Formal Certification of Code-Based Cryptographic Proofs

Gilles Barthe[1,2] Benjamin Grégoire[1,3] Santiago Zanella[1,3]

[1] Microsoft Research - INRIA Joint Centre, France

[2] IMDEA Software, Madrid, Spain [3] INRIA Sophia Antipolis - Méditerranée, France

Gilles.Barthe@imdea.org {Benjamin.Gregoire,Santiago.Zanella}@sophia.inria.fr

Abstract

As cryptographic proofs have become essentially unverifiable, cryptographers have argued in favor of developing techniques that help tame the complexity of their proofs. Game-based techniques provide a popular approach in which proofs are structured as sequences of games, and in which proof steps establish the validity of transitions between successive games. Code-based techniques form an instance of this approach that takes a code-centric view of games, and that relies on programming language theory to justify proof steps. While code-based techniques contribute 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 code-based proofs. CertiCrypt is built upon the general-purpose proof assistant Coq, and draws on many areas, including probability, complexity, algebra, and semantics of programming languages. CertiCrypt 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 game-based techniques such as reasoning about failure events. The usefulness of CertiCrypt is demonstrated through various examples, including a proof of semantic security of OAEP (with a bound that improves upon existing published results), and a proof of existential unforgeability of FDH signatures. Our work provides a first yet significant step towards Halevi's ambitious programme 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—Operational semantics, Denotational semantics, Program analysis.

General Terms Languages, Security, Verification

1. Introduction

Provable security [33], whose origins can be traced back to the pioneering work of Goldwasser and Micali [18], advocates a mathematical approach based on complexity theory in which the goals and requirements of cryptosystems are specified precisely, and where security proofs are carried out rigorously and make all underlying assumptions explicit. In a typical provable security setting, one reasons about effective adversaries, modeled as arbitrary probabilistic polynomial-time Turing machines, and about their probability of thwarting a security objective, e.g. secrecy. In a similar fashion, security assumptions about cryptographic primitives bound the probability of polynomial algorithms to solve hard problems, e.g. computing discrete logarithms. The security proof is performed by reduction by showing that the existence of an effective adversary with a certain advantage in breaking security implies the existence of an effective algorithm contradicting the security assumptions. Although the adoption of provable security has significantly enhanced confidence in security proofs, several published proofs have been found incorrect (cf. [30]), and the cryptographic community is increasingly wary that the field may be approaching a crisis of rigor [8, 19].

The game-playing technique [8, 19, 31] is a general method to structure and unify cryptographic proofs, thus making them less error-prone. Its central idea is to view the interaction between an adversary and the cryptosystem as a game, and to study transformations that preserve security. In a typical game-based proof, one considers transitions of the form $G, A \rightarrow^h G', A'$, where G and G' are games, A and A' are events, and h is a monotonic function such that $\Pr_G[A] \leq h(\Pr_{G'}[A'])$. One can obtain an upper bound for the probability of an event A_0 in some initial game G_0 by successively refining G_0, A_0 into a game/event pair G_n, A_n,

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

and then bounding the probability of event A_n in G_n.

Code-based techniques [8] is an instance of the game-playing technique whose distinguishing feature is to take a code-centric view of games, security hypotheses and computational assumptions, that are expressed using (probabilistic, imperative, polynomial) programs. Under this view, game transformations become program transformations, and can be justified rigorously by semantic means; in particular, many transformations can be viewed as common program optimizations, and are justified by proving that the original and transformed programs are observationally equivalent. Although code-based proofs are easier to verify, they go far beyond established theories of program equivalence and exhibit a surprisingly rich and broad set of reasoning principles that draws on program verification, algebraic reasoning, and probability and complexity theory. Thus, despite the beneficial effect of their underlying framework, code-based proofs remain inherently complex. Whereas Bellare and Rogaway [8] already observed that code-based proofs could be more easily amenable to machine-checking, Halevi [19] argued that formal verification techniques should be used to improve trust in cryptographic proofs, and set up a programme for building a tool that could be used by the cryptographic community to mechanize their proofs.

This article reports on a first yet significant step towards Halevi's programme. We describe CertiCrypt, a framework to construct machine-checked code-based proofs in the Coq proof assistant [34], supporting:

Faithful and rigorous encoding of games. In order to be readily accessible to cryptographers, we have chosen a formalism that is commonly used to describe games. Concretely, the lowest layer of CertiCrypt is the formalization of pWHILE, an imperative programming language with random assignments, structured datatypes, and procedure calls. We provide a deep and dependently-typed embedding of the syntax; thanks to dependent types, the typability of pWHILE programs is obtained for free. We also provide a small-step operational semantics using the measure monad of Audebaud and Paulin [4]. 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. In addition, we also model nonstandard features, such as policies on variable accesses and procedure calls, and use them to capture many assumptions left informal in cryptographic proofs.

Exact security. Many security proofs establish an asymptotic behavior for adversaries and show that the advantage of any effective adversary is negligible w.r.t. a security parameter (which typically determines the length of keys or messages). However, the cryptographic community is increasingly focused on exact security, a much more useful result since it gives hints as to how to choose system parameters in practice to satisfy a security guarantee. The goal of exact security is to provide concrete bounds both for the advantage of the adversary and for its execution time. CertiCrypt supports the former (but for the time being, not the latter).

Full and independently verifiable proofs. CertiCrypt adopts a formal semanticist perspective and goes beyond Halevi's vision in two respects. First, it provides 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, groups, polynomials, etc). Second, one notable feature of Coq, and thus CertiCrypt, is to support independent verifiability of proofs, which is an important motivation behind game-based proofs. More concretely, every proof is represented by 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. CertiCrypt formalizes a Relational Hoare Logic and a theory of observational equivalence, and uses them as stepping stones to support the main tools of code-based reasoning through certified, reflective tactics. In particular, CertiCrypt shows that many transformations used in code-based proofs, including common optimizations, are semantics-preserving. One of its specific contributions is to prove formally the correctness of a variant of lazy sampling, which is used ubiquitously in cryptographic proofs. In addition, CertiCrypt supports methods based on failure events (the so-called fundamental lemma of game-playing).

We have successfully conducted nontrivial case studies that validate our design, show the feasibility of formally verifying cryptographic proofs, and confirm the plausibility of Halevi's programme.

Contents The purpose of this article is to provide an overview of the CertiCrypt project, and to stir further interest in machine-checked cryptographic proofs. Additional details on design choices, on formalizing the semantics of probabilistic programs, and on case studies, will be provided elsewhere. In consequence, the paper is organized as follows: we begin in Section 2 with two introductory examples of game-based proofs, namely the semantic security of ElGamal encryption, and the PRP/PRF switching lemma; in Section 3 we introduce the language we use to represent games and its semantics, and we discuss the notions of complexity and termination; in Section **??** we present a probabilistic relational Hoare logic that forms the core of our framework; in Sections 5 and 6 we overview the formulation and automation of game transformations in CertiCrypt; in Section 7 we report on two significant case studies we have formalized in CertiCrypt: existential unforgeability of the FDH signature scheme, and semantic security of OAEP; we finish with a discussion of related work and concluding remarks.

2. Basic examples

This section illustrates the principles of CertiCrypt on two basic examples of game-based proofs: semantic security of ElGamal encryption and the PRP/PRF switching lemma. The language used to represent games is formally introduced in the next section. We begin with some basic definitions.

The Random Oracle Model is a model of cryptography extensively used in security proofs in which some cryptographic primitives, e.g. hash functions, are assumed to be indistinguishable from random functions (despite the fact that no real function can implement a truly random function [13]). Such primitives are modeled by oracles that return random values in response to queries. The sole condition is that queries are answered consistently: if some value is queried twice, the same response must be given. Our formalism captures the notion of random oracle using stateful procedures that store queries and their results, e.g.

$$\begin{array}{l}\textbf{Oracle}\ \mathcal{O}(x) : \mathsf{if}\ x \notin \mathsf{dom}(L)\ \mathsf{then}\ y \xleftarrow{\$} \{0,1\}^{\eta};\ L \leftarrow (x,y) :: L; \\ \qquad\qquad\quad \mathsf{return}\ L[x]\end{array}$$

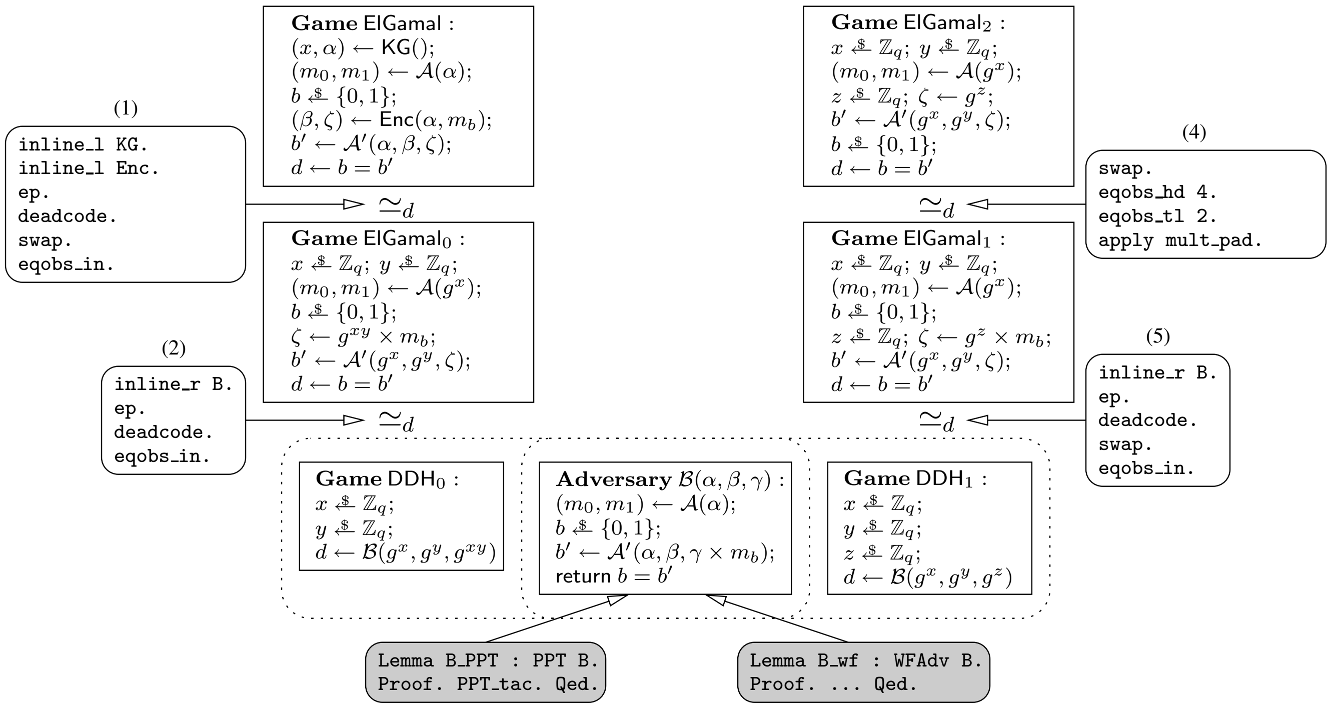
An asymmetric encryption scheme is composed of three algorithms: key generation $\mathsf{KG}(\eta)$, where η is the security parameter; encryption $\mathsf{Enc}(pk, m)$ where pk is a public key and m a plaintext; and decryption—not relevant here. An asymmetric encryption scheme is said to be semantically secure (equivalently, IND-CPA secure) if it is infeasible to gain significant information about a plaintext given only a corresponding ciphertext and the public key. This is formally defined using the following game, where $\mathcal{A}$ and $\mathcal{A}'$ are allowed to share state via global variables and thus are regarded as a single adaptive adversary:

