$ holds. Indeed, using such f it is easy to build a distribution μ on $A \times B$ that satisfies the condition in (16):

$$\mu \stackrel{\text{def}}{=} \mathsf{bind} \; \mathcal{U}_A \; (\lambda a.\mathsf{unit} \; (a, f(a)))$$

This example, as trivial as it may seem, shows that probabilistic reasoning can sometimes be replaced by simpler forms of reasoning. In typical cryptographic proofs, purely probabilistic reasoning is seldom necessary and most *mundane* steps in proofs can be either entirely automated or reduced to verifying simpler conditions, much like in the above example, e.g. showing the existence of a bijection with particular properties.

The way we chose to lift relations over memories to relations over distributions is a generalization to arbitrary relations of the definition of Sabelfeld and Sands [2001] that applies only to equivalence relations. Indeed, there is a simpler but equivalent (see [Jonsson et al. 2001]) way of lifting an equivalence relation to distributions: if R is an equivalence relation on A, then $\mu_1 \, \mathcal{L}(R) \, \mu_2$ holds if and only if for all equivalence classes $[a] \subseteq A$, $\mu_1 \; \mathbb{I}_{[a]} = \mu_2 \; \mathbb{I}_{[a]}$.

Define two functions f and g to be equal modulo a relation Φ iff

$$f =_\Phi g \stackrel{\text{def}}{=} \forall x \; y. \; x \; \Phi \; y \implies f(x) = g(y)$$

It can be easily shown that the above general definition of lifting satisfies

$$\mu_1 \, \mathcal{L}(\Phi) \, \mu_2 \; \wedge \; f =_\Phi g \implies \mu_1 \; f = \mu_2 \; g$$

and analogously for $\leq$. We use this property to prove rules relating observational equivalence to probability in Section 5. It can also be shown that lifting preserves the reflexivity and symmetry of the lifted relation, but proving that it preserves transitivity is not as straightforward. Ideally, one would like to have for probability measures $\mu_1 : \mathcal{D}(A)$, $\mu_2 : \mathcal{D}(B)$, $\mu_3 : \mathcal{D}(C)$ and relations $\Psi \subseteq A \times B$, $\Phi \subseteq B \times C$

$$\mu_1 \, \mathcal{L}(\Psi) \, \mu_2 \; \wedge \; \mu_2 \, \mathcal{L}(\Phi) \, \mu_3 \implies \mu_1 \, \mathcal{L}((\Psi \circ \Phi)) \, \mu_3$$

Proving this for arbitrary distributions requires proving Fubini's theorem for probability measures, which allows to compute integrals with respect to a product measure in terms of iterated integrals with respect to the original measures. Since in practice we consider distributions with a countable support, we do not need the full generality of this result, and we prove it under the assumption that the distribution μ_2 has a countable support, i.e. there exist coefficients $c_i : [0,1]$ and points $b_i : B$ such that

$$\mu_2\ f = \sum_{i=0}^{\infty} c_i f(b_i)$$

LEMMA 3.3. *Consider $d_1 : \mathcal{D}(A)$, $d_2 : \mathcal{D}(B)$, $d_3 : \mathcal{D}(C)$ such that d_2 has countable support. Suppose there exist distributions $\mu_{12} : \mathcal{D}(A \times B)$ and $\mu_{23} : \mathcal{D}(B \times C)$ that make $\mu_1\, \mathcal{L}(\Psi)\, \mu_2$ and $\mu_2\, \mathcal{L}(\Phi)\, \mu_3$ hold. Then, the following distribution over $A \times C$ is a witness for the existential in $\mu_1\, \mathcal{L}((\Psi \circ \Phi))\, \mu_3$:*

$$\begin{aligned}\mu_{13}\ f \stackrel{def}{=}\ \mu_2\ \big(\lambda b.\ \mu_{12}\ \big(\lambda p.\ \big(\mathbb{1}_{\mathsf{snd}(p)=b}/\mu_2\ \mathbb{I}_{\{b\}}\big)\\ \mu_{23}\ \big(\lambda q.\ \big(\mathbb{1}_{\mathsf{fst}(q)=b}/\mu_2\ \mathbb{I}_{\{b\}}\big)\ f(\mathsf{fst}\ p, \mathsf{snd}\ q)\big)\big)\big)\end{aligned}$$

PROOF. The difficult part of the proof is to show that the projections of this distribution coincide with μ_1 and μ_2. For this, we use the fact that μ_2 is discrete to prove that iterative integration with respect to μ_2 and another measure *commutes*, because we can write integration with respect to μ_2 as a summation and we only consider measures that are continue and linear. For instance, to see that the first projection of μ_{13} coincides with μ_1:

$$\begin{aligned}\pi_1(\mu_{13})\ f\ &=\ \sum_{i=0}^{\infty} c_i\ \mu_{12}\ \big(\lambda p.\ \big(\mathbb{1}_{\mathsf{snd}(p)=b_i}/c_i\big)\ \mu_{23}\ \big(\lambda q.\ \big(\mathbb{1}_{\mathsf{fst}(q)=b_i}/c_i\big)\ f(\mathsf{fst}\ p)\big)\big)\\ &=\ \mu_{12}\ \left(\lambda p.\ \sum_{i=0}^{\infty} \mathbb{1}_{\mathsf{snd}(p)=b_i}\ \mu_{23}\ \big(\lambda q.\ \big(\mathbb{1}_{\mathsf{fst}(q)=b_i}/c_i\big)\ f(\mathsf{fst}\ p)\big)\right)\\ &=\ \mu_{12}\ \left(\lambda p.\ f(\mathsf{fst}\ p)\ \sum_{i=0}^{\infty} \mathbb{1}_{\mathsf{snd}(p)=b_i}\right)\\ &=\ \mu_{12}\ (\lambda p.\ f(\mathsf{fst}\ p))\ =\ \mu_1\ f\quad \square\end{aligned}$$

4. GAMES AS PROGRAMS

We describe games as programs in the pWHILE language, a probabilistic extension of an imperative language with procedure calls. This language can be regarded as a mild generalization of the language proposed by Bellare and Rogaway [2006], in that it allows while loops whereas they only consider bounded for loops. The formalization of pWHILE is carefully crafted to exploit key features of Coq: it uses modules to support an extensible expression language that can be adapted according to verification goal, 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.

4.1 The pWHILE Language

We formalize programs in a deep-embedding style, i.e. the syntax of the language is encoded within the proof assistant. 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 to manipulate programs as syntactic objects. This permits to achieve a high level of automation in reasoning about programs through certified tactics that implement syntactic program transformations. Additionally, a deep embedding allows to formalize complexity issues neatly and to reason about programs by induction on the structure of their code.

Given a set $\mathcal{V}$ of variable identifiers, a set $\mathcal{P}$ of procedure identifiers, a set $\mathcal{E}$ of deterministic expressions, and a set $\mathcal{DE}$ of *distribution expressions*, the instructions $\mathcal{I}$ and commands $\mathcal{C}$ of the language can be defined inductively by the clauses:

$$
\begin{array}{rll}
\mathcal{I} ::= & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\
\mid & \mathcal{V} \xleftarrow{\$} \mathcal{DE} & \text{probabilistic assignment} \\
\mid & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
\mid & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{while loop} \\
\mid & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
\mid & \mathsf{assert}\ \mathcal{E} & \text{runtime assertion} \\
\mathcal{C} ::= & \mathsf{skip} & \text{nop} \\
\mid & \mathcal{I};\ \mathcal{C} & \text{sequence}
\end{array}
$$

This inductive definition suffices to understand the remainder of the presentation and the reader may prefer to retain it for further reference. In practice, however, variable and procedure identifiers are annotated with their types, and the syntax of programs is dependently-typed:

$$
\begin{array}{l}
\textbf{Inductive}\ \mathcal{I} : \mathsf{Type} := \\
\mid \mathsf{Assign} : \forall t.\ \mathcal{V}_t \to \mathcal{E}_t \to \mathcal{I} \\
\mid \mathsf{Rand}\ \ : \forall t.\ \mathcal{V}_t \to \mathcal{DE}_t \to \mathcal{I} \\
\mid \mathsf{Cond}\ \ : \mathcal{E}_{\mathbb{B}} \to \mathcal{C} \to \mathcal{C} \to \mathcal{I} \\
\mid \mathsf{While}\ : \mathcal{E}_{\mathbb{B}} \to \mathcal{C} \to \mathcal{I} \\
\mid \mathsf{Call}\ \ \ : \forall l\ t.\ \mathcal{P}_{(l,t)} \to\ \mathcal{V}_t \to {\mathcal{E}_l}^* \to \mathcal{I} \\
\textbf{where}\ \mathcal{C} := \mathcal{I}^*
\end{array}
$$

Thus, the assignment $x \leftarrow e$ is well-formed only if the types of x and e coincide, and $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is well-formed only if e is a Boolean expression and c_1 and c_2 are themselves well-formed. An immediate benefit of using dependent types is that the type system of Coq ensures for free the well-typedness of expressions and commands.

Definition 4.1 *Program.* A program is a pair consisting of a command $c \in \mathcal{C}$ and an environment $E : \forall l\ t.\ \mathcal{P}_{(l,t)} \to \mathsf{decl}_{(l,t)}$, which maps procedure identifiers to their declaration. The declaration of a procedure $p \in \mathcal{P}_{(l,t)}$ consists of its formal parameters, its body, and a return expression,

$$
\mathsf{decl}_{(l,t)} \stackrel{\text{def}}{=} \{\mathsf{args} : {\mathcal{V}_l}^*;\ \mathsf{body} : \mathcal{C};\ \mathsf{re} : \mathcal{E}_t\}
$$

This formalization only allows single-exit procedures. For the sake of readability, we present all examples in a more traditional style, using explicit return statements.

In a typical formalization, the environment will map procedures to closed commands, with the exception of adversaries whose code is unknown, and thus modeled by variables of type $\mathcal{C}$. This is a standard trick to deal with uninterpreted functions in a deep embedding. In the remainder, we will use the terms program and game indistinctly and will sometimes omit environments when they do not add anything to the presentation.

4.2 Semantics of pWHILE Programs

The semantics of commands and expressions depends on a natural number representing the security parameter, which we leave implicit; the interpretation of types may depend on this parameter. Semantics is defined relative to a given memory, i.e. a mapping from variables to values. We let $\mathcal{M}$ denote the set of memories. Since variables are partitioned into local and global variables, we will sometimes represent a memory $m \in \mathcal{M}$ as a pair of mappings $(m.\mathsf{loc}, m.\mathsf{glob})$ for local and global variables, respectively. We let $\varnothing$ denote a mapping associating variables to default values of their respective types. Expressions are deterministic; their semantics is standard and given by a function $[\![\cdot]\!]_{\mathcal{E}}$, that evaluates an expression in a given memory and returns a value. The semantics of distribution expressions is given by a function $[\![\cdot]\!]_{\mathcal{DE}}$. For a distribution expression d of type T, we have that $[\![d]\!]_{\mathcal{DE}} : \mathcal{M} \to \mathcal{D}(X)$, where X is the interpretation of type T. For instance, in Section 2.2 we have used $\{0,1\}^\ell$ to denote the uniform distribution on bitstrings of a certain length ℓ, formally, we have

$$[\![\{0,1\}^\ell]\!]_{\mathcal{DE}}\ m : \mathcal{D}\left(\{0,1\}^\ell\right) \stackrel{\text{def}}{=} \lambda f. \sum_{x \in \{0,1\}^\ell} 2^{-\ell} f(x)$$

Thanks to dependent types, the semantics of expressions and distribution expressions is total. In the following, and whenever there is no confusion, we will drop the subscripts in $[\![\cdot]\!]_{\mathcal{E}}$ and $[\![\cdot]\!]_{\mathcal{DE}}$.

Figure 4 summarizes the denotational semantics of commands. The denotation of a program relates an initial memory to a (sub-)probability distribution over memories using the measure monad presented in the previous section:

$$[\![c]\!] : \mathcal{M} \to \mathcal{D}(\mathcal{M})$$

Note that the function $[\![\cdot]\!]$ maps $\mathcal{M}$ to $\mathcal{D}(\mathcal{M})$, but it is trivial to define a semantic function from $\mathcal{D}(\mathcal{M})$ to $\mathcal{D}(\mathcal{M})$ using the bind operator of the monad.

We have shown that the semantics of programs maps memories to discrete distributions, provided expressions in $\mathcal{DE}$ evaluate to distributions with countable support. We use this together with Lemma 3.3 to prove the soundness of some relational Hoare logic rules (namely, [Comp] and [Trans]) in Section 5.

Computing probabilities. The advantage of using this monadic semantics is that, if we use an arbitrary function as a continuation to the denotation of a program, what we get (for free) as a result is its expected value w.r.t. the distribution of final memories. In particular, we can compute the probability of an event A (represented

$$
\begin{array}{lcl}
[\![\mathsf{skip}]\!]\ m & = & \mathsf{unit}\ m \\
[\![i;\ c]\!]\ m & = & \mathsf{bind}\ ([\![i]\!]\ m)\ [\![c]\!] \\
[\![x \leftarrow e]\!]\ m & = & \mathsf{unit}\ m\,\{[\![e]\!]_{\mathcal{E}}\ m/x\} \\
[\![x \xleftarrow{\$} d]\!]\ m & = & \mathsf{bind}\ ([\![d]\!]_{\mathcal{DE}}\ m)\ (\lambda v.\ \mathsf{unit}\ m\,\{v/x\}) \\
[\![x \leftarrow p(\vec{e})]\!]\ m & = & \mathsf{bind}\ ([\![p.\mathsf{body}]\!]\ (\varnothing\,\{[\![\vec{e}]\!]_{\mathcal{E}}\ m/p.\mathsf{args}\}, m.\mathsf{glob}) \\
 & & \qquad (\lambda m'.\ (m.\mathsf{loc}, m'.\mathsf{glob})\,\{[\![p.\mathsf{re}]\!]_{\mathcal{E}}\ m'/x\}) \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\ m & = & \begin{cases} [\![c_1]\!]\ m & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{true} \\ [\![c_2]\!]\ m & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{false} \end{cases} \\
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]\ m & = & \lambda f.\sup\ (\lambda n.\ [\![[\mathsf{while}\ e\ \mathsf{do}\ c]_n]\!]\ m\ f) \\
 & & \text{where} \\
 & & [\mathsf{while}\ e\ \mathsf{do}\ c]_0 \ = \ \mathsf{assert}\ \neg e \\
 & & [\mathsf{while}\ e\ \mathsf{do}\ c]_{n+1} \ = \ \mathsf{if}\ e\ \mathsf{then}\ c;\ [\mathsf{while}\ e\ \mathsf{do}\ c]_n
\end{array}
$$

Fig. 4. Denotational semantics of pWHILE programs.

as a function in $\mathcal{M} \to \mathbb{B}$) in the distribution obtained after executing a command c in an initial memory m by measuring its characteristic function $\mathbb{1}_A$:

$$\Pr[c, m : A] \ \stackrel{\text{def}}{=}\ [\![c]\!]\ m\ \mathbb{1}_A$$

For instance, one can verify that the denotation of $x \xleftarrow{\$} \{0,1\};\ y \xleftarrow{\$} \{0,1\}$ in an initial memory m is

$$\lambda f.\ \frac{1}{4}\ (f(m\,\{0,0/x,y\}) + f(m\,\{0,1/x,y\}) + f(m\,\{1,0/x,y\}) + f(m\,\{1,1/x,y\}))$$

and conclude that the probability of the event $(x \Rightarrow y)$ after executing the command above is $3/4$.