$$\begin{array}{l}\textbf{Game}\ \mathsf{IND\text{-}CPA}: \\ (sk, pk) \leftarrow \mathsf{KG}(\eta); \\ (m_0, m_1) \leftarrow \mathcal{A}(pk); \\ b \xleftarrow{\$} \{0,1\}; \gamma \leftarrow \mathsf{Enc}(pk, m_b); \\ b' \leftarrow \mathcal{A}'(pk, \gamma)\end{array}$$

The game first generates a new key pair and gives the public key to the adversary, who returns two plaintexts m_0, m_1 of his choice. Then, the challenger 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 scheme is IND-CPA if for every effective adversary $\mathcal{A}, \mathcal{A}'$, $|\Pr_{\mathsf{IND\text{-}CPA}}[b = b'] - \frac{1}{2}|$ is negligible in the security parameter, i.e. the adversary cannot do much better than a blind guess. Formally, a function $\nu : \mathbb{N} \to \mathbb{R}$ is negligible iff $\forall\, c.\ \exists\, n_c.\ \forall\, n.\ n \geq n_c \Rightarrow |\nu(n)| \leq n^{-c}$.

2.1 The ElGamal encryption scheme

ElGamal is a widely used asymmetric encryption scheme, and an emblematic example of game-based proofs, as it embodies many of the techniques described in Sections **??** and 5. The proof fol-



```
Game ElGamal :
  $(x, \alpha) \leftarrow \mathsf{KG}()$;
  $(m_0, m_1) \leftarrow \mathcal{A}(\alpha)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $(\beta, \zeta) \leftarrow \mathsf{Enc}(\alpha, m_b)$;
  $b' \leftarrow \mathcal{A}'(\alpha, \beta, \zeta)$;
  $d \leftarrow b = b'$
```

```
inline_l KG.
inline_l Enc.
ep.
deadcode.
swap.
eqobs_in.
```

```
Game ElGamal_0 :
  $x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\zeta \leftarrow g^{xy} \times m_b$;
  $b' \leftarrow \mathcal{A}'(g^x, g^y, \zeta)$;
  $d \leftarrow b = b'$
```

```
inline_r B.
ep.
deadcode.
eqobs_in.
```

```
Game ElGamal_2 :
  $x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;
  $z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z$;
  $b' \leftarrow \mathcal{A}'(g^x, g^y, \zeta)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $d \leftarrow b = b'$
```

```
swap.
eqobs_hd 4.
eqobs_tl 2.
apply mult_pad.
```

```
Game ElGamal_1 :
  $x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z \times m_b$;
  $b' \leftarrow \mathcal{A}'(g^x, g^y, \zeta)$;
  $d \leftarrow b = b'$
```

```
inline_r B.
ep.
deadcode.
swap.
eqobs_in.
```

```
Game DDH_0 :
  $x \xleftarrow{\$} \mathbb{Z}_q$;
  $y \xleftarrow{\$} \mathbb{Z}_q$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^{xy})$
```

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

```
Game DDH_1 :
  $x \xleftarrow{\$} \mathbb{Z}_q$;
  $y \xleftarrow{\$} \mathbb{Z}_q$;
  $z \xleftarrow{\$} \mathbb{Z}_q$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^z)$
```

```
Lemma B_PPT : PPT B.
Proof. PPT_tac. Qed.
```

```
Lemma B_wf : WFAdv B.
Proof. ... Qed.
```

Figure 1. Code-based proof of ElGamal semantic security

lows [31]; all games are defined in Fig. 1. Given a cyclic group of order q, and a generator g, we define:[1]

- Key generation: $\mathsf{KG}() \overset{\text{def}}{=} x \xleftarrow{\$} \mathbb{Z}_q;\ \mathsf{return}\ (x, g^x)$
- Encryption: $\mathsf{Enc}(\alpha, m) \overset{\text{def}}{=} y \xleftarrow{\$} \mathbb{Z}_q;\ \mathsf{return}\ (g^y, \alpha^y \times m)$

ElGamal is IND-CPA secure under the Decisional Diffie-Hellman (DDH) assumption, which states that it is hard to distinguish between triples of the form (g^x, g^y, g^{xy}) and (g^x, g^y, g^z) where x, y, z are uniformly sampled in $\mathbb{Z}_q$. In our setting, DDH is formulated precisely by stating that for any polynomial-time and well-formed adversary $\mathcal{B}$, $|\Pr_{\mathsf{DDH}_0}[d] - \Pr_{\mathsf{DDH}_1}[d]|$ is negligible in the security parameter. Figure 1 presents a high level view of the proof: the square boxes represent games, whereas the rounded boxes represent proof sketches of the transitions between games; the tactics that appear in these boxes hopefully have self-explanatory names, but are explained in more detail in Section 5. The rounded grey boxes represent proof sketches of side conditions that guarantee that the DDH assumption is correctly applied. The proof proceeds by constructing an adversary $\mathcal{B}$ against DDH such that the distribution of $b = b'$ (i.e. d) after running the IND-CPA game ElGamal is exactly the same as the distribution of d after running DDH_0. Furthermore we show that the probability of d being true in DDH_1 is $\frac{1}{2}$ for the same adversary $\mathcal{B}$. The proof is summarized by the following equations:

$$\begin{aligned}
|\Pr_{\mathsf{ElGamal}}[b = b'] - \tfrac{1}{2}| &= |\Pr_{\mathsf{ElGamal}_0}[d] - \tfrac{1}{2}| && (1)\\
&= |\Pr_{\mathsf{DDH}_0}[d] - \tfrac{1}{2}| && (2)\\
&= |\Pr_{\mathsf{DDH}_0}[d] - \Pr_{\mathsf{ElGamal}_2}[d]| && (3)\\
&= |\Pr_{\mathsf{DDH}_0}[d] - \Pr_{\mathsf{ElGamal}_1}[d]| && (4)\\
&= |\Pr_{\mathsf{DDH}_0}[d] - \Pr_{\mathsf{DDH}_1}[d]| && (5)
\end{aligned}$$

[1] The security parameter, implicit in this presentation, determines this cyclic group by indexing a family of groups where the DDH problem is believed intractable.

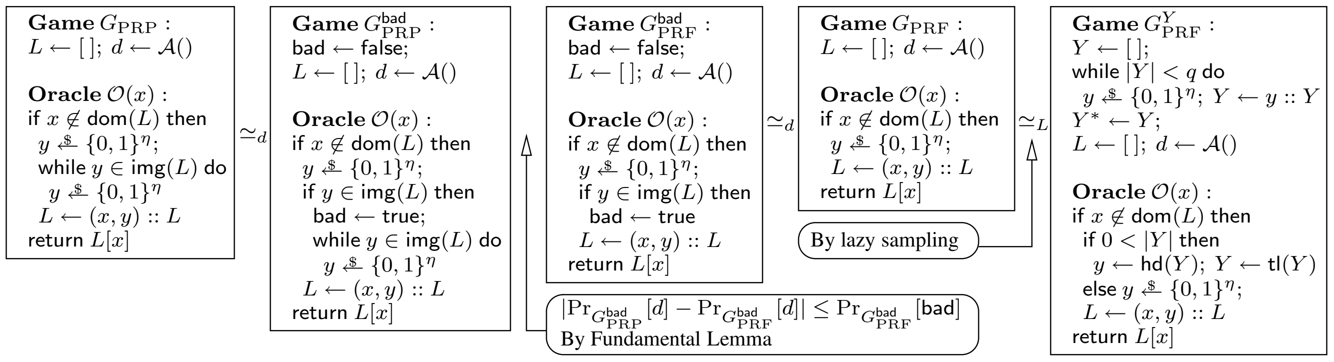
Equation (1) is justified because ElGamal and $\mathsf{ElGamal}_0$ induce the same distribution on d ($\mathsf{ElGamal} \simeq_d \mathsf{ElGamal}_0$). To prove this, we inline the calls to KG and Enc, and then perform expression propagation and dead code elimination (`ep`, `deadcode`). At this point we are left with two almost identical games, except the sampling of y is done later in one game than in the other. The tactic `swap` is used to hoist instructions whenever is possible in order to obtain a common prefix, and allows us to hoist the sampling of y to the right place. We conclude by applying `eqobs_in` that decides observational equivalence of a program with itself. Equations (2) and (5) are obtained similarly, while (3) holds because b' is independent from the sampling of b in $\mathsf{ElGamal}_2$. Finally, to prove equation (4) we begin by removing the common part of the two games with the exception of the instruction $z \xleftarrow{\$} \mathbb{Z}_q$ (`eqobs_hd`, `eqobs_tl`). We then apply an algebraic property of cyclic groups (`mult_pad`): when multiplying a uniformly distributed element of the group by another element, the result is uniformly distributed. 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.

The proof concludes by applying the DDH assumption. We prove that the adversary $\mathcal{B}$ is strict probabilistic polynomial-time and well-formed (under the assumption that $\mathcal{A}$ and $\mathcal{A}'$ are so). The proof of the former condition is automated in CertiCrypt.

2.2 The PRP/PRF switching lemma

In cryptographic proofs, particularly those dealing with blockciphers, it is often convenient to replace a pseudo-random permutation (PRP) by a pseudo-random function (PRF). The PRP/PRF switching lemma establishes that such a replacement does not change significantly the advantage of an effective adversary. In a code-based setting, the Switching Lemma states that

$$|\Pr_{G_{\mathrm{PRP}}}[d] - \Pr_{G_{\mathrm{PRF}}}[d]| \leq \frac{q(q-1)}{2^{\eta+1}}$$



```
Game $G_{\mathrm{PRP}}$ :
$L \leftarrow [\,]$; $d \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
if $x \notin \mathsf{dom}(L)$ then
  $y \xleftarrow{\$} \{0,1\}^\eta$;
  while $y \in \mathsf{img}(L)$ do
    $y \xleftarrow{\$} \{0,1\}^\eta$
  $L \leftarrow (x,y) :: L$
return $L[x]$
```

$\simeq_d$

```
Game $G^{\mathsf{bad}}_{\mathrm{PRP}}$ :
bad $\leftarrow$ false;
$L \leftarrow [\,]$; $d \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
if $x \notin \mathsf{dom}(L)$ then
  $y \xleftarrow{\$} \{0,1\}^\eta$;
  if $y \in \mathsf{img}(L)$ then
    bad $\leftarrow$ true;
    while $y \in \mathsf{img}(L)$ do
      $y \xleftarrow{\$} \{0,1\}^\eta$
  $L \leftarrow (x,y) :: L$
return $L[x]$
```

```
Game $G^{\mathsf{bad}}_{\mathrm{PRF}}$ :
bad $\leftarrow$ false;
$L \leftarrow [\,]$; $d \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
if $x \notin \mathsf{dom}(L)$ then
  $y \xleftarrow{\$} \{0,1\}^\eta$;
  if $y \in \mathsf{img}(L)$ then
    bad $\leftarrow$ true
  $L \leftarrow (x,y) :: L$
return $L[x]$
```

$|\Pr_{G^{\mathsf{bad}}_{\mathrm{PRP}}}[d] - \Pr_{G^{\mathsf{bad}}_{\mathrm{PRF}}}[d]| \leq \Pr_{G^{\mathsf{bad}}_{\mathrm{PRF}}}[\mathsf{bad}]$
By Fundamental Lemma

$\simeq_d$

```
Game $G_{\mathrm{PRF}}$ :
$L \leftarrow [\,]$; $d \leftarrow \mathcal{A}()$

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

By lazy sampling

$\simeq_L$