4.3 Probabilistic Polynomial-Time Programs

In general, cryptographic proofs reason about effective adversaries, that consume polynomially bounded resources. The complexity notion that captures this intuition, and which is pervasive in cryptographic proofs, is that of *strict probabilistic polynomial-time* [Goldreich 2001]. Concretely, a program is said to be strict probabilistic polynomial-time (PPT) whenever there exists a polynomial bound (in some security parameter η) on the cost of each possible execution, regardless of the outcome of its random choices. Said otherwise, a probabilistic program is PPT whenever the same program seen as a non-deterministic program terminates and the cost of each possible run is bounded by a polynomial.

Termination and efficiency are orthogonal. Consider, for instance, the following two programs:

$$
\begin{array}{l}
c_1 \ \stackrel{\text{def}}{=}\ b \leftarrow \mathsf{true};\ \mathsf{while}\ b\ \mathsf{do}\ b \xleftarrow{\$} \{0,1\} \\
c_2 \ \stackrel{\text{def}}{=}\ b \xleftarrow{\$} \{0,1\};\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{while}\ \mathsf{true}\ \mathsf{do}\ \mathsf{skip}
\end{array}
$$

The former terminates with probability 1 (it terminates within n iterations with probability $1 - 2^{-n}$), but may take an arbitrarily large number of iterations to terminate. The latter terminates with probability $1/2$, but when it does, it takes only a constant time. We deal with termination and efficiency separately.

Definition 4.2 *Termination.* The probability that a program c terminates starting from an initial memory m is $\Pr[c, m : \mathsf{true}] = [\![c]\!]\ m\ \mathbb{1}$. We say that a program c is absolutely terminating, and note it $\mathsf{lossless}(c)$, iff it terminates with probability 1 in any initial memory,

$$\mathsf{lossless}(c) \stackrel{\text{def}}{=} \forall m.\ \Pr[c, m : \mathsf{true}] = 1$$

To deal with efficiency, we non-intrusively instrument the semantics of our language to compute the cost of running a program. The instrumented semantics ranges over $\mathcal{D}(\mathcal{M} \times \mathbb{N})$ instead of $\mathcal{D}(\mathcal{M})$. We recall that our semantics is implicitly parametrized by a security parameter η, on which we base our notion of complexity. Our characterization of PPT programs relies on an axiomatization of the execution time and memory usage of expressions:

—We postulate the execution time of each operator, in the form of a function that depends on the inputs of the operator—which corresponds to the so-called functional time model;

—We postulate for each datatype a size measure, in the form of a function that assigns to each value its memory footprint.

We stress that making complexity assumptions on operators is perfectly legitimate. It is a well-known feature of dependent type theories (as is the case of the calculus of Coq) that they cannot express the cost of the computations they purport without using computational reflection, i.e. formalizing an execution model (e.g. probabilistic Turing machines) within the theory itself and proving that functions in type theory denote machines that execute in polynomial time. In our opinion, such a step is overkill. A simpler solution to the problem is to restrict in as much as possible the set of primitive operators, so as to minimize the set of assumptions upon which the complexity proofs rely.

Definition 4.3 *Polynomially bounded distribution.* We say that a family of distributions $\{\mu_\eta : \mathcal{D}(\mathcal{M} \times \mathbb{N})\}$ is (p, q)-bounded, where p and q are polynomials, whenever for every value of the security parameter η and any pair (m, n) occurring with non-zero probability in μ_η, the size of values in m is bounded by $p(\eta)$ and the cost n is bounded by $q(\eta)$. This notion can be formally defined by means of the range predicate introduced in Section 3.3:

$$\mathsf{bounded}(p, q, \mu) \stackrel{\text{def}}{=} \forall \eta.\ \mathsf{range}\ (\lambda(m, n).\ \forall x \in \mathcal{V}.\ |m(x)| \leq p(\eta) \wedge n \leq q(\eta))\ \mu_\eta$$

Definition 4.4 *Strict probabilistic polynomial-time program.* We say that a program c is strict probabilistic polynomial-time (PPT) iff it terminates absolutely, and there exist polynomial transformers F, G such that for every (p, q)-bounded distribution family μ_η, $(\mathsf{bind}\ \mu_\eta\ [\![c]\!])$ is $(F(p), q + G(p))$-bounded.

We can recover some intuition by taking $\mu = \mathsf{unit}\ (m, 0)$ in the above definition. In this case, we may paraphrase the condition as follows: if the size of values in m is bounded by some polynomial p, and an execution of the program in m terminates with non-zero probability in memory m', then the size of values in m' is bounded by the polynomial $F(p)$, and the cost of the execution is bounded by the polynomial $G(p)$. It is in this latter polynomial that bounds the cost of executing the program that we are ultimately interested. The increased complexity in the definition is

needed for proving compositionality results, such as the fact that PPT programs are closed under sequential composition.

Although our formalization of termination and efficiency relies on semantic definitions, it is not necessary for users to reason directly about the semantics of a program to prove it meets those definitions. CertiCrypt implements a certified algorithm showing that every program without loops and recursive calls terminates absolutely.[2] We also provide another algorithm that, together with the first, establishes that a program is PPT provided that, additionally, the program does not contain expressions that might generate values of super-polynomial size or take a super-polynomial time when evaluated in a polynomially bounded memory.

Exact bounds on execution time. Extracting an exact security result from a reductionist game-based proof requires to lower bound the success probability of the reduction and to upper bound the overhead incurred in execution time. Computing a bound on the success probability is what takes most of the effort since it requires examining the whole sequence of games and a careful bookkeeping of the probability of events. On the other hand, bounding the overhead of a reduction only requires examining the last game in the sequence. While we have put a great effort in automating the computation of probability bounds and we developed an automated method to obtain asymptotic polynomial bounds on the execution time of reductions, we did not bother to provide a method to compute exact time bounds. To do so, we would need an alternative cost-instrumented semantics that does not take into account the time spent in evaluating calls to oracles, but instead just records the number of queries that have been made. Assume that an adversary $\mathcal{A}$ executes within time t (without taking into account oracle calls) and makes at most $q_{\mathcal{O}_i}$ queries to oracle $\mathcal{O}_i$. Suppose we have a reduction where an adversary $\mathcal{B}$ uses $\mathcal{A}$ as a sub-procedure; assume wlog that $\mathcal{B}$ only calls $\mathcal{A}$ once and does not make any additional oracle calls. Then, we can argue that if $\mathcal{B}$ executes within time t' without taking into account the cost of evaluating calls to $\mathcal{A}$ (this could easily be computed by considering $\mathcal{A}$ as an oracle for $\mathcal{B}$), then $\mathcal{B}$ executes within time $t + t' + \sum_i q_{\mathcal{O}_i}\ t_{\mathcal{O}_i}$ where $t_{\mathcal{O}_i}$ upper bounds the cost one query to oracle $\mathcal{O}_i$.

4.4 Adversaries

In order to reason about games in the presence of unknown adversaries, we must specify an interface for adversaries and construct proofs under the assumption that adversaries are well-formed against their specification. Assuming that adversaries respect their interface provides us with an induction principle to reason over all (well-formed) adversaries. We make an extensive use of this induction principle: each time a proof system is introduced, the principle allows us to establish proof rules for adversaries. Likewise, each time we implement a program transformation, the induction principle allows us to prove the correctness of the transformation for programs that contain procedure calls to adversaries.

Formally, the interface of an adversary consists of a triple $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$, where $\mathcal{O}$ is the set of procedures that the adversary may call, $\mathcal{RW}$ the set of variables that it

[2]It is of course a weak result in terms of termination of probabilistic programs, but nevertheless sufficient as regards cryptographic applications. Extending our formalization to a certified termination analysis for loops is interesting, but orthogonal to our main goals.

$$I \vdash \mathsf{skip} : I \qquad \frac{I \vdash i : I' \quad I' \vdash c : O}{I \vdash i; c : O} \qquad \frac{\mathsf{writable}(x) \quad \mathsf{fv}(e) \subseteq I}{I \vdash x \leftarrow e : I \cup \{x\}} \qquad \frac{\mathsf{writable}(x) \quad \mathsf{fv}(d) \subseteq I}{I \vdash x \xleftarrow{\$} d : I \cup \{x\}}$$

$$\frac{\mathsf{fv}(e) \subseteq I \quad I \vdash c_1 : O_1 \quad I \vdash c_2 : O_2}{I \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : O_1 \cap O_2} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash c : I}{I \vdash \mathsf{while}\ e\ \mathsf{do}\ c : I}$$

$$\frac{\mathsf{fv}(\vec{e}) \subseteq I \quad \mathsf{writable}(x) \quad p \in \mathcal{O}}{I \vdash x \leftarrow p(\vec{e}) : I \cup \{x\}} \qquad \frac{\mathsf{fv}(\vec{e}) \subseteq I \quad \mathsf{writable}(x) \quad \vdash_{\mathrm{wf}} \mathcal{B}}{I \vdash x \leftarrow \mathcal{B}(\vec{e}) : I \cup \{x\}}$$

$$\frac{\mathcal{RW} \cup \mathcal{R} \cup \mathcal{A}.\mathsf{args} \vdash \mathcal{A}.\mathsf{body} : O \quad \mathsf{fv}(\mathcal{A}.\mathsf{re}) \subseteq O}{\vdash_{\mathrm{wf}} \mathcal{A}}$$

$$\mathsf{writable}(x) \stackrel{\mathrm{def}}{=} \mathsf{local}(x) \vee x \in \mathcal{RW}$$

Fig. 5. Rules for well-formedness of an adversary against interface $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$. A judgment of the form $I \vdash c : O$ can be interpreted as follows: assuming variables in I may be read, the adversarial code fragment c respects the interface and after its execution variables in O may be read. Thus, if $I \vdash c : O$, then $I \subseteq O$.

may read and write, and $\mathcal{R}$ the set of variables that it may only read. We say that an adversary $\mathcal{A}$ with interface $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$ is well-formed if the judgment $\vdash_{\mathrm{wf}} \mathcal{A}$ can be derived from the rules in Fig. 5. Note that the rules are generic, only making sure that the adversary makes a correct use of variables and procedures. These rules guarantee that a well-formed adversary always initializes local variables before using them, only writes global variables in $\mathcal{RW}$ and only reads global variables in $\mathcal{RW} \cup \mathcal{R}$. For convenience, we allow adversaries to call procedures outside $\mathcal{O}$, but these procedures must themselves respect the same interface.

Additional constraints may be imposed on adversaries. For example, exact security proofs usually impose an upper bound to the number of calls adversaries can make to a given oracle, while some properties, such as IND-CCA (see §8.2), restrict the parameters with which oracles may be called at different stages in an experiment. Likewise, some proofs impose extra conditions such as forbidding repeated or malformed queries. These kinds of properties can be formalized using global variables that record calls to oracles and verifying as post-condition that all calls were legitimate.

5. RELATIONAL HOARE LOGIC

Shoup [2004] classifies steps in cryptographic proofs into three categories:

(1) Transitions based on indistinguishability, which are typically justified by appealing to a decisional assumption (e.g. the DDH assumption);
(2) Transitions based on failure events, where it is argued that two games behave identically unless a failure event occurs;
(3) Bridging steps, which correspond to refactoring the code of games in a way that is not observable by adversaries. This is in general done to prepare the ground for applying a *lossy* transition of one of the above two classes.

A bridging step from a game G_1 to a game G_2 typically replaces a program fragment c_1 by an observationally equivalent fragment c_2. In general, however, c_1 and c_2 are observationally equivalent only in the particular context where the substitution is done. We justify such transitions through a relational Hoare logic that generalizes

observational equivalence through pre- and post-conditions that characterize the context where the substitution is valid. This relational Hoare logic may as well be used to establish (in)equalities between the probability of events in two games (as shown by the rules [PrEq] and [PrLe] below) and to establish program invariants that serve to justify other program transformations or more complex probabilistic reasoning.

5.1 Probabilistic Relational Hoare Logic (pRHL)

The relational Hoare logic that we propose elaborates on and extends to probabilistic programs Benton's [2004] relational Hoare logic. Benton's logic uses judgments of the form $\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi$, that relate two programs, c_1 and c_2, w.r.t. a pre-condition Ψ and a post-condition Φ, both defined as relations on deterministic states. Such a judgment states that for every pair of initial memories m_1, m_2 satisfying the pre-condition Ψ, if the evaluations of c_1 in m_1 and c_2 in m_2 terminate with final memories m_1' and m_2' respectively, then $m_1' \ \Phi \ m_2'$ holds. In a probabilistic setting, the evaluation of a program in an initial memory yields instead a (sub-)probability distribution over program memories. In order to give a meaning to a judgment like the above one, we therefore need to lift relations over memories to relations over distributions.[3] We use the mechanism presented in Section 3.

Definition 5.1 *pRHL judgment.* We say that two programs c_1 and c_2 are equivalent with respect to pre-condition Ψ and post-condition Φ iff

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=} \forall m_1\ m_2.\ m_1\ \Psi\ m_2 \Longrightarrow (\llbracket c_1 \rrbracket\ m_1)\, \mathcal{L}(\Phi)\, (\llbracket c_2 \rrbracket\ m_2)$$

We say that two programs c_1 and c_2 are semantically equivalent, and note it as $\models c_1 \equiv c_2$, if they are equivalent w.r.t equality on memories as pre- and post-condition.