```
Game $G^{Y}_{\mathrm{PRF}}$ :
$Y \leftarrow [\,]$;
while $|Y| < q$ do
  $y \xleftarrow{\$} \{0,1\}^\eta$; $Y \leftarrow y :: Y$
$Y^* \leftarrow Y$;
$L \leftarrow [\,]$; $d \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
if $x \notin \mathsf{dom}(L)$ then
  if $0 < |Y|$ then
    $y \leftarrow \mathsf{hd}(Y)$; $Y \leftarrow \mathsf{tl}(Y)$
  else $y \xleftarrow{\$} \{0,1\}^\eta$;
  $L \leftarrow (x,y) :: L$
return $L[x]$
```

Figure 2. Code-based proof of the PRP/PRF switching lemma

where games G_{PRP} and G_{PRF} give the adversary access to an oracle that represents a random permutation and a random function respectively, and where q bounds the number of oracle queries made by $\mathcal{A}$.

The proof is split in two parts: the first part formalizes the intuition that the probability of the adversary outputting a given value is the same if a PRP is replaced by a PRF and no collisions are observed; it uses the Fundamental Lemma of game-playing (Lemma 2 in Sec. 6). The second part provides an upper bound to the probability of a collision; it uses lazy sampling (Lemma 1 in Sec. 5.2).

Figure 2 provides a high-level view of the proof. To apply the Fundamental Lemma, we introduce in the game G_{PRF} a variable bad that is set to true whenever a collision is found; we reformulate G_{PRP} accordingly to be syntactically equal until bad is set. Using `deadcode` to eliminate the variable bad, we show that the resulting games $G^{\mathsf{bad}}_{\mathrm{PRF}}$ and $G^{\mathsf{bad}}_{\mathrm{PRP}}$ are just semantics preserving reformulations of the games G_{PRF} and G_{PRP} respectively. Then, we apply the Fundamental Lemma to conclude that the difference in the probability of $d = \mathsf{true}$ between the two games is at most the probability of bad being set to true in game $G^{\mathsf{bad}}_{\mathrm{PRF}}$.

We then prove that the probability of bad being set to true in game $G^{\mathsf{bad}}_{\mathrm{PRF}}$ is upper bounded by the probability of an element appearing twice in the range of L in G_{PRF}. The proof uses the Relational Hoare Logic and Lemma ($\leq_{[\![]\!]}$) of Section **??** with the following postcondition: if bad is set in $G^{\mathsf{bad}}_{\mathrm{PRF}}$ then some element appears twice in the range of L in G_{PRF}. Next, we introduce a game G^{Y}_{PRF} where the answer to the first q queries to the oracle are sampled at the beginning of the game and stored in a list Y. Using lazy sampling, we prove by induction on q that the game G_{PRF} is equivalent to G^{Y}_{PRF} w.r.t L. Finally, we bound the probability of having a collision in L in G^{Y}_{PRF}. To that end, we prove that any collision in L is also present in Y^* provided the length of L is less than or equal to q (we use Y^* as a *ghost* variable to store the value of Y after being initialized). We conclude by bounding the probability of sampling some value twice in Y by $\frac{q(q-1)}{2^{\eta+1}}$.

3. Games as programs

The essence of code-based cryptographic proofs is to express in a unified semantic framework games, hypotheses, and results. This semanticist perspective allows a precise specification of the interaction between the adversary and the challenger in a game, and to readily answer questions as: Which oracles does the adversary have access to? Can the adversary read/write this variable? How many queries the adversary can make to a given oracle? What is the length of a bitstring returned by the adversary? Can the adversary repeat a query? Furthermore, other notions such as probabilistic polynomial-time complexity or termination fit naturally in the same framework and complete the specification of adversaries and games.

3.1 The pWHILE language

Games are formalized in pWHILE, a probabilistic imperative language with procedure calls. Given a set $\mathcal{V}$ of variable identifiers, a set $\mathcal{P}$ of procedure identifiers, a set $\mathcal{E}$ of expressions, and a set $\mathcal{D}$ of *distribution expressions*, the set of commands can be defined inductively by the clauses:

$\mathcal{I}$	$::=$	$\mathcal{V} \leftarrow \mathcal{E}$	assignment
	$\mid$	$\mathcal{V} \xleftarrow{\$} \mathcal{D}$	random sampling
	$\mid$	if $\mathcal{E}$ then $\mathcal{C}$ else $\mathcal{C}$	conditional
	$\mid$	while $\mathcal{E}$ do $\mathcal{C}$	while loop
	$\mid$	$\mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E})$	procedure call
$\mathcal{C}$	$::=$	nil	nop
	$\mid$	$\mathcal{I};\ \mathcal{C}$	sequence

Rather than adopting the above definition, we impose that programs in pWHILE are typed. Thus, $x \leftarrow e$ is well-formed only if the types of x and e coincide, and if e then c_1 else c_2 is well-formed only if e is a boolean expression. In practice, we assume that variables and values are typed, and define a dependently typed syntax of programs. An immediate benefit of using dependent types is that the type system of Coq ensures for free the well-typedness of expressions and commands. Although the formalization is carefully designed for being extensible w.r.t. user-defined types and operators (and we do exploit this in practice), it is sufficient for the purpose of this paper to consider an instance in which values are booleans, bitstrings, natural numbers, pairs, lists, and elements of a cyclic group. Similarly, we instantiate $\mathcal{D}$ so that values can be uniformly sampled from the set of booleans, natural intervals of the form $[0..n]$, and bitstrings of a certain length. It is important to note that the formalization of expressions is not restricted to many-sorted algebra: we make a critical use of dependent types to record the length of bitstrings. This is used e.g. in the definition of the IND-CPA game for OAEP in Sec. 7.2 to constrain the adversary to return two bitstrings of equal length.

Definition 1 (Program). *A program consists of a command and an environment, which maps a procedure identifier to its declaration, consisting of its formal parameters, its body, and a return expression (we use an explicit* return *when writing games, though),*

$$\mathit{decl} \stackrel{\mathit{def}}{=} \{\mathit{params} : \mathit{list}\ \mathcal{V};\ \mathit{body} : \mathcal{C};\ \mathit{re} : \mathcal{E}\}\ .$$

The environment specifies the type of the parameters and the return expression, so that procedure calls are always well-typed.

In a typical formalization, the environment will map procedures to closed commands, with the exception of the 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 of this section we assume an environment E implicitly given.

In the rest of this paper we let c_i range over $\mathcal{C}$; x_i over $\mathcal{V}$; e_i over $\mathcal{E}$; d_i over $\mathcal{D}$; and G_i over programs. The operator $\oplus$ denotes the bitwise exclusive or on bitstrings of equal length, and $\|$ the concatenation of two bitstrings.

3.2 Operational semantics

Programs in pWHILE are given a small-step semantics using the measure monad $M(X)$, whose type constructor is defined as

$$M(X) \stackrel{\text{def}}{=} (X \to [0,1]) \to [0,1]$$

and whose operators unit and bind are defined as

$$\begin{array}{lll} \mathsf{unit} & : & X \to M(X) \stackrel{\text{def}}{=} \lambda x.\, \lambda f.\, f\ x \\ \mathsf{bind} & : & M(X) \to (X \to M(Y)) \to M(Y) \\ & \stackrel{\text{def}}{=} & \lambda \mu.\, \lambda M.\, \lambda f.\, \mu(\lambda\, x.\, M\ x\ f) \end{array}$$

The monad $M(X)$ was proposed by Audebaud and Paulin [4] as a variant of the expectation monad used by Ramsey and Pfeffer [27], and builds on earlier work by Kozen [22]. The formalization of the semantics heavily relies on Paulin's axiomatization in Coq of the $[0, 1]$ real interval—for our purposes, it has been necessary to add division to the library.

The semantics of commands and expressions are defined relative to a given memory, i.e. a mapping from variables to values. We let $\mathcal{M}$ denote the set of memories. Expressions are deterministic; their semantics is standard and given by a function $[\![\cdot]\!]_{\text{expr}}$, that evaluates an expression in a given memory and returns a value. The semantics of distribution expressions is given by a function $[\![\cdot]\!]_{\text{distr}}$. For a distribution expression d of type T, we have that $[\![d]\!]_{\text{distr}} : \mathcal{M} \to M(X)$, where X is the interpretation of type T. For instance, in the previous section we have used $\{0,1\}^\eta$ to denote the uniform distribution on bitstrings of length η (the security parameter), formally, we have $[\![\{0,1\}^\eta]\!]_{\text{distr}} \stackrel{\text{def}}{=} \lambda m\ f.\ \sum_{bs \in \{0,1\}^\eta} \frac{1}{2^\eta} f(bs)$. 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 $[\![\cdot]\!]_{\text{expr}}$ and $[\![\cdot]\!]_{\text{distr}}$.

The semantics of commands relates a deterministic state to a (sub-)probability distribution over deterministic states and uses a frame stack to deal with procedure calls. Formally, a deterministic state is a triple consisting of the current command $c : \mathcal{C}$, a memory $m : \mathcal{M}$, and a frame stack F : list frame. We let $\mathcal{S}$ denote the set of deterministic states. One step execution $[\![\cdot]\!]^1 : \mathcal{S} \to M(\mathcal{S})$ is defined by the rules of Fig. 3. In the figure, we use $a \rightsquigarrow b$ as a notation for $[\![a]\!]^1 = b$ and loc and glob to project memories on local and global variables respectively.

We briefly comment on the transition rules for calling a procedure (3rd rule) and returning from a call (2nd rule). Upon a call, a new frame is appended to the stack, containing the destination variable, the return expression of the called procedure, the continuation to the call, and the local memory of the caller. The state resulting from the call contains the body of the called procedure, the global part of the memory, a local memory initialized to map the formal parameters to the value of the actual parameters just before the call, and the updated stack. When returning from a call with a non-empty stack, the top frame is popped, the return expression is evaluated and the resulting value is assigned to the destination variable after previously restoring the local memory of the caller; the continuation taken from the frame becomes the current command. If the stack is empty when returning from a call, the execution of the program terminates and the final state is embedded into the monad using the unit operator.

Using the monadic constructions, one can define an n-step execution function $[\![\cdot]\!]_n$:

$$[\![s]\!]_0 \stackrel{\text{def}}{=} \mathsf{unit}\ s \qquad [\![s]\!]_{n+1} \stackrel{\text{def}}{=} \mathsf{bind}\ [\![s]\!]_n\ [\![\cdot]\!]^1$$

Finally, the denotation of a command c in an initial memory m is defined to be the (limit) distribution of reachable final memories:

$$[\![c]\!]\ m : M(\mathcal{M}) \stackrel{\text{def}}{=} \lambda f.\ \sup\ \{[\![(c, m, [\,])]\!]_n\ f|_{\mathsf{final}} \mid n \in \mathbb{N}\}$$

where $f|_{\mathsf{final}} : \mathcal{S} \to [0,1]$ is the function that when applied to a state (c, m, F) gives $f(m)$ if it is a final state and 0 otherwise. Since the sequence $[\![(c, m, [\,])]\!]_n\ f|_{\mathsf{final}}$ is increasing and upper bounded by 1, this least upper bound always exists and corresponds to the limit of the sequence.

We have shown that the semantics is discrete, we use this to apply a variant of Fubini's theorem for proving the rules [R-Comp] and [R-Trans] in the next section.

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 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 $x \stackrel{\$}{\leftarrow} [0..1];\ y \stackrel{\$}{\leftarrow} [0..1]$ in the memory m is

$$\begin{array}{l} \lambda f. \frac{1}{4}(f(m\{0,0/x,y\}) + f(m\{0,1/x,y\}) \\ \quad + f(m\{1,0/x,y\}) + f(m\{1,1/x,y\})) \end{array}$$

and conclude that the probability of the event $x \leq y$ after executing the command above is $\frac{3}{4}$.

3.3 Probabilistic polynomial-time programs

In general, cryptographic proofs reason about effective adversaries, which can only use a polynomially bounded number of resources. The complexity notion that captures this intuition, and which is pervasive in cryptographic proofs, is that of *strict probabilistic polynomial-time*. Concretely, a program is said to be strict probabilistic polynomial-time (PPT) whenever there exists a polynomial bound (on some security parameter η) on the cost of each possible execution, regardless of the outcome of its coin tosses. Otherwise said, 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:

$$b \leftarrow \mathsf{true};\ \mathsf{while}\ b\ \mathsf{do}\ b \stackrel{\$}{\leftarrow} \{0,1\}$$
$$b \stackrel{\$}{\leftarrow} \{0,1\};\ \mathsf{if}\ b\ \mathsf{then\ while\ true\ do\ nil}$$

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 $\frac{1}{2}$, but when it does, it takes only a constant time. We deal with termination and efficiency separately.

Definition 2 (Termination)**.** *The probability that a program c terminates starting from an initial memory m is $[\![c]\!]\ m\ \mathbb{1}_{\mathsf{true}}$. 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.*

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 $M(\mathcal{M} \times \mathbb{N})$ instead of simply $M(\mathcal{M})$. We recall that our semantics is implicitly

$$\begin{array}{rcll}
(\mathsf{nil}, m, [\,]) & \rightsquigarrow & \mathsf{unit}\ (\mathsf{nil}, m, [\,]) & \\
(\mathsf{nil}, m, (x, e, c, l) :: F) & \rightsquigarrow & \mathsf{unit}\ (c, (l, m.\mathsf{glob})\{[\![e]\!]\ m/x\}, F) & \\
(x \leftarrow p(\vec{e});\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (E(p).\mathsf{body}, (\emptyset\{[\![\vec{e}]\!]\ m/E(p).\mathsf{params}\}, m.\mathsf{glob}), (x, E(p).\mathsf{re}, c, m.\mathsf{loc}) :: F) & \\
(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c_1;\ c, m, F) & \mathsf{if}\ [\![e]\!]\ m = \mathsf{true} \\
(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c_2;\ c, m, F) & \mathsf{if}\ [\![e]\!]\ m = \mathsf{false} \\
(\mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \rightsquigarrow & \mathsf{unit}\ (c;\ \mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \mathsf{if}\ [\![e]\!]\ m = \mathsf{true} \\
(\mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \rightsquigarrow & \mathsf{unit}\ (c', m, F) & \mathsf{if}\ [\![e]\!]\ m = \mathsf{false} \\
 & & & \\
(x \leftarrow e;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c, m\{[\![e]\!]\ m/x\}, F) & \\
(x \overset{\$}{\leftarrow} d;\ c, m, F) & \rightsquigarrow & \mathsf{bind}\ ([\![d]\!]\ m)(\lambda v.\ \mathsf{unit}\ (c, m\{v/x\}, F)) &
\end{array}$$

Figure 3. Probabilistic semantics of pWHILE programs

parametrized by a security parameter η, on which we base our notion of complexity.

Definition 3 (Polynomially bounded distribution)**.** *We say that a distribution $\mu : M(\mathcal{M} \times \mathbb{N})$ is (p, q)-bounded, where p and q are polynomials, whenever for every (m, n) occurring with non-zero probability in μ, the size of every value in the memory m is bounded by $p(\eta)$ and the cost n is bounded by $q(\eta)$. This notion is formally defined by means of the* range *predicate introduced in Sec.* **??**.

Definition 4 (Strict probabilistic polynomial-time program)**.** *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 μ, the distribution* $(\mathsf{bind}\ \mu\ [\![c]\!])$ *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 required for proving compositionality results (e.g. the sequential composition of two PPT programs results in a PPT program).

Although our formalizations of termination and efficiency rely 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 is lossless.[2] CertiCrypt also provides 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 superpolynomial size or take a superpolynomial time when evaluated in a polynomially bounded memory.

3.4 Adversaries

In order to reason formally about security, we make explicit which variables and procedures are accessible to adversaries, and provide a simple analysis to check whether an adversary respects its policy. Given a set of procedure identifiers $\mathcal{O}$ (the procedures that may be called by the adversary), and sets of global variables $\mathcal{G}_{\mathcal{A}}$ (those that can be read and written by the adversary) and $\mathcal{G}_{\mathrm{ro}}$ (those that the adversary can only read), we say that an adversary $\mathcal{A}$ is well-formed in an environment E if the judgment $\vdash_{\mathrm{wf}} \mathcal{A}$ can be derived using the rules in Fig. 4. These rules guarantee that each time a variable is written by the adversary, the adversary has the right to do so; and that each time a variable is read by the adversary, it is either a global variable in $\mathcal{G}_{\mathcal{A}} \cup \mathcal{G}_{\mathrm{ro}}$ or a local variable previously initialized. A well-formed adversary is free to call oracles, but any other procedure it calls must be a well-formed adversary itself.

$$I \vdash \mathsf{nil} : I \qquad \frac{I \vdash i : I' \quad I' \vdash c : O}{I \vdash i;\ c : O}$$

$$\frac{\mathsf{Writable}(x) \quad \mathsf{fv}(e) \subseteq I}{I \vdash x \leftarrow e : I \cup \{x\}} \qquad \frac{\mathsf{Writable}(x) \quad \mathsf{fv}(d) \subseteq I}{I \vdash x \overset{\$}{\leftarrow} d : I \cup \{x\}}$$

$$\frac{\mathsf{fv}(e) \subseteq I \quad I \vdash c_i : O_i \quad i = 1, 2}{I \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : O_1 \cap O_2} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash c : I}{I \vdash \mathsf{while}\ e\ \mathsf{do}\ c : I}$$

$$\frac{\mathsf{fv}(\vec{e}) \subseteq I \quad \mathsf{Writable}(x) \quad o \in \mathcal{O}}{I \vdash x \leftarrow o(\vec{e}) : I \cup \{x\}}$$

$$\frac{\mathsf{fv}(\vec{e}) \subseteq I \quad \mathsf{Writable}(x) \quad \mathcal{A} \notin \mathcal{O} \quad \vdash_{\mathrm{wf}} \mathcal{A}}{I \vdash x \leftarrow \mathcal{A}(\vec{e}) : I \cup \{x\}}$$

$$\frac{\mathcal{G}_{\mathcal{A}} \cup \mathcal{G}_{\mathrm{ro}} \cup \mathcal{A}_E.\mathsf{params} \vdash \mathcal{A}_E.\mathsf{body} : O \quad \mathsf{fv}(\mathcal{A}_E.\mathsf{re}) \subseteq O}{\vdash_{\mathrm{wf}} \mathcal{A}}$$

$$\mathsf{Writable}(x) \overset{\mathrm{def}}{=} \mathsf{Local}(x) \vee x \in \mathcal{G}_{\mathcal{A}} \quad \mathcal{A}_E \overset{\mathrm{def}}{=} E(\mathcal{A}).$$

Figure 4. Static analysis for well-formedness of adversaries

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, whereas for some properties such as IND-CCA2 there are some restrictions on the parameters with which the oracles may be called. Likewise, some proofs impose extra conditions such as forbidding repeated or malformed queries. These kinds of properties can be formalized using lists that record the oracle calls, and verifying as postcondition that the calls were legitimate.

4. Relational Hoare Logic

Shoup [31] classifies proof steps into three categories: transitions based on indistinguishability—which typically involve applying a security hypothesis, e.g. the DDH assumption—; transitions based on failure events—which typically amount to bound the probability of bad, as in the Switching Lemma—; and bridging steps—which correspond to replacing or reorganizing code in a way that is not observable by adversaries. In some circumstances, a bridging transition from G_1 to G_2 may replace a program fragment P by another fragment P' observationally equivalent to P. In general, however,

[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, and left for future work.

P and P' are only observationally equivalent in the context where the replacement is done. Such transitions are supported through a relational Hoare logic, that generalizes observational equivalence through preconditions and postconditions which we use to characterize the context where the replacement is valid. Besides, we use relational Hoare logic to establish (in)equalities between probabilities of two events, as shown by the lemmas $(=_{[\![]\!]})$ and $(\leq_{[\![]\!]})$ below, and to establish program invariants, e.g. in the proof of the Switching Lemma in Sec. 2.2.

4.1 Probabilistic Relational Hoare Logic (pRHL)

Our logic pRHL elaborates on and extends to probabilistic programs Benton's Relational Hoare Logic [9]. Benton's logic uses judgments of the form $\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi$, and relates the evaluation of a program G_1 to the evaluation of a program G_2 w.r.t. a precondition Ψ and a postcondition Φ, both defined as relations on deterministic states. Such a judgment states that for any initial memories m_1 and m_2 satisfying the precondition $m_1\ \Psi\ m_2$, if the evaluations of G_1 in m_1 and G_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 w.r.t. an initial memory yields a (sub-)distribution. In order to give a meaning to the above judgment, one therefore needs to lift relations over memories into relations over distributions.[3] We follow early work on probabilistic bisimulations [21]. The lifting to distributions of a unary predicate P and of a binary relation Φ are respectively defined as

$$\begin{array}{lcl}\mathsf{range}\ P\ \mu & \stackrel{\text{def}}{=} & \forall f.\ (\forall a.\ P\ a \Rightarrow f\ a = 0) \Rightarrow \mu\ f = 0 \\ \mu_1 \sim_\Phi \mu_2 & \stackrel{\text{def}}{=} & \exists \mu.\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{range}\ \Phi\ \mu\end{array}$$

where the projections of μ are defined as

$$\pi_1(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda(x,y).\mathsf{unit}\ x) \quad \pi_2(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda(x,y).\mathsf{unit}\ y)$$

This way of lifting relations over memories to relations over distributions is a generalization to arbitrary relations of the definition of Sabelfeld and Sands [29]. Their definition applies only to PERs, whereas our definition applies to arbitrary relations. Nevertheless, both definitions coincide for PERs as established by Jonsson, Larsen and Yi [21].

Definition 5 (pRHL judgments)**.** *Programs G_1 and G_2 are equivalent w.r.t. precondition Ψ and postcondition Φ iff*

$$\begin{array}{l}\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=} \\ \qquad \forall\, m_1\, m_2.\ m_1\ \Psi\ m_2 \Rightarrow [\![G_1]\!]\ m_1 \sim_\Phi [\![G_2]\!]\ m_2\end{array}$$

Our approach slightly departs from Benton's: rather than defining the rules for pRHL and proving them sound w.r.t. 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 5 gathers some representative derived rules. To improve readability, in the figure and in the remainder of the paper we let $e\langle i\rangle$ denote $\lambda m_1\ m_2.\ [\![e]\!]\ m_i = \mathsf{true}$, where e is a boolean expression. As pRHL allows for arbitrary relations, we freely use higher-order logic; in particular, PER and SYM are predicates over relations that stand for *partial equivalence relation* and *symmetric relation* respectively. There are two points worth noting. First, most rules admit, in addition to their symmetrical version of Fig. 5, one-sided (left and right) variants, e.g. for assignments

$$\frac{m_1\ \Phi'\ m_2 \stackrel{\text{def}}{=} (m_1\{[\![e_1]\!]m_1/x_1\})\ \Phi\ m_2}{\vDash E_1, x_1 \leftarrow e_1 \sim E_2, \mathsf{nil} : \Phi' \Rightarrow \Phi}$$

Second, some rules of pRHL do not appear in RHL, or generalize existing rules. The rule [R-Case] allows to do a case analysis on the evaluation of an arbitrary relation in the initial memories. Together with simple rules in the spirit of

$$\frac{\vDash E_1, c_1 \sim E_2, c : \Psi \wedge e\langle 1\rangle \Rightarrow \Phi}{\vDash E_1, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sim E_2, c : \Psi \wedge e\langle 1\rangle \Rightarrow \Phi}$$

it subsumes [R-Cond] and allows to prove judgments that would otherwise not be derivable, such as the equivalence between (if e then c_1 else c_2) and (if $\neg e$ then c_2 else c_1). We also use [R-Case] to prove the correctness of dataflow analyses that exploit the information provided by entering branches.

In addition, we often use the rule [R-Inv] that generalizes the rule [R-Sym] to inverse of relations

$$\frac{\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi}{\vDash G_2 \sim G_1 : \Psi^{-1} \Rightarrow \Phi^{-1}}\text{[R-Inv]}$$

and we make an extensive use of the rule [R-Comp] that generalizes the rule [R-Tr] to composition of relations[4]

$$\frac{\vDash G_1 \sim G : \Psi' \Rightarrow \Phi' \qquad \vDash G \sim G_2 : \Psi'' \Rightarrow \Phi''}{\vDash G_1 \sim G_2 : \Psi' \circ \Psi'' \Rightarrow \Phi' \circ \Phi''}\text{[R-Comp]}$$

The benefits of the rule [R-Comp], as opposed to [R-Tr], are illustrated by considering "independent" preconditions and postconditions of the form

$$\Psi \stackrel{\text{def}}{=} \lambda x\ y\ .\Psi_1\ x \wedge \Psi_2\ y \qquad\qquad \Phi \stackrel{\text{def}}{=} \lambda x\ y\ .\Phi_1\ x \wedge \Phi_2\ y$$

In order to apply the rule [R-Tr] to G_1 and G_2, we are essentially forced to have $\Psi_1 = \Psi_2$ and $\Phi_1 = \Phi_2$, and furthermore we must also choose the same pre and postcondition for the intermediate game G. This constraint makes the rule [R-Tr] impractical, we use instead the rule [R-Comp] to introduce intermediate games that do not satisfy the same specification as G_1 or G_2.

The rule [R-Rand] is also (obviously) not present in RHL. Let $\mathbb{I}_x \stackrel{\text{def}}{=} (\lambda v.\text{if } x = v \text{ then } 1 \text{ else } 0)$, and define the support of a distribution, $\mathsf{supp}([\![d]\!]\ m)$, by the clause

$$v \in \mathsf{supp}([\![d]\!]\ m) \Leftrightarrow [\![d]\!]\ m\ \mathbb{I}_v \neq 0$$

Finally, let $[\![d_1]\!]\ m_1 =_g [\![d_2]\!]\ m_2$ iff there exists a set X and a bijection $g : X \rightarrow X$ such that $\mathsf{supp}([\![d_1]\!]\ m_1) = \mathsf{supp}([\![d_2]\!]\ m_2) = X$ and $[\![d_1]\!]\ m_1\ \mathbb{I}_a = [\![d_2]\!]\ m_2\ \mathbb{I}_{(g\ a)}$ for all a in X. To apply rule [R-Rand], it is necessary to exhibit a function f such that for all memories m_1 and m_2, $[\![d_1]\!]\ m_1 =_{f\ m_1\ m_2} [\![d_2]\!]\ m_2$. Thus, if $d_1 = d_2 = [0..n]$ for some constant n, and we take f to be the identity function, the premise simplifies to the expected,

$$m_1\ \Psi\ m_2 \stackrel{\text{def}}{=} \forall\, v \in [0..n].\ (m_1\{v/x_1\})\ \Phi\ (m_2\{v/x_2\})$$

Section 5.3 shows that the generality of the rule is required for applications such as optimistic sampling.

It is often fruitful to understand pRHL judgments in terms of the inability of the postcondition to separate between the two commands of the judgment. Define two functions f and g to be equivalent w.r.t. a predicate Φ iff

$$f =_\Phi g \stackrel{\text{def}}{=} \forall\, m_1\, m_2.\ m_1\ \Phi\ m_2 \Rightarrow f(m_1) = g(m_2)$$

The definition of pRHL judgments entails

$$\left.\begin{array}{c}\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi \\ f =_\Phi g \\ m_1\ \Psi\ m_2\end{array}\right\} \Rightarrow [\![G_1]\!]\ m_1\ f = [\![G_2]\!]\ m_2\ g \qquad (=_{[\![]\!]})$$

[3] An alternative would be to develop a logic in which Ψ and Φ are relations over distributions. However, it is not clear whether such a logic would allow a similar level of proof automation. This is left for future work.

[4] The machine-checked rule requires that Φ, Φ' are decidable, and uses **Set**-valued existential quantification $\exists_{\mathbf{Set}}$ in the composition for preconditions, i.e. $x\ (\Psi \circ \Psi')\ z \stackrel{\text{def}}{=} \exists_{\mathbf{Set}} y.\ x\ \Psi\ y \wedge y\ \Psi' z$.