Rather than defining the rules for pRHL and proving them sound in terms of the meaning of judgments, we place ourselves in a semantic setting and derive the rules as lemmas. This allows to easily extend the system by deriving extra rules, or even to resort to the semantic definition if the system turns out to be insufficient. Figure 6 gathers some representative derived rules. Most rules admit, in addition to their symmetrical version of Fig. 6, one-sided (left and right) variants, e.g. for assignments

$$\frac{m_1\ \Psi\ m_2 = (m_1\,\{\llbracket e_1 \rrbracket m_1 / x_1\})\ \Phi\ m_2}{\models x_1 \leftarrow e_1 \sim \mathsf{skip} : \Psi \Rightarrow \Phi}[\text{Assn1}]$$

The rule [Case] allows to reason by case analysis on the evaluation of an arbitrary relation in the initial memories. Together with simple rules in the spirit of

$$\frac{\models c_1 \sim c : \Psi \wedge e\langle 1 \rangle \Rightarrow \Phi}{\models \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sim c : \Psi \wedge e\langle 1 \rangle \Rightarrow \Phi}[\text{Cond1T}]$$

it subsumes [Cond] and allows to prove judgments that otherwise would not be derivable, such as the semantic equivalence of the programs ($\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$)

[3] An alternative would be to develop a logic in which Ψ and Φ are relations over distributions. However, we do not believe such a logic would allow a similar level of proof automation.

$$\models \mathsf{skip} \sim \mathsf{skip} : \Phi \Rightarrow \Phi \ [\text{Skip}] \qquad \frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Theta \quad \models c_1' \sim c_2' : \Theta \Rightarrow \Phi}{\models c_1; c_1' \sim c_2; c_2' : \Psi \Rightarrow \Phi}[\text{Seq}]$$

$$\frac{m_1 \ \Psi \ m_2 = (m_1 \{[\![e_1]\!] m_1 / x_1\}) \ \Phi \ (m_2 \{[\![e_2]\!] m_2 / x_2\})}{\models x_1 \leftarrow e_1 \sim x_2 \leftarrow e_2 : \Psi \Rightarrow \Phi}[\text{Assn}]$$

$$\frac{m_1 \ \Psi \ m_2 \Longrightarrow ([\![d_1]\!] \ m_1) \, \mathcal{L}(\Theta) \, ([\![d_2]\!] \ m_2) \quad \text{where } v_1 \ \Theta \ v_2 = (m_1 \{v_1/x_1\}) \ \Phi \ (m_2 \{v_2/x_2\})}{\models x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Psi \Rightarrow \Phi}[\text{Rnd}]$$

$$\frac{\begin{array}{c} m_1 \ \Psi \ m_2 \Longrightarrow [\![e_1]\!] \ m_1 = [\![e_2]\!] \ m_2 \\ \models c_1 \sim c_2 : \Psi \wedge e_1\langle 1\rangle \Rightarrow \Phi \quad \models c_1' \sim c_2' : \Psi \wedge \neg e_1\langle 1\rangle \Rightarrow \Phi \end{array}}{\models \mathsf{if} \ e_1 \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_1' \sim \mathsf{if} \ e_2 \ \mathsf{then} \ c_2 \ \mathsf{else} \ c_2' : \Psi \Rightarrow \Phi}[\text{Cond}]$$

$$\frac{m_1 \ \Phi \ m_2 \Longrightarrow [\![e_1]\!] \ m_1 = [\![e_2]\!] \ m_2 \quad \models c_1 \sim c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Phi}{\models \mathsf{while} \ e_1 \ \mathsf{do} \ c_1 \sim \mathsf{while} \ e_2 \ \mathsf{do} \ c_2 : \Phi \Rightarrow \Phi \wedge \neg e_1\langle 1\rangle}[\text{While}]$$

$$\frac{\Psi \subseteq \Psi' \quad \models c_1 \sim c_2 : \Psi' \Rightarrow \Phi' \quad \Phi' \subseteq \Phi}{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi}[\text{Sub}] \qquad \frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \mathsf{SYM}(\Psi) \quad \mathsf{SYM}(\Phi)}{\models c_2 \sim c_1 : \Psi \Rightarrow \Phi}[\text{Sym}]$$

$$\models c \equiv c \ [\text{Refl}] \qquad \frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \models c_2 \sim c_3 : \Psi \Rightarrow \Phi \quad \mathsf{PER}(\Psi) \quad \mathsf{PER}(\Phi)}{\models c_1 \sim c_3 : \Psi \Rightarrow \Phi}[\text{Trans}]$$

$$\frac{\models c_1 \sim c_2 : \Psi \wedge \Psi' \Rightarrow \Phi \quad \models c_1 \sim c_2 : \Psi \wedge \neg\Psi' \Rightarrow \Phi}{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi}[\text{Case}]$$

Fig. 6. Selection of derived rules of pRHL.

and (if $\neg e$ then c_2 else c_1):

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\models c_1 \sim c_1 : = \wedge \neg\neg e\langle 2\rangle \Rightarrow =}[\text{Sub,Refl}]}{\models c_1 \sim \mathsf{if} \ \neg e \ \mathsf{then} \ c_2 \ \mathsf{else} \ c_1 : = \wedge \neg\neg e\langle 2\rangle \Rightarrow =}[\text{Cond2F}]}{\models c_1 \sim \mathsf{if} \ \neg e \ \mathsf{then} \ c_2 \ \mathsf{else} \ c_1 : = \wedge \ e\langle 1\rangle \Rightarrow =}[\text{Sub}]}{\models \mathsf{if} \ e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \sim \mathsf{if} \ \neg e \ \mathsf{then} \ c_2 \ \mathsf{else} \ c_1 : = \wedge \ e\langle 1\rangle \Rightarrow =}[\text{Cond1T}] \ \cdots}{\models \mathsf{if} \ e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \equiv \mathsf{if} \ \neg e \ \mathsf{then} \ c_2 \ \mathsf{else} \ c_1}[\text{Case}]$$

We use [Case] as well to justify the correctness of dataflow analyses that exploit the information provided by entering branches.

The rule [Sym] can be generalized by taking the inverse of the relations instead of requiring that pre- and post-condition be symmetric:

$$\frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi}{\models c_2 \sim c_1 : \Psi^{-1} \Rightarrow \Phi^{-1}}[\text{Inv}]$$

The rule [Trans], although appealing, is of limited practical use. Consider, for instance, "independent" pre- and post-conditions of the form

$$m_1 \ \Psi \ m_2 \stackrel{\text{def}}{=} \Psi_1 \ m_1 \wedge \Psi_2 \ m_2 \qquad m_1 \ \Phi \ m_2 \stackrel{\text{def}}{=} \Phi_1 \ m_1 \wedge \Phi_2 \ m_2$$

In order to apply the rule [Trans], we are essentially forced to have $\Psi_1 = \Psi_2$ and $\Phi_1 = \Phi_2$, and we must also choose the same pre- and post-condition for the intermediate game c_2. This constraints make the rule [Trans] impractical in some

cases; we use instead the rule [Comp] to introduce intermediate games:

$$\frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \models c_2 \sim c_3 : \Psi' \Rightarrow \Phi'}{\models c_1 \sim c_3 : \Psi \circ \Psi' \Rightarrow \Phi \circ \Phi'}[\text{Comp}]$$

The soundness of this rule relies on Lemma 3.3 and on the fact that the denotation of a program maps an initial memory to a distribution with countable support. This is true if we only allow values to be sampled from distributions with countable support, a reasonable restriction that does not affect our application to cryptographic proofs.

We can specialize rule [Rnd] when the distributions from where random values are sampled have countable support. In this case, there is a simpler condition that makes the hypothesis of the rule hold. We say that two distributions $\mu_1 : \mathcal{D}(A)$ and $\mu_2 : \mathcal{D}(B)$ with countable support are equivalent modulo a relation $R \subseteq A \times B$, and note it $\mu_1 \simeq_R \mu_2$, when there exists a bijection $f : \mathsf{support}(\mu_1) \to \mathsf{support}(\mu_2)$ such that

$$\forall a \in \mathsf{support}(\mu_1).\ \mu_1\ \mathbb{I}_{\{a\}} = \mu_2\ \mathbb{I}_{\{f(a)\}}\ \wedge\ R(a, f(a))$$

We can prove that the following rule is sound:

$$\frac{m_1\ \Psi\ m_2 \Longrightarrow [\![d_1]\!]m_1 \simeq_\Theta [\![d_2]\!]m_2 \quad v_1\ \Theta\ v_2 = (m_1\,\{v_1/x_1\})\ \Phi\ (m_2\,\{v_2/x_2\})}{\models x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Psi \Rightarrow \Phi}[\text{Perm}]$$

If d_1 and d_2 are both interpreted as uniform distributions over some set of values, the premise of the rule boils down to exhibiting a bijection f between the supports of $([\![d_1]\!]m_1)$ and $([\![d_2]\!]m_2)$ such that $\Theta(v, f(v))$ holds for any v in the support of $[\![d_1]\!]m_1$. To see that the rule is sound, note that $\mu_1 \simeq_R \mu_2$ implies $\mu_1\,\mathcal{L}(R)\,\mu_2$; it suffices to take the following distribution as a witness for the existential:

$$\mu \stackrel{\text{def}}{=} \mathsf{bind}\ \mu_1\ (\lambda v.\ \mathsf{unit}(v, f(v)))$$

Hence, the soundness of the above rule is immediate from the soundness of rule [Rnd]. Section 6.2 shows that rule [Perm] is enough to prove several program equivalences appearing in cryptographic proofs. However, observe that rule [Perm] is far from being complete as shown by the following program equivalence that cannot be derived using only this rule:

$$\models a \xleftarrow{\$} [0..1] \sim b \xleftarrow{\$} [0..3];\ a \leftarrow b \bmod 2 : \mathsf{true} \Rightarrow =_{\{a\}}$$

One cannot use the above rule to prove such a equivalence because the supports of the distributions from where random values are sampled in the programs do not have the same size and hence it is not possible to find a bijection relating them. We can further generalize the rule to prove the above equivalence by requiring instead the existence of a bijection between the support of one distribution and a partition of the support of the other, as in the following rule:

$$\frac{\begin{array}{l} m_1\ \Psi\ m_2 \Longrightarrow \mathsf{let}\ S_1 = \mathsf{support}([\![d_1]\!]\ m_1),\ S_2 = \mathsf{support}([\![d_2]\!]\ m_2)\ \mathsf{in} \\ \quad \exists f : S_1 \to \mathcal{P}(S_2).\ \bigcup_{v \in S_1} f(v) = S_2\ \wedge\ (\forall v_1, v_2 \in S_1.\ f(v_1) \cap f(v_2) = \emptyset)\ \wedge \\ \qquad \big(\forall v \in S_1.\ \mu_1\ \mathbb{I}_{\{v\}} = \mu_2\ \mathbb{I}_{f(v)}\ \wedge\ \forall w \in f(v).\ (m_1\,\{v/x_1\})\ \Phi\ (m_2\,\{w/x_2\})\big) \end{array}}{\models x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Psi \Rightarrow \Phi}$$

The following two rules allow to fall back from the world of pRHL into the world of probabilities, in which security statements are expressed:

$$\frac{m_1 \ \Psi \ m_2 \quad \models c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \Phi \Longrightarrow (A\langle 1\rangle \iff B\langle 2\rangle)}{\Pr[c_1, m_1 : A] \ = \ \Pr[c_2, m_2 : B]}[\text{PrEq}]$$

and analogously,

$$\frac{m_1 \ \Psi \ m_2 \quad \models c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \Phi \Longrightarrow (A\langle 1\rangle \implies B\langle 2\rangle)}{\Pr[c_1, m_1 : A] \ \leq \ \Pr[c_2, m_2 : B]}[\text{PrLe}]$$

By taking $A = B = \mathsf{true}$ we can observe that observational equivalence enjoys some form of termination sensitivity:

$$(\models c_1 \sim c_2 : \Psi \Rightarrow \Phi) \wedge m_1 \ \Psi \ m_2 \Longrightarrow [\![c_1]\!] \ m_1 \ \mathbb{1} = [\![c_2]\!] \ m_2 \ \mathbb{1}$$

We conclude with an example that nicely illustrates some of the intricacies of pRHL. Let $c = b \xleftarrow{\$} \{0,1\}$ and $\Phi = (b\langle 1\rangle = b\langle 2\rangle)$. We have for any pair of initial memories $([\![c]\!] \ m_1) \, \mathcal{L}(\Phi) \, ([\![c]\!] \ m_2)$. Indeed, the following distribution is a witness for the existential of the lifting:

$$\mu \ f \ = \ \frac{1}{2} f(m_1\{0/b\}, m_2\{0/b\}) \ + \ \frac{1}{2} f(m_1\{1/b\}, m_2\{1/b\})$$

Perhaps more surprisingly, we also have $([\![c]\!] \ m_1) \, \mathcal{L}(\neg\Phi) \, ([\![c]\!] \ m_2)$, for which it suffices to take the following distribution as a witness for the existential:

$$\mu'(f) = \frac{1}{2} f(m_1\{0/b\}, m_2\{1/b\}) + \frac{1}{2} f(m_1\{1/b\}, m_2\{0/b\})$$

Thus, we have at the same time $\models c \sim c : \mathsf{true} \Rightarrow \Phi$ and $\models c \sim c : \mathsf{true} \Rightarrow \neg\Phi$ (but of course not $\models c \sim c : \mathsf{true} \Rightarrow \mathsf{false}$) and as a consequence the "obvious" rule

$$\frac{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \qquad \models c_1 \sim c_2 : \Psi \Rightarrow \Phi'}{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \wedge \Phi'}$$

is unsound. While this example may seem unintuitive or even inconsistent if one reasons in terms of deterministic states, its intuitive significance in a probabilistic setting is that observing either Φ or $\neg\Phi$ is not enough to tell apart the distributions resulting from two executions of c. This example shows why lifting a relation to distributions involves an existential quantification, and why it is not possible to always use the product distribution as a witness (one cannot establish neither of the above judgments using the product distribution). This interpretation of pRHL judgments is strongly connected to the relation between relational logics and information flow [Amtoft et al. 2006; Benton 2004]—formally characterized for instance by Benton's embedding of a type system for secure information flow into RHL.

As an additional example, observe that we have

$$\models x \xleftarrow{\$} \{0,1\}; \ y \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\}; \ y \leftarrow x : \mathsf{true} \Rightarrow =_{\{x\}}$$
$$\models x \xleftarrow{\$} \{0,1\}; \ y \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\}; \ y \leftarrow x : \mathsf{true} \Rightarrow =_{\{y\}}$$

but clearly the following judgment does not hold

$$\models x \xleftarrow{\$} \{0,1\}; \ y \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\}; \ y \leftarrow x : \mathsf{true} \Rightarrow =_{\{x,y\}}$$

since after executing the program on the right-hand side the values of x and y always coincide while this happens only with probability $1/2$ for the program on the left-hand side.

5.2 Observational Equivalence

Observational equivalence is derived as an instance of relational Hoare judgments in which pre- and post-conditions are restricted to equality over a subset of program variables. Observational equivalence of programs c_1, c_2 w.r.t. an input set of variables I and an output set of variables O is defined as

$$\models c_1 \simeq_O^I c_2 \stackrel{\text{def}}{=} \models c_1 \sim c_2 : =_I \Rightarrow =_O$$

The rules of pRHL can be specialized to the case of observational equivalence. For example, for conditional statements we have

$$\frac{m_1 =_I m_2 \Longrightarrow [\![e_1]\!]\, m_1 = [\![e_2]\!]\, m_2 \quad \models c_1 \simeq_O^I c_2 \quad \models c_1' \simeq_O^I c_2'}{\models \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \simeq_O^I \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2'}$$

It follows that observational equivalence is symmetric and transitive, although it is not reflexive. Indeed, observational equivalence can be seen as a generalization of probabilistic non-interference: if we take $I = O = L$, the set of *low* variables, then c is non-interferent iff $\models c \simeq_L^L c$.

Observational equivalence is more amenable to mechanization than full-fledged pRHL. To support automation, CertiCrypt implements a calculus of variable dependencies and provides a function eqobs_in, that given a program c and a set of output variables O, computes a set of input variables I such that $\models c \simeq_O^I c$. Analogously, it provides a function eqobs_out, that given a set of input variables I, computes a set of output variables O such that $\models c \simeq_O^I c$. This allows a simple procedure to establish a self-equivalence of the form $\models c_1 \simeq_O^I c_2$: just compute a set I' such that $\models c \simeq_O^{I'} c$ using eqobs_in and check whether $I' \subseteq I$, or equivalently, compute a set O' such that $\models c \simeq_{O'}^I c$ using eqobs_out and check whether $O \subseteq O'$.

CertiCrypt provides as well a (sound, but incomplete) relational weakest precondition calculus that can be used to automate proofs of program invariants; it deals with judgments of the form

$$\models c_1 \sim c_2 : \Psi \Rightarrow =_O \wedge\, \Phi$$

and requires that the programs have (almost) the same control-flow structure.

6. PROOF METHODS FOR BRIDGING STEPS

CertiCrypt provides a powerful set of tactics and algebraic equivalences to automate bridging steps in proofs. Most tactics rely on an implementation of a certified optimizer for pWhile. Algebraic equivalences are provided as lemmas that follow from algebraic properties of the interpretation of language constructs.