$$\vDash E_1, \mathsf{nil} \sim E_2, \mathsf{nil} : \Phi \Rightarrow \Phi \text{ [R-Skip]} \qquad \frac{\vDash E_1, c_1 \sim E_2, c_2 : \Phi \Rightarrow \Phi' \quad \vDash E_1, c_1' \sim E_2, c_2' : \Phi' \Rightarrow \Phi''}{\vDash E_1, c_1; c_1' \sim E_2, c_2; c_2' : \Phi \Rightarrow \Phi''} \text{[R-Seq]}$$

$$\vDash E_1, x_1 \leftarrow e_1 \sim E_2, x_2 \leftarrow e_2 : (\lambda\, m_1\, m_2.\ (m_1\{[\![e_1]\!]m_1/x_1\})\ \Phi\ (m_2\{[\![e_2]\!]m_2/x_2\})) \Rightarrow \Phi \text{ [R-Ass]}$$

$$\frac{m_1\ \Psi\ m_2 \overset{\text{def}}{=} \quad [\![d_1]\!]m_1 =_{f\ m_1\ m_2} [\![d_2]\!]m_2 \wedge \forall\, v \in \mathsf{supp}([\![d_1]\!]m_1).\ (m_1\{v/x_1\})\ \Phi\ (m_2\{f\ m_1\ m_2\ v/x_2\})}{\vDash E_1, x_1 \xleftarrow{\$} d_1 \sim E_2, x_2 \xleftarrow{\$} d_2 : \Psi \Rightarrow \Phi} \text{[R-Rand]}$$

$$\frac{\forall m_1\, m_2.\ m_1\ \Psi\ m_2 \Rightarrow [\![e_1]\!]\ m_1 = [\![e_2]\!]\ m_2 \quad \vDash E_1, c_1 \sim E_2, c_2 : \Psi \wedge e_1\langle 1\rangle \Rightarrow \Phi \quad \vDash E_1, c_1' \sim E_2, c_2' : \Psi \wedge \neg e_1\langle 1\rangle \Rightarrow \Phi}{\vDash E_1, \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim E_2, \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' : \Psi \Rightarrow \Phi} \text{[R-Cond]}$$

$$\frac{\forall m_1\, m_2.\ m_1\ \Phi\ m_2 \Rightarrow [\![e_1]\!]\ m_1 = [\![e_2]\!]\ m_2 \quad \vDash E_1, c_1 \sim E_2, c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Phi}{\vDash E_1, \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim E_2, \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Phi \Rightarrow \Phi \wedge \neg e_1\langle 1\rangle} \text{[R-Whl]}$$

$$\frac{\vDash G_1 \sim G_2 : \Psi' \Rightarrow \Phi' \quad \forall m_1\, m_2.\ m_1\ \Psi\ m_2 \Rightarrow m_1\ \Psi'\ m_2 \quad \forall m_1\, m_2.\ m_1\ \Phi'\ m_2 \Rightarrow m_1\ \Phi\ m_2}{\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi} \text{[R-Sub]}$$

$$\frac{\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi \quad \mathsf{SYM}(\Psi) \quad \mathsf{SYM}(\Phi)}{\vDash G_2 \sim G_1 : \Psi \Rightarrow \Phi} \text{[R-Sym]} \qquad \frac{\vDash G_1 \sim G : \Psi \Rightarrow \Phi \quad \vDash G \sim G_2 : \Psi \Rightarrow \Phi \quad \mathsf{PER}(\Psi) \quad \mathsf{PER}(\Phi)}{\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi} \text{[R-Tr]}$$

$$\frac{\vDash G_1 \sim G_2 : \Psi \wedge \Psi' \Rightarrow \Phi \quad \vDash G_1 \sim G_2 : \Psi \wedge \neg\, \Psi' \Rightarrow \Phi}{\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi} \text{[R-Case]}$$

Figure 5. Selection of derived rules of pRHL

By instantiating f and g to $\mathbb{1}$, one can observe that observational equivalence enjoys some form of termination sensitivity

$$(\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi) \wedge m_1\ \Psi\ m_2 \Rightarrow [\![G_1]\!]\ m_1\ \mathbb{1} = [\![G_2]\!]\ m_2\ \mathbb{1}$$

This interpretation of pRHL judgments is strongly connected to the relation between relational logics and information flow [3, 9]. We extensively use $(=_{[\![]\!]})$, and its variant $(\leq_{[\![]\!]})$ below, to fall back from the world of pRHL into the world of probabilities, in which security statements are expressed;

$$\left.\begin{array}{c} \vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi \\ f \leq_\Phi g \\ m_1\ \Psi\ m_2 \end{array}\right\} \Rightarrow [\![G_1]\!]\ m_1\ f \leq [\![G_2]\!]\ m_2\ g \qquad (\leq_{[\![]\!]})$$

where $f \leq_\Phi g \overset{\text{def}}{=} \forall\, m_1\, m_2.\ m_1\ \Phi\ m_2 \Rightarrow f(m_1) \leq g(m_2)$.

We conclude with an example that nicely illustrates some of the intricacies of pRHL. Let $c = b \xleftarrow{\$} \{0, 1\}$ and define $m_1\ \Phi\ m_2 \overset{\text{def}}{=} m_1\ b = m_2\ b$. Then $\vDash c \sim c : \mathsf{true} \Rightarrow \Phi$. Indeed, take μ such that $\mu\ \mathbb{1}_{\langle 0,0\rangle} = \mu\ \mathbb{1}_{\langle 1,1\rangle} = 1/2$ and $\mu\ \mathbb{1}_{\langle 0,1\rangle} = \mu\ \mathbb{1}_{\langle 1,0\rangle} = 0$. One can check that μ ensures $[\![c]\!]\ m \sim_\Phi [\![c]\!]\ m'$ for all m and m'. This example shows why the lifting of a binary relation involves an existential quantification, and why it is not possible to always instantiate μ as the product distribution in the definition of $\sim_\Phi$ (one cannot establish the above judgment using the product distribution). Perhaps more surprisingly, $\vDash c \sim c : \mathsf{true} \Rightarrow \neg\Phi$ also holds. Indeed, take μ such that $\mu\ \mathbb{1}_{\langle 0,0\rangle} = \mu\ \mathbb{1}_{\langle 1,1\rangle} = 0$ and $\mu\ \mathbb{1}_{\langle 0,1\rangle} = \mu\ \mathbb{1}_{\langle 1,0\rangle} = 1/2$. One can check that μ ensures $[\![c]\!]\ m \sim_{\neg\Phi} [\![c]\!]\ m'$ for all m and m'. Thus, the "obvious" rule

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

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 neither of the relations Φ and $\neg\Phi$ are enough to tell apart the two final distributions.

4.2 Observational equivalence

Observational equivalence is derived as an instance of relational Hoare judgments in which pre and postconditions are restricted to relations based on equality over a subset of variables. Given a set X of variables, we define $=_X$ as

$$m_1 =_X m_2 \overset{\text{def}}{=} \forall\, x \in X.\ m_1\ x = m_2\ x$$

Then, the observational equivalence of G_1 and G_2 w.r.t. an input set I and an output set O is defined as

$$\vDash G_1 \simeq^I_O G_2 \overset{\text{def}}{=} \vDash G_1 \sim G_2 : =_I \Rightarrow =_O$$

All derived rules for pRHL can be specialized to the case of observational equivalence. For example, we have

$$\frac{\vDash e_1 \simeq^I e_2 \qquad \vDash E_1, c_1 \simeq^I_O E_2, c_2 \qquad \vDash E_1, c_1' \simeq^I_O E_2, c_2'}{\vDash E_1, \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \simeq^I_O E_2, \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2'}$$

where $\vDash e_1 \simeq^I e_2$ iff for every memories m_1 and m_2, $m_1 =_I m_2$ implies $[\![e_1]\!]\ m_1 = [\![e_2]\!]\ m_2$.

To support automation, CertiCrypt implements a calculus of variable dependencies and provides two functions, eqobsIn and eqobsOut, that given a command c and a set O (respectively I) of output (input) variables compute a set I (O) of input (output) variables such that $\vDash E_1, c \simeq^I_O E_2, c$. CertiCrypt also provides tactics for two variants of observational equivalence that are widely used in game-based proofs, namely

$$\begin{array}{rcl} \vDash_\varphi G_1 \simeq^I_O G_2 & \overset{\text{def}}{=} & \vDash G_1 \sim G_2 : =_I \wedge\, \varphi \Rightarrow =_O \wedge\, \varphi \\ \vDash_{\Psi,\Phi} G_1 \simeq^I_O G_2 & \overset{\text{def}}{=} & \vDash G_1 \sim G_2 : =_I \wedge\, \Psi \Rightarrow =_O \wedge\, \Phi \end{array}$$

These tactics use a (sound but incomplete) weakest precondition calculus for relational judgments.

5. Proof methods for bridging steps

CertiCrypt provides a powerful set of tactics and algebraic equivalences to automate bridging steps. Most tactics rely on an implementation of a certified optimizer for pWHILE, with the exception of *lazy sampling* which has an ad hoc implementation. Algebraic equivalences are provided as lemmas that follow from algebraic properties of the interpretation of language constructs.

5.1 Certified program transformations

CertiCrypt provides automated support for transformations that consist in applying compiler optimizations. More precisely, it supports transformations based on dependencies and dataflow analyses; we briefly discuss them below. Additionally, CertiCrypt provides support for inlining procedure calls and performing register allocation (not discussed here).

Transformations based on dependencies The functions eqobsIn and eqobsOut presented in Sec. **??**, provide the foundations to support transformations such as dead code elimination, code motion, and common context elimination.

First, CertiCrypt features a function context that strips off two commands c and c' 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

$$\frac{\mathsf{context}(I, c_1, c_2, O) = (I', c_1', c_2', O') \quad \vDash E_1, c_1' \simeq_{O'}^{I'} E_2, c_2'}{\vDash E_1, c_1 \simeq_O^I E_2, c_2}$$

Using the same idea, CertiCrypt provides algorithms for removing only a common prefix (`eqobs_hd`) or suffix (`eqobs_tl`).

Second, CertiCrypt provides a tactic that given two commands repeatedly tries to hoist their common instructions to obtain a maximal common prefix[5], which can then be eliminated using the previous rule. Its correctness is based on the rule

$$\frac{\vDash E, c_1 \simeq_{O_1}^{I_1} E, c_1 \quad \vDash E, c_2 \simeq_{O_2}^{I_2} E, c_2 \quad \mathsf{modify}(E, c_i, O_i) \qquad O_1 \cap O_2 = \emptyset \quad I_1 \cap O_2 = \emptyset \quad I_2 \cap O_1 = \emptyset}{\vDash E, c_1;\ c_2 \sim E, c_2;\ c_1 : = \Rightarrow =}$$

where $\mathsf{modify}(E, c, X)$ is a semantic predicate expressing that only variables in X are modified by the command c in the environment E. This is formally expressed by $\forall\ m.\ \mathsf{range}\ (\lambda m'.\ m =_{\mathcal{V}\setminus X} m')\ (\llbracket E, c \rrbracket\ m)$ which ensures that reachable final memories are equal to the initial one except maybe on variables in X. The tactic `swap` is based on the rule above and on an algorithm that over-approximates the set of modified variables.

Third, CertiCrypt allows performing dead code elimination relative to a set O of output variables (`deadcode`). The algorithm 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 (replacing if true then c_1 else c_2 by c_1), branch coalescing (replacing if e then c else c by c), and self-assignment elimination. Its correctness relies on the rule

$$\frac{\mathsf{modify}(E_1, c, X) \quad \mathsf{Lossless}(E_1, c) \quad \mathsf{fv}(\varphi) \cap X = \emptyset}{\vDash E_1, c \sim E_2, \mathsf{nil} : \varphi \Rightarrow \varphi}$$

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 transforming expressions in the language into their optimized versions (using the result of the analysis), CertiCrypt automatically constructs the certified optimization function $\mathsf{optim} : \mathcal{C} \to D \to \mathcal{C} \times D$. When given a command c and an element $\delta \in D$, this function transforms c into its optimized version c' assuming the validity of δ. In addition, it returns an abstract postcondition $\delta' \in D$ which is valid after executing c (or c'). We use these abstract postconditions to state the correctness of the optimization, and to apply the optimization recursively.

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

The correctness of optim is proved using a mixture of the techniques of [9] and [10, 24]: 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 runtime memory m. Then, correctness is expressed in terms of a pRHL judgment:

$$\textbf{let } (c', \delta') := \mathsf{optim}(c, \delta) \textbf{ in} \vDash E, c \sim E, c' : \asymp_\delta \Rightarrow \asymp_{\delta'}$$

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 [R-Comp]

$$\frac{\forall m_1\, m_2.\ m_1\ \Psi\ m_2 \Rightarrow \mathsf{Valid}(\delta, m_1) \qquad \mathsf{optim}(c_1, \delta) = (c_1', \delta') \quad \vDash E_1, c_1' \sim E_2, c_2 : \Psi \Rightarrow \Phi}{\vDash E_1, c_1 \sim E_2, c_2 : \Psi \Rightarrow \Phi}\text{[R-Opt]}$$

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

5.2 Lazy sampling

It is sometimes convenient to defer random choices in games until they are actually needed, or conversely, to make random choices as early as possible. The lazy sampling technique, allows to delay the random sampling of a value until the point in the program where it is first used. Conversely, eager sampling allows to choose a random value, which would be otherwise sampled later, at the beginning of a game. These techniques are presented in [8], where the authors discuss some of its subtleties. In this section we present a syntax-oriented criterion for the correctness of lazy sampling that can be applied provided the sampling is adequately guarded. Here by *context* we mean a program context with multiple holes that may appear either in the main program or any of the procedures in the environment.

Lemma 1 (Lazy/eager sampling)**.** *Let $C[\cdot]$ be a context, c_1 and c_2 commands, e a boolean expression, d a distribution expression, and z a variable, such that $C[\cdot]$ does not modify $\mathsf{fv}(e) \cup \mathsf{fv}(d)$ and does not use z. Assume*

1. $\vDash z \stackrel{\$}{\leftarrow} d; c_1;$ if e then $z \stackrel{\$}{\leftarrow} d \sim z \stackrel{\$}{\leftarrow} d; c_1 : (= \wedge\ e\langle 1\rangle) \Rightarrow =$
2. $\vDash c_2 \sim c_2 : (= \wedge \neg e\langle 1\rangle) \Rightarrow (= \wedge \neg e\langle 1\rangle)$

Let $c =$ if e then $z \stackrel{\$}{\leftarrow} d; c_1$ else c_2 and $c' =$ if e then c_1 else c_2. Then, $\vDash C[c];$ if e then $z \stackrel{\$}{\leftarrow} d \sim z \stackrel{\$}{\leftarrow} d; C[c'] : (= \wedge\ e\langle 1\rangle) \Rightarrow =$

Intuitively, in the above lemma e indicates whether z has not been used in the game since it was last sampled. If it has not been used, then it is perfectly fine to resample it. The first two hypotheses ensure that e has exactly this meaning, c_1 must set it to false if it has used the value sampled in z, and c_2 must not reset e if it is false. The first hypothesis is the one that allows to swap c_1 with $z \stackrel{\$}{\leftarrow} d$, provided the value of z is not used in c_1. Note that, for clarity, we have omitted environments in the above lemma, and so the second hypothesis is not as trivial as it may seem because both programs may have different environments.

Suppose we want to eagerly sample the answer that a random oracle

```
$\mathcal{O}_{\rm l}(x) \stackrel{\text{def}}{=}$ if $x \notin \mathsf{dom}(L)$ then $y \stackrel{\$}{\leftarrow} d;\ L \leftarrow (x, y) :: L;$
          return $L[x]$
```

gives to a particular query x', i.e. we want to transform $\mathcal{O}_{\rm l}$ into

```
$\mathcal{O}_{\rm e}(x) \stackrel{\text{def}}{=}$ if $x \notin \mathsf{dom}(L)$ then
             if $x = x'$ then $y \leftarrow y'$ else $y \stackrel{\$}{\leftarrow} d$;
             $L \leftarrow (x, y) :: L$
          return $L[x]$
```

Define $\mathcal{O}_{\rm l}'(x) \stackrel{\text{def}}{=}$ if $x' \notin \mathsf{dom}(L)$ then $y' \stackrel{\$}{\leftarrow} d; \mathcal{O}_{\rm e}(x)$ else $\mathcal{O}_{\rm l}(x)$, $\mathcal{O}_{\rm e}'(x) \stackrel{\text{def}}{=}$ if $x' \notin \mathsf{dom}(L)$ then $\mathcal{O}_{\rm e}(x)$ else $\mathcal{O}_{\rm l}(x)$, both returning $L[x]$. Oracles $\mathcal{O}_{\rm l}$ and $\mathcal{O}_{\rm e}$ result semantically equivalent to $\mathcal{O}_{\rm l}'$

and $\mathcal{O}'_{\mathrm{e}}$, respectively. Lemma 1 can be applied taking $e = x' \notin \mathsf{dom}(L)$, $z = y'$, and $c_1 = \mathcal{O}_{\mathrm{e}}(x), c_2 = \mathcal{O}_{\mathrm{l}}(x)$ to safely replace oracle $\mathcal{O}_{\mathrm{l}}$ by oracle $\mathcal{O}_{\mathrm{e}}$ in the environment of a program, sampling y' at the beginning. Whenever a bound for the number of queries to a random oracle is known in advance, the above trick can be iteratively applied to completely remove randomness from the oracle code, as it is done in the proof of the Switching Lemma in Sec. 2.2.

5.3 Algebraic equivalences

Bridging steps frequently use algebraic properties of language operators. 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:

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

In the proof of security of OAEP we use optimistic sampling:

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

and incremental sampling modulo a permutation f:

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

We show the usefulness of [R-Rand] by sketching the proof of optimistic sampling, as promised in Section **??**. For readability, we let $e_{|i}$ denote $[\![e]\!]\ m_i$. Define

$$\Psi \overset{\text{def}}{=} z_{|1} = z_{|2} \qquad \Phi \overset{\text{def}}{=} x_{|1} = x_{|2} \wedge y_{|1} = y_{|2} \wedge z_{|1} = z_{|2}$$

Let $\Phi' \overset{\text{def}}{=} \Phi\{x_{|1} \oplus z_{|1}/y_{|1}, y_{|2} \oplus z_{|2}/x_{|2}\}$. Then, by [R-Ass] we have

$$\vDash y \leftarrow x \oplus z \sim x \leftarrow y \oplus z : \Phi' \Rightarrow \Phi$$

Now take $f\ m_1\ m_2\ v \overset{\text{def}}{=} v \oplus z_{|2}$, and apply [R-Rand] and [R-Sub] to obtain $\vDash\ x \overset{\$}{\leftarrow} \{0,1\}^k \sim y \overset{\$}{\leftarrow} \{0,1\}^k : \Psi \Rightarrow \Phi'$. Note that the precondition we obtain after applying [R-Rand] is equivalent to Ψ because f is a bijection on $\{0,1\}^k$, and because $\forall v.\ \Phi'\{v/x_{|1}, v \oplus z_{|2}/y_{|2}\}$ is equivalent to $z_{|1} = z_{|2}$ by algebraic properties of the $\oplus$ operator. We conclude by applying [R-Seq].

6. Proof methods for failure events

A technique used very often to relate two games 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 a given event in two games: one identifies a failure event and argues that both games behave identically until this event occurs. One can then bound the difference in probability of another event by the probability of occurrence of the failure event in either game.

Lemma 2 (Fundamental lemma)**.** *Let G_1 and G_2 be two games, A an event defined on G_1, B an event defined on G_2 and F an event defined in both games. If*

$$\Pr_{G_1}[A \wedge \neg F] = \Pr_{G_2}[B \wedge \neg F]\ \text{, and}$$
$$\Pr_{G_1}[F] \leq \Pr_{G_2}[F]$$

then $|\Pr_{G_1}[A] - \Pr_{G_2}[B]| \leq \Pr_{G_2}[F]$.[6]

In code-based proofs, the failure condition is generally indicated by setting a global flag variable (usually called bad) to true. This specialization allows to define a syntactic criterion for deciding whether two games behave equivalently up to the raise of the failure condition: we say that two games G_1 and G_2 are equal up to bad and note it $\mathsf{uptobad}(G_1, G_2)$ whenever they are syntactically equal up to every point where the bad flag is set to true and they do not reset the bad flag to false afterward. For instance, games $G^{\mathsf{bad}}_{\mathrm{PRP}}$ and $G^{\mathsf{bad}}_{\mathrm{PRF}}$ in Fig. 2 satisfy this condition. We have used this syntactic criterion to implement a specialization of the fundamental lemma for game-based proofs.

[6] The second hypothesis is usually omitted in the literature under the assumption that both games are absolutely terminating. In that case, either G_1 or G_2 will do on the right-hand side.