6.1 Certified Program Transformations

We automate several transformations that consist in applying compiler optimizations. More precisely, we provide support for a rich set of transformations based on dependency and dataflow analyses, and for inlining procedure calls in programs.

Each transformation is implemented as a function in CertiCrypt that performs the transformation itself, together with a rule that proves its correctness and a tactic that applies the rule backwards.

Transformations based on dependencies. The functions eqobs_in and eqobs_out and the relational Hoare logic presented in Section 5.1 provide the foundations to support transformations such as dead code elimination and code reordering.

We write and prove the correctness of a function context that strips off two programs c_1 and c_2 their maximal common context relative to sets I and O of input and output variables. The correctness of context is expressed by the following rule

$$\frac{\mathsf{context}(I, c_1, c_2, O) = (I', c_1', c_2', O') \quad \models c_1' \simeq^{I'}_{O'} c_2'}{\models c_1 \simeq^{I}_{O} c_2}$$

The tactic `eqobs_ctxt` applies this rule backwards. Using the same idea, we implement tactics that strip off two programs only their common prefix (`eqobs_hd`) or suffix (`eqobs_tl`).

We provide a tactic (`swap`) that given two programs tries to hoist their common instructions to obtain a maximal common prefix[4], which can then be eliminated using the above tactics. Its correctness is based on the rule

$$\frac{\begin{array}{c}\models c_1 \simeq^{I_1}_{O_1} c_1 \quad \models c_2 \simeq^{I_2}_{O_2} c_2 \quad \mathsf{mod}(c_1, O_1) \quad \mathsf{mod}(c_2, O_2) \\ O_1 \cap O_2 = \emptyset \quad I_1 \cap O_2 = \emptyset \quad I_2 \cap O_1 = \emptyset\end{array}}{\models c_1;\ c_2 \equiv c_2;\ c_1}$$

where $\mathsf{mod}(c, X)$ is a semantic predicate expressing that program c only modifies variables in X. This is formally expressed by

$$\forall m.\ \mathsf{range}\ (\lambda m'.\ m =_{\mathcal{V}\setminus X} m')\ (\llbracket c \rrbracket\ m)$$

which ensures that reachable final memories coincide with the initial memory except maybe on variables in X. The tactic `swap` uses an algorithm that over-approximates the set of modified variables to decide whether two instructions can be swapped.

We provide a tactic (`deadcode`) that performs dead code elimination relative to a set O of output variables. The corresponding transformation behaves more like an aggressive slicing algorithm, i.e., it removes portions of code that do not affect variables in O and performs at the same time branch prediction (substituting c_1 for $\mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2$), branch coalescing (substituting c for $\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c$), and self-assignment elimination. Its correctness relies on the rule

$$\frac{\mathsf{mod}(c, X) \quad \mathsf{lossless}(c) \quad \mathsf{fv}(\Phi) \cap X = \emptyset}{\models c \sim \mathsf{skip} : \Phi \Rightarrow \Phi}$$

Optimizations based on dataflow analyses. CertiCrypt has built-in, generic, support for such optimizations: given an abstract domain D (a semi-lattice) for the analysis, transfer functions for assignment and branching instructions, and an operator that optimizes expressions in the language, we construct a certified optimization function $\mathsf{optimize} : \mathcal{C} \to D \to \mathcal{C} \times D$. When given a command c and an

[4]One could also provide a complementary tactic that hoists instructions to obtain a maximal common suffix.

element $\delta \in D$, this function transforms c into its optimized version c' assuming the validity of δ. In addition, it returns an abstract post-condition $\delta' \in D$, valid after executing c (or c'). We use these abstract post-conditions to state the correctness of the optimization and to apply it recursively. The correctness of optimize is proved using a mixture of the techniques of [Benton 2004] and [Bertot et al. 2006; Leroy 2006]: we express the validity of the information contained in the analysis domain using a predicate $\mathsf{valid}(\delta, m)$ that states the agreement between the compile time abstract values in δ and the run time memory m. Correctness is expressed in terms of a pRHL judgment:

$$\textbf{let } (c', \delta') := \mathsf{optimize}(c, \delta) \textbf{ in } \models c \sim c' : \asymp_\delta \Rightarrow \asymp_{\delta'}$$

where $m_1 \asymp_\delta m_2 \stackrel{\text{def}}{=} m_1 = m_2 \wedge \mathsf{valid}(\delta, m_1)$. The following useful rule is derived using [Comp]:

$$\frac{m_1 \ \Psi \ m_2 \Longrightarrow \mathsf{valid}(\delta, m_1) \quad \mathsf{optimize}(c_1, \delta) = (c_1', \delta') \quad \models c_1' \sim c_2 : \Psi \Rightarrow \Phi}{\models c_1 \sim c_2 : \Psi \Rightarrow \Phi}[\text{Opt}]$$

Our case studies extensively use instantiations of [Opt] to perform expression propagation (tactic ep). In contrast, we found that common subexpression elimination is seldom used.

6.2 Algebraic Equivalences

Bridging steps frequently make use of algebraic properties of language constructs. The proof of semantic security of ElGamal uses the fact that in a cyclic multiplicative group, multiplication by a uniformly sampled element acts as a one-time pad:

$$\models x \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ \alpha \leftarrow g^x \times \beta \simeq_{\{\alpha\}} y \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ \alpha \leftarrow g^y$$

In the proof of IND-CCA security of OAEP described in Section 8.2 we use the equivalences

$$\models x \stackrel{\$}{\leftarrow} \{0,1\}^k;\ y \leftarrow x \oplus z \simeq^{\{z\}}_{\{x,y,z\}} y \stackrel{\$}{\leftarrow} \{0,1\}^k;\ x \leftarrow y \oplus z$$

and (for a permutation f):

$$\models x \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho};\ y \stackrel{\$}{\leftarrow} \{0,1\}^{\rho};\ z \leftarrow f(x\|y) \simeq_{\{z\}} z \stackrel{\$}{\leftarrow} \{0,1\}^k$$

We show the usefulness of rule [Perm] by proving the first of these two equivalences, known as *optimistic sampling*. Define

$$\begin{array}{lcl} \Psi & \stackrel{\text{def}}{=} & z\langle 1\rangle = z\langle 2\rangle \\ \Phi & \stackrel{\text{def}}{=} & x\langle 1\rangle = x\langle 2\rangle \wedge y\langle 1\rangle = y\langle 2\rangle \wedge z\langle 1\rangle = z\langle 2\rangle \\ \Theta & \stackrel{\text{def}}{=} & m_1\{x\langle 1\rangle \oplus z\langle 1\rangle / y\}\ \Phi\ m_2\{y\langle 2\rangle \oplus z\langle 2\rangle / x\} \\ & = & x\langle 1\rangle = y\langle 2\rangle \oplus z\langle 2\rangle \wedge z\langle 1\rangle = z\langle 2\rangle \end{array}$$

By rule [Assn] we have

$$\models y \leftarrow x \oplus z \sim x \leftarrow y \oplus z : \Theta \Rightarrow \Phi \tag{17}$$

We apply rule [Perm] to prove

$$\models x \stackrel{\$}{\leftarrow} \{0,1\}^k \sim y \stackrel{\$}{\leftarrow} \{0,1\}^k : \Psi \Rightarrow \Theta \tag{18}$$

For doing so we must show that for any pair of memories m_1, m_2 that coincide on z there exists a permutation f on $\{0,1\}^k$ such that

$$\forall v \in \{0,1\}^k.\ v = f(v) \oplus m_2(z) \wedge m_1(z) = m_2(z)$$

Take $f(v) \stackrel{\text{def}}{=} v \oplus m_2(z)$ to be such a permutation. Conclude from (17) and (18) by a final application of rule [Seq].

6.3 Interprocedural Code Motion

Game-based proofs commonly include bridging steps consisting in a semantics-preserving reordering of instructions. When the reordering is intraprocedural, the tactic `swap` presented in the previous section generally suffices to justify the transformation. However, proofs in the random oracle model (see §2.2 for an example of a random oracle) often include transformations where random values used inside oracles are sampled beforehand, or conversely, where sampling a random value at some point in a game is deferred to a later point, possibly in a different procedure. The former type of transformation, called eager sampling, is useful for moving random choices upfront: a systematic application of eager sampling transforms a probabilistic game G that samples a fixed number of values into a semantically equivalent game $S; G'$, where S samples the values that might be needed in G, and G' is a completely deterministic program to the exception of adversaries that may still make their own random choices.[5] The dual transformation, called lazy sampling, can be used to postpone sampling random values until they are actually used for the first time—thus, one readily knows the exact distribution of these values by reasoning locally, without the need to maintain and reason about probabilistic invariants. In this section, we present a general method to prove the correctness of interprocedural code motion. The method is based on a logic for swapping statements that generalizes our earlier lemma reported in [Barthe et al. 2009b].

A logic for swapping statements. The primary tool for performing eager/lazy sampling is an extension of the relational Hoare logic with rules for swapping statements. As the goal is to move code across procedures, it is essential that the logic considers two potentially different environments E and E'. The logic deals with judgments of the form

$$\models E, (c; S) \sim E', (S; c') : \Psi \Rightarrow \Phi$$

In most cases, the logic will be applied with S being a sequence of (guarded) sampling statements; however, we do not constrain S and merely require that it satisfies three basic properties for some sets of variables X and I:

$$\mathsf{mod}(E, S, X) \qquad \mathsf{mod}(E', S, X) \qquad \models E, S \simeq_X^{I \cup X} E', S$$

Some rules of the logic are given in Fig. 7; for the sake of readability all rules are specialized to $\equiv$, although we formalized more general versions of the rules, e.g. for

[5] Making adversaries deterministic is the goal of the *coin fixing* technique, as described by Bellare and Rogaway [2006].

$$\frac{x \notin I \cup X \qquad \mathsf{fv}(e) \cap X = \emptyset}{\models E, (x \leftarrow e; S) \equiv E', (S; x \leftarrow e)}\text{[S-Assn]} \qquad \frac{x \notin I \cup X \qquad \mathsf{fv}(d) \cap X = \emptyset}{\models E, (x \xleftarrow{\$} d; S) \equiv E', (S; x \xleftarrow{\$} d)}\text{[S-Rnd]}$$

$$\frac{\models E, (c_1; S) \equiv E', (S; c_1') \quad \models E, (c_2; S) \equiv E', (S; c_2')}{\models E, (c_1; c_2; S) \equiv E', (S; c_1'; c_2')}\text{[S-Seq]}$$

$$\frac{\models E, (c_1; S) \equiv E', (S; c_1') \qquad \models E, (c_2; S) \equiv E', (S; c_2') \qquad \mathsf{fv}(e) \cap X = \emptyset}{\models E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \equiv E', (S; \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2')}\text{[S-Cond]}$$

$$\frac{\models E, (c; S) \equiv E', (S; c') \quad \mathsf{fv}(e) \cap X = \emptyset}{\models E, (\mathsf{while}\ e\ \mathsf{do}\ c; S) \equiv E', (S; \mathsf{while}\ e\ \mathsf{do}\ c')}\text{[S-While]}$$

$$\frac{\begin{array}{c}\models E, (f.\mathsf{body}; S) \equiv E', (S; f.\mathsf{body}) \quad E(f).\mathsf{args} = E'(f).\mathsf{args} \quad E(f).\mathsf{re} = E'(f).\mathsf{re} \\ \mathsf{fv}(E(f).\mathsf{re}) \cap X = \emptyset \quad x \notin I \cup X \quad \mathsf{fv}(\vec{e}) \cap X = \emptyset\end{array}}{\models E, (x \leftarrow f(\vec{e}); S) \equiv E', (S; x \leftarrow f(\vec{e}))}\text{[S-Call]}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{wf}} \mathcal{A} \quad X \cap (\mathcal{RW} \cup \mathcal{R}) = \emptyset \quad I \cap \mathcal{RW} = \emptyset \quad \forall f \notin \mathcal{O}.\ E(f) = E'(f) \\ \forall f \in \mathcal{O}. E(f).\mathsf{args} = E'(f).\mathsf{args} \wedge E(f).\mathsf{re} = E'(f).\mathsf{re} \wedge \models E, (f.\mathsf{body}; S) \equiv E', (S; f.\mathsf{body})\end{array}}{\models E, (x \leftarrow \mathcal{A}(\vec{e}); S) \equiv E', (S; x \leftarrow \mathcal{A}(\vec{e}))}\text{[S-Adv]}$$

Fig. 7. Selected rules of a logic for swapping statements.

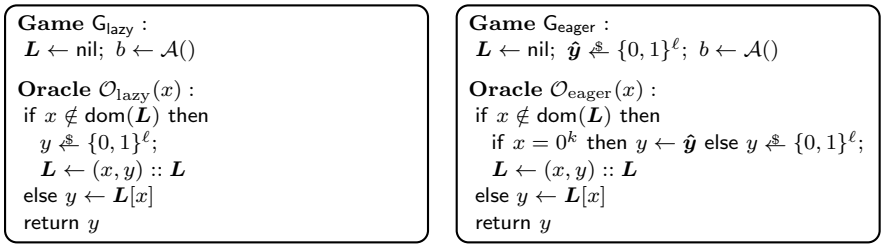

Fig. 8. An example of eager sampling justified by interprocedural code motion.

conditional statements,

$$\frac{\begin{array}{c}\models E, (c_1; S) \sim E', (S; c_1') : P \wedge e\langle 1\rangle \Rightarrow Q \quad P \Longrightarrow e\langle 1\rangle = e'\langle 2\rangle \\ \models E, (c_2; S) \sim E', (S; c_2') : P \wedge \neg e\langle 1\rangle \Rightarrow Q \quad \mathsf{fv}(e') \cap X = \emptyset\end{array}}{\models E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \sim E', (S; \mathsf{if}\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2') : P \Rightarrow Q}\ \text{[S-Cond]}$$

An application. Consider the games $\mathsf{G}_{\mathsf{lazy}}$ and $\mathsf{G}_{\mathsf{eager}}$ in Fig. 8. Both games define an oracle $\mathcal{O} : \{0,1\}^k \to \{0,1\}^\ell$. While in game $\mathsf{G}_{\mathrm{lazy}}$ the oracle is implemented as a typical random oracle that chooses its answers on demand, in $\mathsf{G}_{\mathrm{eager}}$ we use a fresh variable $\hat{\boldsymbol{y}}$ to fix in advance the response to a query of the form 0^k. We can prove that both games are perfectly indistinguishable from the point of view of an adversary $\mathcal{A}$ (who cannot write $\boldsymbol{L}$). Define

$$c \stackrel{\text{def}}{=} b \leftarrow \mathcal{A}() \qquad S \stackrel{\text{def}}{=} \mathsf{if}\ 0^k \notin \mathsf{dom}(\boldsymbol{L})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} \{0,1\}^\ell\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{L}[0^k]$$

and take $I = \{\boldsymbol{L}\}$, $X = \{\hat{\boldsymbol{y}}\}$. We introduce an intermediate game using rule [Trans],

$$\frac{\models \mathsf{G}_{\mathsf{lazy}} \simeq^{\mathcal{V}}_{\{b\}} E_{\mathrm{lazy}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; c; S) \quad \models E_{\mathrm{lazy}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; c; S) \simeq^{\mathcal{V}}_{\{b\}} \mathsf{G}_{\mathsf{eager}}}{\models \mathsf{G}_{\mathsf{lazy}} \simeq^{\mathcal{V}}_{\{b\}} \mathsf{G}_{\mathsf{eager}}}\ [\text{Trans}]$$

We prove the premise on the left by eliminating S as dead code, since it does not modify variable b. To prove the other premise, we introduce an intermediate game $(E_{\mathrm{eager}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; S; c))$. Its equivalence with $\mathsf{G}_{\mathrm{eager}}$ is direct by propagating the initial assignment to $\boldsymbol{L}$ to the condition in S and then simplifying the conditional to its first branch. Its equivalence to $(E_{\mathrm{lazy}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; c; S))$ is justified by appealing to rule [S-Adv],

$$\frac{\dfrac{}{\models \boldsymbol{L} \leftarrow \mathsf{nil} \equiv \boldsymbol{L} \leftarrow \mathsf{nil}}[\text{Refl}] \quad \dfrac{\models E_{\mathsf{lazy}}, (\mathcal{O}_{\mathsf{lazy}}; S) \equiv E_{\mathsf{eager}}, (S; \mathcal{O}_{\mathsf{eager}})}{\models E_{\mathsf{lazy}}, (c; S) \equiv E_{\mathsf{eager}}, (S; c)}[\text{S-Adv}]}{\models E_{\mathrm{lazy}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; c; S) \simeq^{\mathcal{V}}_{\{b\}} E_{\mathrm{eager}}, (\boldsymbol{L} \leftarrow \mathsf{nil}; S; c)}\ [\text{Seq}]$$

We are thus left to show

$$\models E_{\mathsf{lazy}}, (\mathcal{O}_{\mathsf{lazy}}.\mathsf{body}; S) \equiv E_{\mathsf{eager}}, (S; \mathcal{O}_{\mathsf{eager}}.\mathsf{body})$$

The proof of this latter judgment starts by an application of the generalized rule for conditionals of the logic for swapping statements. Let

$$\begin{array}{rcl} e & = & e' = x \notin \mathsf{dom}(\boldsymbol{L}) \\ c_1 & = & y \xleftarrow{\$} \{0,1\}^{\ell};\ \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L} \\ c_2 & = & c_2' = y \leftarrow \boldsymbol{L}[x] \\ c_1' & = & (\mathsf{if}\ x = 0^k\ \mathsf{then}\ y \leftarrow \hat{\boldsymbol{y}}\ \mathsf{else}\ y \xleftarrow{\$} \{0,1\}^{\ell});\ \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L} \end{array}$$

There are two non-trivial proof obligations:

(1) $\models c_2; S \sim S; c_2' : =_{\mathcal{V}} \wedge (x \in \mathsf{dom}(\boldsymbol{L}))\langle 1\rangle \Rightarrow =_{\mathcal{V}}$
This corresponds to showing that the code in the *else* branch in the conditional of each implementation of $\mathcal{O}$ commutes with S, and follows from [S-Assn];

(2) $\models c_1; S \sim S; c_1' : =_{\mathcal{V}} \wedge (x \notin \mathsf{dom}(\boldsymbol{L}))\langle 1\rangle \Rightarrow =_{\mathcal{V}}$
By case analysis on $x = 0^k$:
   (a) If $x = 0^k$, we can invoke certified program transformations—using the pre-condition that $x \notin \mathsf{dom}(\boldsymbol{L})$—to simplify the goal to the following easily provable form:

$$\models y \xleftarrow{\$} \{0,1\}^{\ell}; \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}; \hat{\boldsymbol{y}} \leftarrow y \equiv \hat{\boldsymbol{y}} \xleftarrow{\$} \{0,1\}^{\ell}; y \leftarrow \hat{\boldsymbol{y}}; \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$$

   (b) Otherwise, we do a further case analysis on $0^k \in \mathsf{dom}(\boldsymbol{L})$
      i. If $0^k \in \mathsf{dom}(\boldsymbol{L})$, we have to prove that $\models c_1; \hat{\boldsymbol{y}} \leftarrow \boldsymbol{L}[0^k] \equiv \hat{\boldsymbol{y}} \leftarrow \boldsymbol{L}[0^k]; c_1$ which is trivial;
      ii. Otherwise, the goal simplifies to $\models c_1; \hat{\boldsymbol{y}} \xleftarrow{\$} \{0,1\}^{\ell} \equiv \hat{\boldsymbol{y}} \xleftarrow{\$} \{0,1\}^{\ell}; c_1$ which is also trivial. □

7. PROOF METHODS FOR FAILURE EVENTS

One common technique to justify a *lossy* transformation $G, A \to G', A$, where $\Pr[G : A] \neq \Pr[G' : A]$ is based on what cryptographers call *failure events*. This technique relies on a *fundamental lemma* that allows to bound the difference in

the probability of an event in two games: one identifies a failure event and argues that both games behave identically until failure occurs. One can then bound the difference in probability of another event by the probability of failure in either game. Consider for example the following two program snippets and their instrumented versions:

$$\begin{array}{lcl@{\qquad}lcl} s & \stackrel{\text{def}}{=} & \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \ \mathsf{else}\ c_2 & s_{\mathbf{bad}} & \stackrel{\text{def}}{=} & \mathsf{if}\ e\ \mathsf{then}\ c_1; \mathbf{bad} \leftarrow \mathsf{true}; c\ \ \mathsf{else}\ c_2 \\ s' & \stackrel{\text{def}}{=} & \mathsf{if}\ e\ \mathsf{then}\ c_1; c'\ \mathsf{else}\ c_2 & s'_{\mathbf{bad}} & \stackrel{\text{def}}{=} & \mathsf{if}\ e\ \mathsf{then}\ c_1; \mathbf{bad} \leftarrow \mathsf{true}; c'\ \mathsf{else}\ c_2 \end{array}$$

If we ignore variable **bad**, s and $s_{\mathbf{bad}}$, and s' and $s'_{\mathbf{bad}}$, respectively, are observationally equivalent. Moreover, $s_{\mathbf{bad}}$ and $s'_{\mathbf{bad}}$ behave identically unless **bad** is set. Thus, the difference of the probability of an event A in a game G containing the program fragment s and a game G' containing instead s' can be bounded by the probability of **bad** being set in either G or G'.

LEMMA 7.1 FUNDAMENTAL LEMMA. *Let G_1, G_2 be two games and let A, B, and F be events. If $\Pr[G_1 : A \wedge \neg F] = \Pr[G_2 : B \wedge \neg F]$, then*

$$|\Pr[G_1 : A] - \Pr[G_2 : B]| \leq \max(\Pr[G_1 : F], \Pr[G_2 : F])$$

To apply this lemma, we developed a syntactic criterion to discharge its hypothesis for the case where $A = B$ and $F = \mathbf{bad}$. The hypothesis can be automatically established by inspecting the code of both games: it holds if their code differs only after program points setting the flag **bad** to true and **bad** is never reset to false afterwards. Note also that if both games terminate with probability 1, then $\Pr[G_1 : \mathbf{bad}] = \Pr[G_2 : \mathbf{bad}]$, and that if, for instance, only game G_2 terminates with probability 1, it must be the case that $\Pr[G_1 : \mathbf{bad}] \leq \Pr[G_2 : \mathbf{bad}]$.

7.1 A Logic for Bounding the Probability of Events

Many steps in game-based proofs require to provide an upper bound for the measure of some function g after the execution of a command c (throughout this section, we assume a fixed environment E that we omit from the presentation). This is typically the case when applying the Fundamental Lemma presented in the previous section: we need to bound the probability of the failure event **bad** (equivalently, the expected value of its characteristic function $\mathbb{1}_{\mathbf{bad}}$). A function f is an upper bound of $(\lambda m.\ [\![c]\!]\ m\ g)$ when

$$\vDash [\![c]\!] g \preceq f \quad \stackrel{\text{def}}{=} \quad \forall m.\ [\![c]\!]\ m\ g \leq f\ m$$

Figure 9 gathers some rules for proving the validity of such triples. The rule for adversary calls assumes that f depends only on variables that the adversary cannot modify directly (but may modify indirectly through oracle calls, of course). The correctness of this rule is proved using the induction principle for well-formed adversaries together with the rest of the rules of the logic. The rules bear some similarity with the rules of (standard) Hoare logic. However, there are some subtle differences. For example, the premises of the rules for branching statements do not consider guards. The rule

$$\frac{\vdash [\![c_1]\!] g \preceq f_{|e} \quad \vdash [\![c_2]\!] g \preceq f_{|\neg e}}{\vdash [\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f}$$

$$\vdash [\![\mathsf{skip}]\!] f \preceq f \qquad \frac{f = \lambda m.\ g(m\,\{[\![e]\!]\ m/x\})}{\vdash [\![x \leftarrow e]\!] g \preceq f} \qquad \frac{f = \lambda m.\ [\![d]\!]\ m\ (\lambda v.\ g(m\,\{v/x\}))}{\vdash [\![x \xleftarrow{\$} d]\!] g \preceq f}$$

$$\frac{\vdash [\![c_1]\!] g \preceq f \quad [\![c_2]\!] h \preceq g}{\vdash [\![c_1; c_2]\!] h \preceq f} \qquad \frac{\vdash [\![c_1]\!] g \preceq f \quad [\![c_2]\!] g \preceq f}{\vdash [\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f} \qquad \frac{\vdash [\![c]\!] f \preceq f}{\vdash [\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] f \preceq f}$$

$$\frac{\vdash g \leq g' \quad [\![c]\!] g' \preceq f' \quad f' \leq f}{\vdash [\![c]\!] g \preceq f} \qquad \frac{\vdash [\![p.\mathsf{body}]\!] g \preceq f \quad f =_X f \quad g =_Y g \quad x \notin (X \cup Y)}{\vdash [\![x \leftarrow p(\vec{e})]\!] g \preceq f}$$

$$\frac{\vdash_{\mathrm{wf}} \mathcal{A} \quad \forall p \in \mathcal{O}.\ \vdash [\![p.\mathsf{body}]\!] f \preceq f \quad f =_X f \quad X \cap (\{x\} \cup \mathcal{RW}) = \emptyset}{\vdash [\![x \leftarrow \mathcal{A}(\vec{e})]\!] f \preceq f}$$

$$\frac{f =_I f \quad \models c \simeq^I_O c' \quad g =_O g \quad \vdash [\![c']\!] g \preceq f}{\vdash [\![c]\!] g \preceq f}$$

Fig. 9. Selected rules of a logic for bounding the probability of events.

where $f_{|e}$ is defined as $(\lambda m.$ if $[\![e]\!]m$ then $f(m)$ else $0)$ can be derived from the rule for conditionals in the figure by two simple applications of the "rule of consequence". Moreover, the rule for conditional statements (and its variant above) is incomplete: consider a statement of the form $[\![\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ such that $[\![c_1]\!] g \preceq f$ is valid, but not $[\![c_2]\!] g \preceq f$; the triple $[\![\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ is valid, but to derive it one needs to resort to observational equivalence. More general rules exist, but we have not formalized them since we did not need them in our proofs.[6]

Discussion. The differences between the above triples and those of Hoare logic are inherent to their definition, which is tailored to establish upper bounds for the probability of events. Nevertheless, the validity of a Hoare triple $\{P\}\ c\ \{Q\}$ (in which pre- and post-conditions are Boolean-valued predicates) is equivalent to the validity of the triple $[\![c]\!]\mathbb{1}_{\neg Q} \preceq \mathbb{1}_{\neg P}$. We can consider dual triples of the form $[\![c]\!] g \succeq f$ whose validity is defined as:

$$\models [\![c]\!] g \succeq f \quad \stackrel{\text{def}}{=} \quad \forall m.\ [\![c]\!]\ m\ g \geq f\ m$$

This allows to express termination of a program as $[\![c]\!]\mathbb{1} \succeq \mathbb{1}$ and admits an embedding of Hoare triples, mapping $\{P\}\ c\ \{Q\}$ to $[\![c]\!]\mathbb{1}_Q \succeq \mathbb{1}_P$. However, this embedding does not preserve validity for non-terminating programs under the partial correctness interpretation. Consider a program c that never terminates: we have $\{\mathsf{true}\}\ c\ \{\mathsf{false}\}$, but clearly not $[\![c]\!]\mathbb{1}_{\mathsf{false}} \succeq \mathbb{1}$.

7.2 Automation

In most applications of Lemma 7.1, failure can only be triggered by oracle calls. Typically, the flag **bad** that signals failure is set in the code of an oracle for which an upper bound for the number of queries made by the adversary is known. The following lemma provides a general method for bounding the probability of failure under such circumstances.

[6]More generally, it seems possible to make the logic complete, at the cost of considering more complex statements with pre-conditions on memories.

LEMMA 7.2 FAILURE EVENT LEMMA. *Consider an event F and a game G that gives adversaries access to an oracle $\mathcal{O}$. Let* $\mathsf{cntr} : \mathcal{E}_{\mathbb{N}}$, $h : \mathbb{N} \to [0,1]$ *be such that* cntr *and F do not depend on variables that can be written outside $\mathcal{O}$, and for any initial memory m,*

$$\neg F(m) \Longrightarrow \Pr\left[\mathcal{O}.\mathsf{body}, m : F\right] \leq h(\llbracket \mathsf{cntr} \rrbracket\ m)$$

and

$$\begin{array}{l} \mathsf{range}\ (\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m)\ (\lambda m'.\ \llbracket \mathsf{cntr} \rrbracket\ m < \llbracket \mathsf{cntr} \rrbracket\ m') \vee \\ \mathsf{range}\ (\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m)\ (\lambda m'.\ \llbracket \mathsf{cntr} \rrbracket\ m = \llbracket \mathsf{cntr} \rrbracket\ m' \wedge F\ m' = F\ m) \end{array}$$

Then, for any initial memory m satisfying $\neg F(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m = 0$,

$$\Pr\left[G, m : F \wedge \mathsf{cntr} \leq q\right] \leq \sum_{i=0}^{q-1} h(i)$$

PROOF. Define $f : \mathcal{M} \to [0,1]$ as follows

$$f(m) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \llbracket \mathsf{cntr} \rrbracket\ m > q \\ \mathbb{1}_F(m) + \mathbb{1}_{\neg F}(m) \sum\limits_{i=\llbracket \mathsf{cntr} \rrbracket m}^{q-1} h(i) & \mathsf{if}\ \llbracket \mathsf{cntr} \rrbracket\ m \leq q \end{cases}$$

We show $\llbracket G \rrbracket f \preceq f$ by structural induction on the code of G using the rules of the logic presented in the previous section. We first prove that $\mathcal{O}$ satisfies the triple $\llbracket \mathcal{O}.\mathsf{body} \rrbracket f \preceq f$. We must show that for every m, $\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f \leq f(m)$. This is trivial when cntr is not incremented, because we have

$$\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f = f(m)\ (\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ \mathbb{1}) \leq f(m)$$

When cntr is incremented and $\llbracket \mathsf{cntr} \rrbracket\ m \geq q$, this is trivial too, because the left hand side becomes 0. We are left with the case where $\mathcal{O}.\mathsf{body}$ increments cntr and $\llbracket \mathsf{cntr} \rrbracket\ m < q$. If $F(m)$, the right hand side is equal to 1 and the inequality holds. Otherwise, we have from the hypotheses that

$$\begin{aligned} \llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f\ &\leq\ \llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ \left(\lambda m'.\mathbb{1}_F(m') + \mathbb{1}_{\neg F}(m') \sum_{i=\llbracket \mathsf{cntr} \rrbracket m'}^{q-1} h(i)\right) \\ &\leq\ \Pr\left[\mathcal{O}.\mathsf{body}, m : F\right] + \Pr\left[\mathcal{O}.\mathsf{body}, m : \neg F\right] \sum_{i=\llbracket \mathsf{cntr} \rrbracket m+1}^{q-1} h(i) \\ &\leq\ h(\llbracket \mathsf{cntr} \rrbracket\ m) + \sum_{i=\llbracket \mathsf{cntr} \rrbracket m+1}^{q-1} h(i)\ =\ \sum_{i=\llbracket \mathsf{cntr} \rrbracket m}^{q-1} h(i)\ =\ f(m) \end{aligned}$$

Using the rules in Fig. 9, we can then extend this result to adversary calls and to the rest of the game, showing that $\llbracket G \rrbracket f \preceq f$.