```
Game EU_FDH :
$L \leftarrow [\,];\ S \leftarrow [\,];$
$(m, x) \leftarrow \mathcal{A}();$
$d \leftarrow H(m)$
```

```
Oracle H(m) :
if $m \notin \mathsf{dom}(L)$ then
  $r \overset{\$}{\leftarrow} \{0,1\}^k;$
  $L \leftarrow (m, r) :: L$
return $L[x]$
```

```
Oracle Sign(m) :
$S \leftarrow m :: S$
$r \leftarrow \mathsf{H}(m);$
return $f^{-1}(r)$
```

Figure 6. Initial game in the proof of FDH unforgeability

Lemma 3 (Syntactic criterion for fundamental lemma)**.**

$$\forall\, G_1\ G_2\ A.\ \mathsf{uptobad}(G_1, G_2) \Rightarrow$$
$$\Pr_{G_1}[A \wedge \neg\mathsf{bad}] = \Pr_{G_2}[A \wedge \neg\mathsf{bad}]$$

The first hypothesis in Lemma 2 may be proved automatically by using this syntactic criterion. To prove the second hypothesis it suffices to show that game G_2 is absolutely terminating, for which we already have implemented a semi-decision procedure (see Sec. 3.3).

7. Case studies

The purpose of this section is to announce the successful completion of two experiments that validate the design and usability of CertiCrypt. A detailed presentation of these works will be given elsewhere.

7.1 Existential unforgeability of FDH

The Full Domain Hash (FDH) scheme is a hash-and-sign signature scheme based on the RSA family of trapdoor permutations, and in which the message is hashed onto the full domain of the RSA function. However, the same construction—and the reduction that proves its security—remains valid for any family of trapdoor permutations. We have proved that FDH is existentially unforgeable under adaptive chosen-message attacks in the random oracle model for the improved bound in [15]. The proof is about 3,500 lines long.

We will consider a generic family of trapdoor permutations f (and their inverses f^{-1}) indexed by the security parameter, and an ideal hash function $H : \{0,1\}^* \to \{0,1\}^k$, which we model as a random oracle. The initial game of the proof is shown in Fig. 6.

Definition 6 (Trapdoor permutation security)**.** *We say that a trapdoor permutation is (t, ϵ)-secure if an inverter running within time $t(k)$, when given a challenge y uniformly drawn from $\{0,1\}^k$ succeeds in finding $f^{-1}(y)$ with probability at most $\epsilon(k)$, i.e. if for any well-formed adversary $\mathcal{B}$ running within time $t(k)$ in G_f we have $\Pr_{G_f}[x = f^{-1}(y)] \leq \epsilon(k)$, where* **Game** $\mathsf{G}_f : y \overset{\$}{\leftarrow} \{0,1\}^k;\ x \leftarrow \mathcal{B}(y)$.

Theorem 1 (FDH exact security)**.** *Assume the underlying trapdoor permutation is (t', ϵ')-secure. Then, a forger that makes at most q_{hash} and q_{sign} queries to the hash and signing oracles respectively and runs within time $t(k) = t'(k) - (q_{\mathsf{hash}}(k) + q_{\mathsf{sign}}(k) + 1)\ O(T_f)$, succeeds in forging a signature for a new message—different from the ones asked to the signing oracle—with probability at most*

$$\epsilon(k) = \frac{q_{\mathsf{sign}}(k)}{(1 - \frac{1}{q_{\mathsf{sign}}(k)+1})^{q_{\mathsf{sign}}(k)+1}}\ \epsilon'(k) \approx \exp(1)\ q_{\mathsf{sign}}(k)\ \epsilon'(k)$$

i.e. for a well-formed adversary $\mathcal{A}$ running within $t(k)$ we have $\Pr_{\mathsf{EU_{FDH}}}[h = f(x)] \leq \epsilon(k)$.

7.2 Semantic security of OAEP

OAEP is a padding scheme whose history perfectly illustrates the difficulty in achieving a correct proof. Indeed, it was initially believed that OAEP was IND-CCA2 secure [7], but it was later discovered it was only IND-CCA1 secure [30], a weaker security notion (where the adversary does not have access to the decryption oracle after receiving the challenge). It is possible to recover IND-CCA2 security by using it together with a suitable encryption scheme, as it is the case for RSA-OAEP [17]. Here we focus on the game-based proof of [8] which shows IND-CPA security of OAEP in the random oracle model.

The definition of OAEP is parametrized by a trapdoor one-way permutation $f : \{0,1\}^k \to \{0,1\}^k$, and two hash functions $G : \{0,1\}^\rho \to \{0,1\}^{k-\rho}$ and $H : \{0,1\}^{k-\rho} \to \{0,1\}^\rho$. OAEP adds randomness into the plaintext m and uses the functions G and H to mask it before applying f to the result, as formalized by the straightforward code for encryption:

$$R \xleftarrow{\$} \{0,1\}^\rho; S \leftarrow G(R) \oplus m; T \leftarrow H(S) \oplus R; \mathsf{return}\ f(S\|T)$$

We have proved that OAEP is IND-CPA secure. The proof is about 3,000 lines long.

Theorem 2 (OAEP semantic security)**.** *For well-formed adversaries $\mathcal{A}$ and $\mathcal{A}'$ making together at most q_G queries to G,*

$$|\Pr_{\mathsf{IND\text{-}CPA_{OAEP}}}[b = b'] - \frac{1}{2}| \leq \Pr_{G_f}[x = f^{-1}(y)] + \frac{q_G}{2^\rho}$$

where $\Pr_{G_f}[x = f^{-1}(y)]$ is the probability of an adversary inverting f on a random element of its codomain, as in Definition 6.

Our sequence of games differs from [8]: in their initial transitions, Bellare and Rogaway use the fundamental lemma, whereas we use lazy sampling. As a result, our bound for OAEP is tighter than the bound published in [8], which also involves the number q_H of calls to H:

$$|\Pr_{\mathsf{IND\text{-}CPA_{OAEP}}}[b = b'] - \frac{1}{2}| \leq \Pr_{G_f}[x = f^{-1}(y)] + \frac{2q_G}{2^\rho} + \frac{q_H}{2^{k-\rho}}$$

We consider that our proof of OAEP is highly emblematic, because of its complexity and its history. In retrospect, the bound we prove, which is independent of q_H, shows that formalizing proofs sometimes leads to improvements on previous results (to the best of our knowledge). However, cryptographers are really interested in a proof of IND-CCA2, and thus our next objective is to machine-check the results of [17].

8. 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 [25]. Traditionally, protocols have been verified in a symbolic model, for which effective decision procedures exist under suitable hypotheses. Although the symbolic model assumes perfect cryptography, soundness results such as [1] relate the symbolic model with the computational model, provided the cryptographic primitives are secure. It is possible to combine symbolic methods and soundness proofs to achieve guarantees in the computational model, as done e.g. in [5, 32]. One drawback of this approach is that the security proof relies on intricate soundness proofs. Besides, it is not clear whether computational soundness results will always exist to allow factoring verification through symbolic methods. Consequently, some authors attempt to provide guarantees directly at the computational level [11, 23, 28].

In contrast, the formal verification of cryptographic functionalities is an emerging trend. An early work of Barthe, Cederquist and Tarento [6] 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 for cryptography. Corin and den Hartog [14] also prove ElGamal semantic security, using a probabilistic (non-relational) Hoare logic. However, their formalism is not sufficiently powerful to express precisely the security goals: notions such as well-formed and effective adversary are not modeled.

Blanchet and Pointcheval [12] were among the first to use verification tools to carry out game-based proofs of cryptographic schemes. They used CryptoVerif to prove exact security of the FDH signature scheme, for a bound weaker than the one given in Section 7.1. More recently, Courant et al. [16] have developed a form of strongest postcondition calculus that can establish automatically asymptotic security (IND-CPA and IND-CCA2) of schemes that use one-way functions and random oracles. They show soundness and provide a prototype implementation that covers many examples of the literature, including OAEP+. We believe the two approaches are complementary to ours: compiling CryptoVerif sequences of games and embedding the type system of [16] in CertiCrypt, are interesting research directions.

In parallel, several authors have initiated formalizations of game-based proofs in proof assistants, and shown the security of basic examples. Nowak [26] gives a game-based proof of ElGamal semantic security in Coq. Nowak uses a shallow embedding to model games; as a result, its 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. [2] formalize a game-based proof of the 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. All in all, both 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. It is not possible to report on these developments here, so we limit ourselves to singling out Paulin's formalization of the measure monad, which we use extensively in our work, and the work of Hurd et al. [20], who developed a mechanized theory in the HOL theorem prover for reasoning about pGCL programs, a probabilistic extension of Dijkstra's guarded command language.

9. Conclusion

Summary and perspectives 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 is 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 provided by CertiCrypt. Thus, CertiCrypt only provides a first step towards the completion of Halevi's programme, 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.

Future work Numerous research directions remain to be explored. Our first objective is to strengthen our result for OAEP. It would also be useful to formalize cryptographic meta-results such as the equivalence between IND-CPA and IND-CCA2 under plaintext awareness. Another objective would be to formalize proofs of computational soundness of the symbolic model, see e.g. [1] and proofs of automated methods for proving security of primitives and protocols, see e.g. [16, 23]. Finally, it would also be worthwhile to explore applications of CertiCrypt outside cryptography, in particular to randomized algorithms and complexity.

Acknowledgments

We would like to acknowledge Romain Janvier and Federico Olmedo for their enthusiastic participation in the project, Christine Paulin for the support provided in using her library, and the anonymous reviewers for their insightful comments.

References

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

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

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 91–102. ACM Press, 2006.

[4] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 2008.

[5] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 370–379. ACM Press, 2006.

[6] G. Barthe, J. Cederquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In *2nd International Joint Conference on Automated Reasoning*, pages 385–399. Springer-Verlag, 2004.

[7] M. Bellare and P. Rogaway. Optimal asymmetric encryption – How to encrypt with RSA. In *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995.

[8] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT'06*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, 2006.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31th ACM Symposium on Principles of Programming Languages*, pages 14–25. ACM Press, 2004.

[10] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *International Workshop on Types for Proofs and Programs*, volume 3839 of *LNCS*, pages 66–81. Springer-Verlag, 2006.

[11] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.

[12] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO'06*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer-Verlag, 2006.

[13] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[14] R. Corin and J. den Hartog. A probabilistic Hoare-style logic for game-based cryptographic proofs. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, volume 4052 of *LNCS*, pages 252–263, 2006.

[15] J.-S. Coron. On the exact security of Full Domain Hash. In *Advances in Cryptology*, volume 1880 of *Lecture Notes in Computer Science*, pages 229–235. Springer-Verlag, 2000.

[16] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption in the random oracle model. In *Computer and Communications Security*. ACM Press, 2008.

[17] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, 2004.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

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

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

[21] B. Jonsson, K. G. Larsen, and W. Yi. Probabilistic extensions of process algebras. In *Handbook of Process Algebra*, pages 685–711. Elsevier, 2001.

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

[23] P. Laud. Semantics and program analysis of computationally secure information flow. In *European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.

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

[25] C. Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003.

[26] D. Nowak. A framework for game-based security proofs. In *Information and Communications Security*, volume 4861, pages 319–333. Springer-Verlag, 2007.

[27] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 154–165. ACM Press, 2002.

[28] A. Roy, A. Datta, A. Derek, and J. C. Mitchell. Inductive proofs of computational secrecy. In *European Symposium On Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 2007.

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

[30] V. Shoup. OAEP reconsidered. In *Advances in Cryptology – CRYPTO'01*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259. Springer-Verlag, 2001.

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

[32] C. Sprenger and D. Basin. Cryptographically-sound protocol-model abstractions. In *Proceedings of CSF'08*, pages 115–129. IEEE Computer Society, 2008.

[33] J. Stern. Why provable security matters? In *Advances in Cryptology – EUROCRYPT'03*, volume 2656 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[34] The Coq development team. The Coq Proof Assistant Reference Manual v8.1, 2006. Available at `http://coq.inria.fr`.