Finally, let m be a memory such that $\neg F(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m = 0$. It follows immediately from $\llbracket G \rrbracket f \preceq f$ that

$$\Pr\left[G, m : F \wedge \mathsf{cntr} \leq q\right]\ \leq\ \llbracket G \rrbracket\ m\ f\ \leq\ f(m)\ =\ \sum_{i=0}^{q-1} h(i) \quad \square$$

When failure is defined as the probability of a flag **bad** being set by an oracle and the number of queries the adversary makes to this oracle is upper bounded by q, the above lemma can be used to bound the probability of failure by taking $F = \mathbf{bad}$ and defining h suitably. In most practical applications the probability of an oracle call raising failure is history-independent and hence h is a constant function. The proof of Lemma 8.6 given in Section 8.3.2 is an exception for which the full generality of the lemma is needed.

8. CASE STUDIES

In this Section we overview four significant case studies that have been verified in CertiCrypt: the existential unforgeability of Full-Domain Hash signatures under adaptive chosen-message attacks [Zanella Béguelin et al. 2009], the indistinguishability under adaptive chosen-ciphertext attacks of OAEP ciphertexts, two compact proofs of the PRP/PRF switching lemma [Barthe et al. 2010a], and a formalization of a large class of zero-knowledge protocols [Barthe et al. 2010b].

8.1 Existential Unforgeability of FDH

Full-Domain Hash (FDH) was proposed by Bellare and Rogaway [1996] as an efficient RSA-based signature scheme, but is in fact an instance of an earlier construction described by the same authors in [1993]. We consider this latter, more general construction, which is based on a family of trapdoor permutations—RSA being just one possible choice.

Definition 8.1 *Trapdoor permutation.* A family of trapdoor permutations is a triple of algorithms $(\mathcal{KG}, f, f^{-1})$. For a given value of the security parameter η, the key generator $\mathcal{KG}(\eta)$ randomly selects a pair of keys (pk, sk) such that $f(pk, \cdot)$ is a permutation on its domain and $f^{-1}(sk, \cdot)$ is its inverse.

Definition 8.2 *FDH signature scheme.* Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor permutations on cyclic groups G_η and let H be a family of hash functions from bitstrings of arbitrary length onto the domain of the permutation. The FDH signature scheme is composed of the following triple of algorithms:

$$\begin{array}{lcl} \mathcal{KG}(\eta) & \stackrel{\text{def}}{=} & (pk, sk) \leftarrow \mathcal{KG}_f(\eta);\ \mathsf{return}\ (pk, sk) \\ \mathsf{Sign}(sk, m) & \stackrel{\text{def}}{=} & \mathsf{return}\ f^{-1}(sk, H(m)) \\ \mathsf{Verify}(pk, m, \sigma) & \stackrel{\text{def}}{=} & \mathsf{return}\ (f(pk, \sigma) = H(m)) \end{array}$$

The signature of a message $m \in \{0,1\}^*$ is simply $f^{-1}(sk, H(m))$, the preimage of its digest under f. To verify a purported signature σ on a message m, it suffices to check whether $H(m)$ and $f(pk, \sigma)$ coincide.

We prove in the random oracle model that if the underlying trapdoor permutation is homomorphic and one-way (i.e. difficult to invert), then FDH is secure against existential forgery under adaptive chosen-message attacks. This means that modeling H as a truly random function, an efficient adversary that can ask for signatures of messages of its choice, can only succeed with negligible probability in forging a signature for a fresh message. We give a game-based proof of an exact security bound that relates the problem of forging a signature to the problem of inverting the trapdoor permutation. The initial and final games encoding both problems appear in Figure 10; the proof in CertiCrypt is about 3,500 lines long.

THEOREM 8.3 EXISTENTIAL UNFORGEABILITY OF FDH. *Assume the underlying trapdoor permutation $(\mathcal{KG}_f, f, f^{-1})$ is homomorphic with respect to the group operation in its domain, i.e. for every (pk, sk) that might be output by $\mathcal{KG}_f$, and every x, y, $f(pk, x \times y) = f(pk, x) \times f(pk, y)$. Let $\mathcal{A}$ be an adversary against the existential unforgeability of FDH that makes at most q_H and q_S queries to the hash and signing oracles respectively. Suppose $\mathcal{A}$ succeeds in forging a signature for a fresh message within time t with probability ϵ during experiment G_{EF}. Then, there exists an inverter $\mathcal{B}$ that finds the preimage of an element uniformly drawn from the range of f with probability ϵ' within time t' during experiment G_{OW}, where*

$$t' \leq t + (q_H + q_S + 1)\ O(t_f) \qquad \epsilon' \geq p\ (1-p)^{q_S}\ \epsilon$$

and t_f is an upper bound for the time needed to compute the image of a group element under f.

The inverter first selects $q_H + q_S + 1$ bits at random, choosing true with probability p and false with probability $(1-p)$, and stores them in a list $\boldsymbol{T}$. It then runs $\mathcal{A}$ simulating the hash and signing oracles. It answers to the 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 i-th entry in $\boldsymbol{T}$: if it is true, answers with $y \times f(pk, r)$ where y is its challenge, if it is false answers with simply $f(pk, r)$. In both cases the answers are indistinguishable from those of a random oracle. When $\mathcal{A}$ asks for the signature of a message m, the inverter makes the corresponding hash query itself and then answers with $\boldsymbol{P}[m]$. 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 respective entries in $\boldsymbol{P}$ coincide with the preimage of their hash value; this happens with probability $(1-p)^{q_S}$. When $\mathcal{A}$ halts returning a pair (m, σ), $\mathcal{B}$ returns $\sigma \times \boldsymbol{P}[m]^{-1}$. For a valid forgery, this value is a preimage of y when the entry corresponding to m in $\boldsymbol{T}$ is true; this happens with probability p. Thus, we have

$$\Pr\left[\mathsf{G}_{\mathsf{OW}} : x = f^{-1}(sk, y)\right] \geq p\ (1-p)^{q_S}\ \Pr\left[\mathsf{G}_{\mathsf{EF}} : h = f(pk, \sigma)\right]$$

The aim of the inverter is to inject its challenge y in as many hash queries as possible, while at the same time maximizing the probability of the simulation being correct. The value $p = (q_S + 1)^{-1}$ maximizes the factor $p\ (1-p)^{q_S}$ and thus maximizes the success probability of the reduction. For this value of p, the factor approximates $\exp(-1)\ q_S^{-1}$ for large values of q_S. This reduction is optimal [Coron 2002] and its success probability 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.

8.2 Ciphertext Indistinguishability of OAEP under Chosen-Ciphertext Attacks

Optimal Asymmetric Encryption Padding (OAEP) [Bellare and Rogaway 1994] is a prominent public-key encryption scheme based on trapdoor permutations, most commonly used in combination with the RSA and Rabin functions. OAEP is widely deployed; many variants of OAEP are recommended by several standards, including IEEE P1363, PKCS, ISO 18033-2, ANSI X9, CRYPTREC and SET. Yet, the history of OAEP security is fraught with difficulties. The original paper of Bellare and

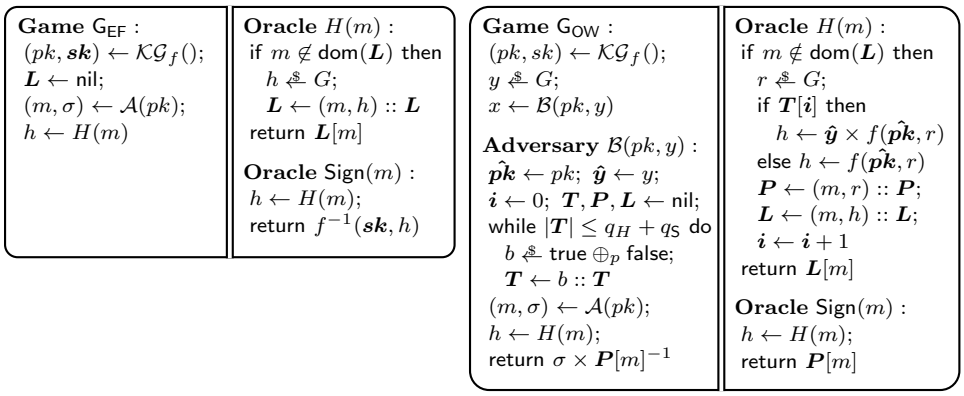

Fig. 10. Initial and final games in the proof of security of FDH. We use ($\mathsf{true} \oplus_p \mathsf{false}$) to denote a Bernoulli distribution with success probability p, i.e. the discrete distribution that takes value true with probability p and false with probability $(1 - p)$.

Rogaway [1994] proves that, under the hypothesis that the underlying trapdoor permutation family is one-way, OAEP is semantically secure under chosen-ciphertext attacks. Shoup [2001] discovered later that this proof only established the security of OAEP against non-adaptive chosen-ciphertext attacks (IND-CCA), and not, as was believed at that time, against the stronger version of ciphertext indistinguishability that allows the adversary to adaptively obtain the decryption of ciphertexts of her choice (IND-CCA). In response, Shoup suggested a modified scheme, secure against adaptive attacks under the one-wayness of the underlying permutation, and gave a proof of the adaptive security of the original scheme when it is used in combination with RSA with public exponent $e = 3$. Simultaneously, Fujisaki et al. [2004] proved that OAEP in its original formulation is indeed secure against adaptive attacks, but under the assumption that the underlying permutation family is *partial-domain* one-way. Since for the particular case of RSA this latter assumption is no stronger than (full-domain) one-wayness, this finally established the adaptive IND-CCA security of RSA-OAEP. Unfortunately, when one takes into account the additional cost of reducing the problem of inverting RSA to the problem of partially-inverting it, the security bound becomes less attractive. We note that there exist variants of OAEP that admit more efficient reductions when used in combination with the RSA and Rabin functions, notably SAEP, SAEP+ [Boneh 2001], and alternative schemes with tighter generic reductions, e.g. REACT [Okamoto and Pointcheval 2001b].

Here we report on a machine-checked proof of the IND-CCA security of generic OAEP in the random oracle model. We use as a starting point the proof of Pointcheval [2005], which fills several gaps in the reduction of Fujisaki et al. [2004], resulting in a weaker security bound. Our proof unveils minor glitches in the proof of Pointcheval [2005] and marginally improves on its exact security bound (reducing the coefficients) by performing an aggressive analysis of oracle queries earlier in the sequence of games. The initial and final games of the reduction appear in Figure 11; the proof in CertiCrypt is about 10,000 lines long.

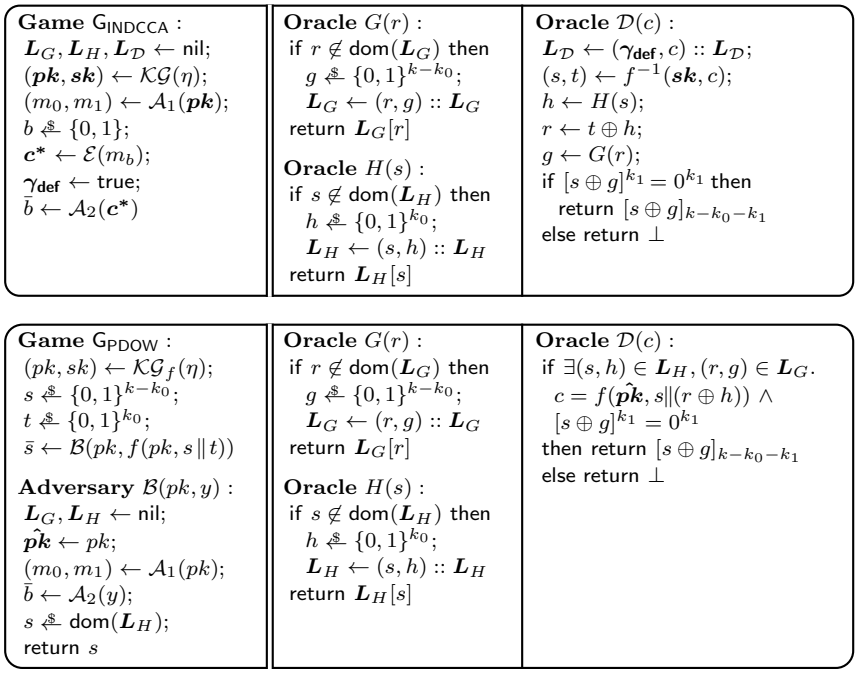

Fig. 11. Initial and final games in the reduction of the IND-CCA security of OAEP to the problem of partially inverting the underlying permutation. We exclude cheating adversaries who query the decryption oracle with the challenge ciphertext during the second phase of the experiment by requiring $(\mathsf{true}, \boldsymbol{c}^*) \notin \boldsymbol{L}_{\mathcal{D}}$ to be a post-condition of the initial game.

Definition 8.4 *OAEP encryption scheme.* Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor permutations on $\{0,1\}^k$, and let

$$G : \{0,1\}^{k_0} \to \{0,1\}^{k-k_0} \qquad H : \{0,1\}^{k-k_0} \to \{0,1\}^{k_0}$$

be two hash functions, with $k > k_0 + k_1$. The OAEP scheme is composed of the following triple of algorithms:

$$\begin{array}{lcl}
\mathcal{KG}(\eta) & \stackrel{\text{def}}{=} & (pk, sk) \leftarrow \mathcal{KG}_f(\eta);\ \mathsf{return}\ (pk, sk) \\
\mathcal{E}(pk, m) & \stackrel{\text{def}}{=} & r \xleftarrow{\$} \{0,1\}^{k_0};\ s \leftarrow G(r) \oplus (m \| 0^{k_1});\ t \leftarrow H(s) \oplus r;\ \mathsf{return}\ f(pk, s \| t) \\
\mathcal{D}(sk, c) & \stackrel{\text{def}}{=} & (s \,\|\, t) \leftarrow f^{-1}(sk, c);\ r \leftarrow t \oplus H(s);\ m \leftarrow s \oplus G(r); \\
 & & \mathsf{if}\ [m]_{k_1} = 0^{k_1}\ \mathsf{then\ return}\ [m]^{k-k_0-k_1}\ \mathsf{else\ return}\ \perp
\end{array}$$

where $[x]_n$ (resp. $[x]^n$) denotes the n least (resp. most) significant bits of x.

Theorem 8.5 Ciphertext Indistinguishability of OAEP. *Let $\mathcal{A}$ be an adversary against the ciphertext indistinguishability of OAEP under an adaptive chosen-ciphertext attack that makes at most q_G and q_H queries to the hash oracles G and H, respectively, and at most q_D queries to the decryption oracle $\mathcal{D}$.*[7] *Suppose*

[7] The machine-checked proof slightly relaxes this condition; it requires that the length of $\boldsymbol{L}_G$ be less than $q_D + q_G + 1$ (the 1 accounting for the call to G needed to compute the challenge ciphertext), so that the adversary could trade calls to $\mathcal{D}$ for calls to G.

$\mathcal{A}$ achieves an advantage ϵ within time t during game $\mathsf{G}_{\mathsf{INDCCA}}$. Then, there exists an inverter $\mathcal{B}$ that finds a partial preimage (the $k - k_0$ most significant bits of a preimage) of an element uniformly drawn from the range of f with probability ϵ' within time t' during experiment $\mathsf{G}_{\mathsf{PDOW}}$, where

$$t' \leq t + q_G\ q_H\ q_D\ O(t_f) \qquad \epsilon' \geq \frac{1}{q_H}\left(\epsilon - \frac{3q_Dq_G + q_D^2 + 4q_D + q_G}{2^{k_0}} - \frac{2q_D}{2^{k_1}}\right)$$

8.3 The PRP/PRF Switching Lemma

Suppose you give an adversary black-box access to either a random function or a random permutation, and you ask her to tell you which is the case. For the sake of concreteness let us assume the domain of the permutation (and the domain and range of the function) is $\{0,1\}^\ell$. No matter what strategy the adversary follows, due to the birthday problem, after roughly $2^{\ell/2}$ queries to the oracle she will be able to tell in which scenario she is with a high probability. If the oracle is a random function, a collision is almost sure to occur, whereas it could not occur when the oracle is a random permutation. The birthday problem gives a lower bound for the advantage of the adversary. The PRP/PRF Switching Lemma gives an upper bound. In a code-based setting, its formulation is given in terms of two games G_{RP} and G_{RF}, that give the adversary access to an oracle that represents a random permutation and a random function, respectively:

Game G_{RP} :
$\boldsymbol{L} \leftarrow \mathsf{nil};\ b \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$;
  $\boldsymbol{L} \leftarrow (x,y) :: \boldsymbol{L}$
return $\boldsymbol{L}[x]$

Game G_{RF} :
$\boldsymbol{L} \leftarrow \mathsf{nil};\ b \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} \{0,1\}^\ell$;
  $\boldsymbol{L} \leftarrow (x,y) :: \boldsymbol{L}$
return $\boldsymbol{L}[x]$

where the instruction $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$ samples uniformly a bitstring of length ℓ that is not in the range of the association list $\boldsymbol{L}$, thus ensuring that oracle $\mathcal{O}$ in G_{RP} implements an injective—and therefore bijective—function.

LEMMA 8.6 PRP/PRF SWITCHING LEMMA. *Suppose $\mathcal{A}$ makes at most $q > 0$ queries to oracle $\mathcal{O}$. Then,*

$$|\Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \leq \frac{q(q-1)}{2^{\ell+1}} \tag{19}$$

The standard *proof* of the PRP/PRF Switching Lemma is due to Impagliazzo and Rudich [1989, Theorem 5.1]. Bellare and Rogaway [2006] report a subtle error in the reasoning and give another game-based proof under the additional assumption that the adversary never asks the same query twice. Their proof uses the Fundamental Lemma (see §7) to bound the advantage of the adversary by the probability of a failure event, but their justification of the bound on the probability of failure remains informal. Shoup [2004, §5.1] gives another game-based proof of the lemma under the assumption that the adversary makes exactly q distinct queries. In his proof, the challenger acts as an intermediary between the oracle and the adversary: rather than the adversary calling the oracle at her discretion, it is the challenger

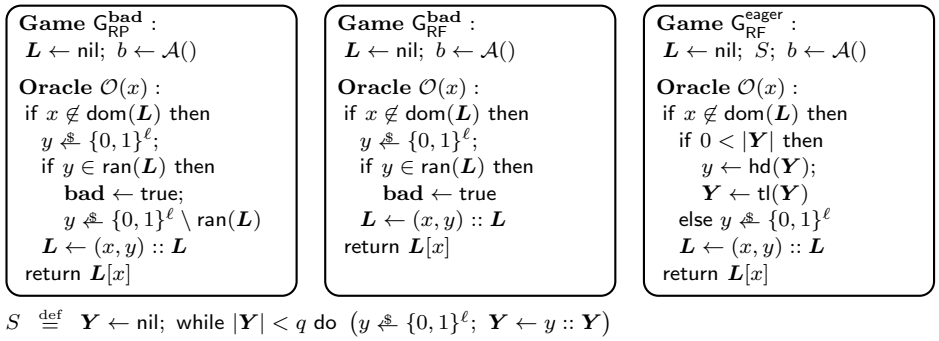

Fig. 12. Games used in the proofs of the PRP/PRF Switching Lemma.

who calls the adversary to get a query and who forwards it to the oracle. The PRP/PRF Switching Lemma has been formalized previously. Affeldt et al. [2007] present a Coq proof of a variant of the lemma that only holds for non-adaptive and deterministic adversaries; Barthe et al. [2009b] report on a formalization of the full result in CertiCrypt. Here we overview two significantly shorter proofs that better exploit the code-based techniques presented in earlier sections. Both proofs use the Fundamental Lemma to bound the advantage of the adversary by the probability of a failure event. The first proof uses the eager sampling technique of Section 6.3 to bound the probability of failure, whereas the second one relies on Lemma 7.2. We begin by introducing in Fig. 12 annotated versions $\mathsf{G}_{\mathsf{RP}}^{\mathbf{bad}}$ and $\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}}$ of the games G_{RP} and G_{RF}. From Lemma 7.1, we readily have

$$|\Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \ \leq\ \Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}} : \mathbf{bad}\right]$$

8.3.1 *A Proof Based on Eager Sampling.* We make a first remark: the probability of **bad** being set in game $\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}}$ is bounded by the probability of having a collision in $\mathsf{ran}(\boldsymbol{L})$ at the end of the game; let us write this latter event as $\mathsf{col}(\boldsymbol{L})$. We prove this by showing that $\mathbf{bad} \Longrightarrow \mathsf{col}(\boldsymbol{L})$ is an invariant of the game.

Using the logic for swapping statements, we then modify the oracle in $\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}}$ so that the responses to the first q queries are instead chosen at the beginning of the game and stored in a list $\boldsymbol{Y}$, thus obtaining the equivalent eager version $\mathsf{G}_{\mathsf{RF}}^{\mathsf{eager}}$ shown in Fig. 12. Each time a query is made, the oracle pops a value from list $\boldsymbol{Y}$ and gives it back to the adversary as the response.

Since the initialization code S terminates and does not modify $\boldsymbol{L}$, we can have that

$$\Pr[\mathsf{G}_{\mathsf{RF}} : \mathsf{col}(\boldsymbol{L})] = \Pr[\mathsf{G}_{\mathsf{RF}};\ S : \mathsf{col}(\boldsymbol{L})] = \Pr[\mathsf{G}_{\mathsf{RF}}^{\mathsf{eager}} : \mathsf{col}(\boldsymbol{L})]$$

We prove using the relational Hoare logic that having a collision in the range of $\boldsymbol{L}$ at the end of this last game is the same as having a collision in $\boldsymbol{Y}$ immediately after executing S. We conclude that the bound in (19) holds by analyzing the loop in S. Observe that if there are no collisions in $\boldsymbol{Y}$ in a memory m, we can prove by induction on $(q - |\boldsymbol{Y}|)$ that the probability of sampling a colliding value in the

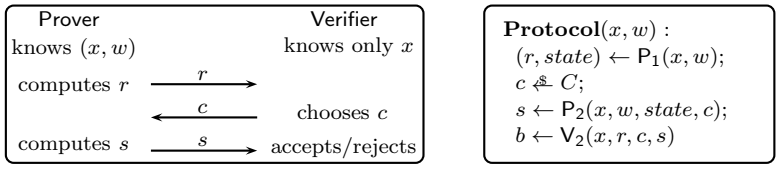

Fig. 13. Characteristic 3-step interaction in a Σ-protocol and its formalization as a game.

remaining loop iterations is

$$\Pr\left[S, m : \exists i, j \in \mathbb{N}.\ i < j < q \wedge \boldsymbol{Y}[i] = \boldsymbol{Y}[j]\right] = \sum_{i=|\boldsymbol{Y}|}^{q-1} \frac{i}{2^{\ell}}$$

8.3.2 *A Proof Based on the Failure Event Lemma.* The bound in (19) follows from a direct application of Lemma 7.2. It suffices to take $F = \mathbf{bad}$, $h(i) = i\,2^{-\ell}$, and $\mathsf{cntr} = |\boldsymbol{L}|$. If **bad** is initially set to false in memory m, we have

$$\Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}}, m : \mathbf{bad}\right] = \Pr\left[b \leftarrow \mathcal{A}(), m\,\{\mathsf{nil}/\boldsymbol{L}\} : \mathbf{bad} \wedge |\boldsymbol{L}| \leq q\right] \leq \sum_{i=0}^{q-1} h(i) = \frac{q(q-1)}{2^{\ell+1}}$$

The first equation holds because $\mathcal{A}$ does not make more than q queries to $\mathcal{O}$; the inequality is obtained from Lemma 7.2; we use the logic in Fig. 9 to bound the probability of **bad** being set in one call to the oracle by $h(\mathsf{cntr})$.

Remark. The resulting proof is considerably shorter compared to the one presented in the previous section, about 100 lines of Coq compared to 400 lines (both proofs are significantly more compact than the 900-lines proof reported in [Barthe et al. 2009b]).

8.4 Formalization of Σ-Protocols

A Σ-protocol is a 3-step interactive protocol where a prover P interacts with a verifier V. Both parties have access to a common input x, and the goal of the prover is to convince the verifier that she knows some value w suitably related to x, without revealing anything beyond this assertion. The protocol begins with the prover sending a commitment r to the verifier, who responds with a random challenge c chosen uniformly from a set C; the prover then computes a response s and sends it back to the verifier, who either accepts or rejects the conversation. Fig. 13 shows a diagram of a run of a Σ-protocol and the formalization of the interaction as a game in CertiCrypt, where the different phases of each party are represented as procedures. Formally, a Σ-protocol is defined with respect to a knowledge relation R and must satisfy the following three properties:

(1) Completeness: Given an $x \in \mathrm{dom}(R)$ the prover always convinces the verifier:

$$R(m(x), m(w)) \Longrightarrow \Pr\left[\mathbf{Protocol}(x, w), m : b = \mathsf{true}\right] = 1$$

(2) Honest-Verifier Zero-Knowledge: There exists an efficient simulator S that given $x \in \mathrm{dom}(R)$ computes triples (r, c, s) with the same distribution as a valid conversation:

$$\models \mathbf{Protocol}(x, w) \sim (r, c, s) \leftarrow \mathsf{S}(x) : =_{\{x\}} \wedge\ R(x, w)\langle 1\rangle \Rightarrow =_{\{r,c,s\}}$$

Table I. Special homomorphisms in selected Σ^ϕ-protocols. In the table, $\mathbb{Z}_q^+$ stands for the additive group of integers modulo q, $\mathbb{Z}_p^*$ for the multiplicative group of integers modulo p; N is an RSA modulus and e a public RSA exponent coprime with $\varphi(N)$.

Protocol	G	H	ϕ	u	v
Schnorr	$\mathbb{Z}_q^+$	$\mathbb{Z}_p^*$	$x \mapsto g^x$	$x \mapsto 0$	q
Okamoto	$(\mathbb{Z}_q^+, \mathbb{Z}_q^+)$	$\mathbb{Z}_p^*$	$(x_1, x_2) \mapsto g_1^{x_1} \otimes g_2^{x_2}$	$x \mapsto (0,0)$	q
Fiat-Shamir	$\mathbb{Z}_N^*$	$\mathbb{Z}_N^*$	$x \mapsto x^2$	$x \mapsto x$	2
Guillou-Quisquater	$\mathbb{Z}_N^*$	$\mathbb{Z}_N^*$	$x \mapsto x^e$	$x \mapsto x$	e
Feige-Fiat-Shamir	$\{-1,1\} \times \mathbb{Z}_N^*$	$\mathbb{Z}_N^*$	$(s,x) \mapsto s.x^2$	$x \mapsto (1,x)$	2

(3) Special soundness: Given two accepting conversations for an input x with the same commitment r but with different challenges, there exists an efficient *knowledge extractor* KE that computes a witness w such that $R(x,w)$:

$$(r,c_1,s_1),(r,c_2,s_2) \text{ accepting } \wedge\ c_1 \neq c_2 \Longrightarrow \Pr\left[w \leftarrow \mathsf{KE}(x,r,c_1,c_2,s_1,s_2) : R(x,w)\right] = 1$$

We showed in [Barthe et al. 2010b] that many protocols in the literature are instances of an abstract protocol that proves knowledge of preimages under a homomorphism.

Definition 8.7 *Special homomorphism.* We say that a homomorphism ϕ between a finite additive group $(G,\oplus)$ and a multiplicative group $(H,\otimes)$ is special if there exists a value $v \in \mathbb{Z} \setminus \{0\}$ (called *special exponent*) and an efficient algorithm that given $x \in H$ computes $u \in G$ such that $\phi(u) = x^v$.

THEOREM 8.8 Σ^ϕ-PROTOCOLS FOR SPECIAL HOMOMORPHISMS. *If ϕ is special and c^+ is smaller than any prime divisor of the special exponent v, then there exists a Σ-protocol with knowledge relation $R \stackrel{def}{=} \{(x,w) \mid x = \phi(w)\}$ and challenge set $C = [0..c^+]$.*

Table I shows several archetypal zero-knowledge protocols that can be construed as Σ^ϕ-protocols, and for which we have used the previous theorem to obtain compact proofs of their completeness, honest-verifier zero-knowledge and special soundness.

9. RELATED WORK

Cryptographic protocol verification is an established area of formal methods, and a wealth of automated and deductive methods have been developed to the purpose of verifying that protocols provide the expected level of security [Meadows 2003]. Traditionally, protocols have been verified in a symbolic model, for which effective decision procedures exist under suitable hypotheses [Abadi and Cortier 2006]. Although the symbolic model assumes perfect cryptography, soundness results such as [Abadi and Rogaway 2002]—see [Cortier et al. 2010] for a recent survey—relate the symbolic model with the computational model, provided the cryptographic primitives satisfy adequate notions of security. It is possible to combine symbolic methods and soundness proofs to achieve guarantees in the computational model, as done e.g. in [Backes and Laud 2006; Sprenger and Basin 2008; Backes et al. 2010]. One drawback of this approach is that the security proof relies on intricate soundness proofs and hypotheses that unduly restrict the usage of primitives.

Besides, it is not clear whether computational soundness results will always exist to allow factoring verification through symbolic methods [Backes and Pfitzmann 2005]. Consequently, some authors attempt to provide guarantees directly at the computational level [Blanchet 2008; Laud 2001; Roy et al. 2008].

In contrast, the formal verification of cryptographic functionalities is an emerging trend. An early work of Barthe et al. [2004] proves the security of ElGamal in Coq, but the proof relies on the generic model, a very specialized and idealized model that elides many of the issues that are relevant to cryptography. Den Hartog [2008] also proves ElGamal semantic security using a probabilistic (non-relational) Hoare logic. However, their formalism is not sufficiently powerful to express precisely security goals: notions such as well-formed and effective adversary are not modeled.

Blanchet and Pointcheval [2006] were among the first to use verification tools to carry out game-based proofs of cryptographic schemes. They used CryptoVerif to prove the existential unforgeability of the FDH signature scheme, with a bound weaker than the one given in Section 8.1. CryptoVerif has also been used to verify the security of many protocols, including Kerberos [Blanchet et al. 2008]. It is difficult to assess CryptoVerif ability to handle automatically more complex cryptographic proofs (or tighter security bounds), e.g. for schemes such as OAEP; on the other hand, compiling CryptoVerif sequences of games in CertiCrypt is an interesting research direction that would increase automation in CertiCrypt and confidence in CryptoVerif—by generating independently verifiable proofs.

Impagliazzo and Kapron [2006] were the first to develop a logic to reason about indistinguishability. Their logic is built upon a more general logic whose soundness relies on non-standard arithmetic; they show the correctness of a pseudo-random generator and that next-bit unpredictability implies pseudo-randomness. Recently, Zhang [2009] developed a similar logic on top of Hofmann's SLR system [Hofmann 1998] and reconstructed the examples of Impagliazzo and Kapron [2006]. These logics have limited applicability because they lack support for oracles or adaptive adversaries and so cannot capture many of the the standard patterns for reasoning about cryptographic schemes. More recently Barthe et al. [2010] developed a general logic, called Computational Indistinguishability Logic (CIL), that captures reasoning patterns that are common in provable security, such as simulation and reduction, and deals with oracles and adaptive adversaries. They use CIL to prove the security of the Probabilistic Signature Scheme, a widely used signature scheme that forms part of the PKCS standard [Bellare and Rogaway 1996]. CIL subsumes an earlier logic by Courant et al. [2008], who developed a form of strongest post-condition calculus that can establish automatically asymptotic security (IND-CPA and IND-CCA) of encryption schemes that use one-way functions and hash functions modeled as random oracles. They show soundness and provide a prototype implementation that covers many examples in the literature.

In parallel, several authors have initiated formalizations of game-based proofs in proof assistants and shown the security of basic examples. Nowak [2007] gives a game-based proof of ElGamal semantic security in Coq. Nowak uses a shallow embedding to model games; his framework ignores complexity issues and has limited support for proof automation: because there is no special syntax for writing games, mechanizing syntactic transformations becomes very difficult. Affeldt et al. [2007]

formalize a game-based proof of the PRP/PRF switching lemma in Coq. However, their formalization is tailored towards the particular example they consider, which substantially simplifies their task and hinders generality. They deal with a weak (non-adaptive) adversary model and ignore complexity. In another attempt to build a system supporting provable security, Backes et al. [2008] formalize a language for games in the Isabelle proof assistant and prove the Fundamental Lemma; however, no examples are reported. All in all, these works appear like preliminary experiments that are not likely to scale.

Leaving the realm of cryptography, CertiCrypt relies on diverse mathematical concepts and theories that have been modeled for their own sake. We limit ourselves to singling out Audebaud and Paulin-Mohring [2009] formalization of the measure monad, which we use extensively, and the work of Hurd et al. [2005], who developed a mechanized theory in the HOL theorem prover for reasoning about pGCL programs, a probabilistic extension of Dijkstra's guarded command language.

10. CONCLUSION

CertiCrypt is a fully formalized framework that supports machine-checked game-based proofs; we have validated its design through formalizing standard cryptographic proofs. Our work shows that machine-checked proofs of cryptographic schemes are not only plausible but indeed feasible. However, constructing machine-checked proofs requires a high-level of expertise in formal proofs and remains time consuming despite the high level of automation achieved. Thus, CertiCrypt only provides a first step towards the completion of Halevi's program, in spite of the amount of work invested so far (the project was initiated in June 2006). A medium-term objective would be to develop a minimalist interface that eases the writing of games and provides a fixed set of mechanisms (tactics, proof-by-pointing) to prove some basic transitions, leaving the side conditions as hypotheses. We believe that such an interface would help cryptographers ensure that there are no obvious flaws in their definitions and proofs, and to build sketches of security proofs. In fact, it is our experience that the type system and the automated tactics provide valuable information in debugging proofs.

Numerous research directions remain to be explored. Our main priority is to improve proof automation. In particular, we expect that one can automate many proofs in pRHL, by relying on a combination of standard verification tools: weakest pre-condition generators, invariant inference tools, SMT solvers.

In addition, it would be useful to formalize cryptographic meta-results such as the equivalence between IND-CPA and IND-CCA under plaintext awareness, or the transformation of an IND-CPA-secure scheme into an IND-CCA-secure scheme [Fujisaki and Okamoto 1999]. Another direction would be to formalize proofs of computational soundness of the symbolic model, see e.g. [Abadi and Rogaway 2002] and proofs of automated methods for proving security of primitives and protocols, see e.g. [Courant et al. 2008; Laud 2001]. Finally, it would also be worthwhile to explore applications of CertiCrypt outside the realm cryptography, in particular to randomized algorithms and complexity.

REFERENCES

Abadi, M. and Cortier, V. 2006. Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci. 367,* 1-2, 2–32.

Abadi, M. and Rogaway, P. 2002. Reconciling two views of cryptography (The computational soundness of formal encryption). *J. Cryptology 15,* 2, 103–127.

Affeldt, R., Tanaka, M., and Marti, N. 2007. Formal proof of provable security by game-playing in a proof assistant. In *1st International Conference on Provable Security, ProvSec 2007*. Lecture Notes in Computer Science, vol. 4784. Springer, Berlin, 151–168.

Amtoft, T., Bandhakavi, S., and Banerjee, A. 2006. A logic for information flow in object-oriented programs. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*. ACM, New York, 91–102.

Audebaud, P. and Paulin-Mohring, C. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program. 74,* 8, 568–589.

Backes, M., Berg, M., and Unruh, D. 2008. A formal language for cryptographic pseudocode. In *15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2008*. Lecture Notes in Computer Science, vol. 5330. Springer, Berlin, 353–376.

Backes, M. and Laud, P. 2006. Computationally sound secrecy proofs by mechanized flow analysis. In *13th ACM Conference on Computer and Communications Security, CCS 2006*. ACM, New York, 370–379.

Backes, M., Maffei, M., and Unruh, D. 2010. Computationally sound verification of source code. In *17th ACM Conference on Computer and Communications Security, CCS 2010*. ACM, New York.

Backes, M. and Pfitzmann, B. 2005. Limits of the cryptographic realization of Dolev-Yao-style XOR. In *Computer Security – ESORICS 2005, 10th European Symposium on Research in Computer Security*. Lecture Notes in Computer Science, vol. 3679. Springer, Berlin, 178–196.

Barthe, G., Cederquist, J., and Tarento, S. 2004. A machine-checked formalization of the generic model and the random oracle model. In *Automated Reasoning, 2nd International Joint Conference, IJCAR 2004*. Lecture Notes in Computer Science, vol. 3097. Springer, Berlin, 385–399.

Barthe, G., Daubignard, M., Kapron, B., and Lakhnech, Y. 2010. Computational indistinguishability logic. In *17th ACM Conference on Computer and Communications Security, CCS 2010*. ACM, New York.

Barthe, G., Grégoire, B., Heraud, S., and Zanella Béguelin, S. 2009a. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International Workshop on Formal Aspects in Security and Trust, FAST 2008*. Lecture Notes in Computer Science, vol. 5491. Springer, Berlin, 1–19.

Barthe, G., Grégoire, B., and Zanella Béguelin, S. 2009b. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM, New York, 90–101.

Barthe, G., Grégoire, B., and Zanella Béguelin, S. 2010a. Programming language techniques for cryptographic proofs. In *1st International Conference on Interactive Theorem Proving, ITP 2010*. Lecture Notes in Computer Science, vol. 6172. Springer, Berlin, 115–130.

Barthe, G., Hedin, D., Zanella Béguelin, S., Gregoire, B., and Heraud, S. 2010b. A machine-checked formalization of Sigma-protocols. In *23rd IEEE Computer Security Foundations Symposium, CSF 2010*. IEEE Computer Society, Los Alamitos, Calif.A, 246–260.

Bellare, M. and Rogaway, P. 1993. Random oracles are practical: a paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security, CCS 1993*. ACM, New York, 62–73.

Bellare, M. and Rogaway, P. 1994. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT 1994*. Lecture Notes in Computer Science, vol. 950. Springer, Berlin, 92–111.

Bellare, M. and Rogaway, P. 1996. The exact security of digital signatures – How to sign with RSA and Rabin. In *Advances in Cryptology – EUROCRYPT 1996*. Lecture Notes in Computer Science, vol. 1070. Springer, Berlin, 399–416.

Bellare, M. and Rogaway, P. 2006. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*. Lecture Notes in Computer Science, vol. 4004. Springer, Berlin, 409–426.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*. ACM, New York, 14–25.

Bertot, Y., Grégoire, B., and Leroy, X. 2006. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 3839. Springer, Berlin, 66–81.

Blanchet, B. 2008. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput. 5,* 4, 193–207.

Blanchet, B., Jaggard, A. D., Scedrov, A., and Tsay, J.-K. 2008. Computationally sound mechanized proofs for basic and public-key Kerberos. In *15th ACM Conference on Computer and Communications Security, CCS 2008*. ACM, New York, 87–99.

Blanchet, B. and Pointcheval, D. 2006. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*. Lecture Notes in Computer Science, vol. 4117. Springer, Berlin, 537–554.

Boneh, D. 2001. Simplified OAEP for the RSA and Rabin functions. In *Advances in Cryptology – CRYPTO 2001*. Lecture Notes in Computer Science, vol. 2139. Springer, Berlin, 275–291.

Coron, J.-S. 2002. Optimal security proofs for PSS and other signature schemes. In *Advances in Cryptology – EUROCRYPT 2002*. Lecture Notes in Computer Science, vol. 2332. Springer, Berlin, 272–287.

Cortier, V., Kremer, S., and Warinschi, B. 2010. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 1–35.

Courant, J., Daubignard, M., Ene, C., Lafourcade, P., and Lakhnech, Y. 2008. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *15th ACM Conference on Computer and Communications Security, CCS 2008*. ACM, New York, 371–380.

den Hartog, J. 2008. Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic Hoare style logic. *Sci. Comput. Program. 74,* 1-2, 52–63.

Fujisaki, E. and Okamoto, T. 1999. How to enhance the security of public-key encryption at minimum cost. In *2nd International Workshop on Practice and Theory in Public Key Cryptography, PKC 1999*. Lecture Notes in Computer Science, vol. 1560. Springer, Berlin, 634–634.

Fujisaki, E., Okamoto, T., Pointcheval, D., and Stern, J. 2004. RSA-OAEP is secure under the RSA assumption. *J. Cryptology 17,* 2, 81–104.

Goldreich, O. 2001. *Foundations of Cryptography: Basic Tools*. Vol. 1. Cambridge University Press, Cambridge, UK.

Goldwasser, S. and Micali, S. 1984. Probabilistic encryption. *J. Comput. Syst. Sci. 28,* 2, 270–299.

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

Håstad, J., Impagliazzo, R., Levin, L. A., and Luby, M. 1999. A pseudorandom generator from any one-way function. *SIAM J. Comput. 28,* 4, 1364–1396.

Hofmann, M. 1998. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *11th International Workshop on Computer Science Logic, CSL 1997*. Lecture Notes in Computer Science, vol. 1414. Springer, Berlin, 275–294.

Hurd, J., McIver, A., and Morgan, C. 2005. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci. 346,* 1, 96–112.

Impagliazzo, R. and Kapron, B. M. 2006. Logics for reasoning about cryptographic constructions. *J. Comput. Syst. Sci. 72,* 2, 286–320.

Impagliazzo, R. and Rudich, S. 1989. Limits on the provable consequences of one-way permutations. In *21st Annual ACM Symposium on Theory of Computing, 1989*. ACM, New York, 44–61.

Jonsson, B., Yi, W., and Larsen, K. G. 2001. Probabilistic extensions of process algebras. In *Handbook of Process Algebra*, J. Bergstra, A. Ponse, and S. Smolka, Eds. Elsevier, Amsterdam, 685–710.

Kozen, D. 1981. Semantics of probabilistic programs. *J. Comput. Syst. Sci. 22,* 3, 328–350.

Laud, P. 2001. Semantics and program analysis of computationally secure information flow. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*. Lecture Notes in Computer Science, vol. 2028. Springer, Berlin, 77–91.

Leroy, X. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*. ACM, New York, 42–54.

Meadows, C. 2003. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE J. Sel. Areas Commun. 21,* 1, 44–54.

Nowak, D. 2007. A framework for game-based security proofs. In *9th International Conference on Information and Communications Security, ICICS 2007*. Lecture Notes in Computer Science, vol. 4861. Springer, Berlin, 319–333.

Okamoto, T. and Pointcheval, D. 2001. The gap-problems: A new class of problems for the security of cryptographic schemes. In *4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001*. Lecture Notes in Computer Science, vol. 1992. Springer, Berlin, 104–118.

Okamoto, T. and Pointcheval, D. 2001b. REACT: Rapid Enhanced-Security Asymmetric Cryptosystem Transform. In *Topics in Cryptology – CT-RSA 2001*. Lecture Notes in Computer Science, vol. 2020. Springer, Berlin, 159–174.

Pointcheval, D. 2005. Provable security for public key schemes. In *Advanced Courses on Contemporary Cryptology*. Birkhäuser Basel, Chapter D, 133–189.

Ramsey, N. and Pfeffer, A. 2002. Stochastic lambda calculus and monads of probability distributions. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002*. ACM, New York, 154–165.

Roy, A., Datta, A., Derek, A., and Mitchell, J. 2008. Inductive proofs of computational secrecy. In *Computer Security – ESORICS 2007, 12th European Symposium on Research In Computer Security*. Lecture Notes in Computer Science, vol. 4734. Springer, Berlin, 219–234.

Sabelfeld, A. and Sands, D. 2001. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation 14,* 1, 59–91.

Shoup, V. 2001. OAEP reconsidered. In *Advances in Cryptology – CRYPTO 2001*. Lecture Notes in Computer Science, vol. 2139. Springer, Berlin, 239–259.

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

Sprenger, C. and Basin, D. 2008. Cryptographically-sound protocol-model abstractions. In *21st IEEE Computer Security Foundations Symposium, CSF 2008*. IEEE Computer Society, Los Alamitos, Calif., 115–129.

Stern, J. 2003. Why provable security matters? In *Advances in Cryptology – EUROCRYPT 2003*. Lecture Notes in Computer Science, vol. 2656. Springer, Berlin, 644–644.

The Coq development team. 2009. The Coq Proof Assistant Reference Manual Version 8.2. Online – `http://coq.inria.fr`.

Zanella Béguelin, S., Grégoire, B., Barthe, G., and Olmedo, F. 2009. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy, S&P 2009*. IEEE Computer Society, Los Alamitos, Calif.A, 237–250.

Zhang, Y. 2009. The computational SLR: A logic for reasoning about computational indistinguishability. In *8th International Conference on Typed Lambda Calculi and Applications, TLCA 2008*. Lecture Notes in Computer Science, vol. 5608. Springer, Berlin, 401–415.