Programming Language Techniques for Cryptographic Proofs

Gilles Barthe, Benjamin Grégoire, Santiago Zanella-Béguelin

Programming Language Techniques for Cryptographic Proofs*

Gilles Barthe[1], Benjamin Grégoire[2], and Santiago Zanella Béguelin[1]

[1] IMDEA Software, Madrid, Spain, {Gilles.Barthe,Santiago.Zanella}@imdea.org
[2] INRIA Sophia Antipolis - Méditerranée, France, Benjamin.Gregoire@inria.fr

Abstract. CertiCrypt is a general framework to certify the security of cryptographic primitives in the Coq proof assistant. CertiCrypt adopts the code-based paradigm, in which the statement of security, and the hypotheses under which it is proved, are expressed using probabilistic programs. It provides a set of programming language tools (observational equivalence, relational Hoare logic, semantics-preserving program transformations) to assist in constructing proofs. Earlier publications of CertiCrypt provide an overview of its architecture and main components, and describe its application to signature and encryption schemes. This paper describes programming language techniques that arise specifically in cryptographic proofs. The techniques have been developed to complete a formal proof of IND-CCA security of the OAEP padding scheme. In this paper, we illustrate their usefulness for showing the PRP/PRF Switching Lemma, a fundamental cryptographic result that bounds the probability of an adversary to distinguish a family of pseudorandom functions from a family of pseudorandom permutations.

1 Introduction

The goal of provable security [**?**] is to provide a rigorous analysis of cryptographic schemes in the form of mathematical proofs. 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 the security of a cryptographic scheme as a program verification problem: concretely, this is achieved by formulating security goals and hypotheses in terms of the semantics of probabilistic programs, and by defining the adversarial model in terms of

* This work was partially supported by French ANR SESUR-012, SCALP, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS, and Microsoft Research-INRIA Joint Centre.

a class of programs, e.g. probabilistic polynomial-time programs. The code-based approach leads to statements that are unambiguous and amenable to formalization. Yet, standard methods to verify programs (e.g. in terms of program logics) are ineffective to address the kind of verification goal that arises from cryptographic statements. The game-based approach [**?**,**?**] is an alternative to standard program verification methods that establishes the verification goal through successive transformations of the program and the goal. In a nutshell, the game-based approach proceeds by performing 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 $h_1 \circ \cdots \circ h_n(\Pr_{G_n}[A_n]) \leq p$.

CertiCrypt [**?**] is a fully machine-checked framework built on top of the Coq proof assistant [7] to help constructing and verifying game-based cryptographic proofs. An ancillary goal of CertiCrypt is to isolate and formalize precisely the reasoning principles that underlie game-based proofs and to automate their application. While many proof steps use standard reasoning principles based on observational equivalence and semantics-preserving program transformations, some essential techniques arise only in cryptographic proofs. The goal of this article is to present two such techniques and to illustrate their usefulness. The first technique is based on *failure events* and allows to formalize non-semantics-preserving transformations. It applies to transitions of the form $G, A \rightarrow^h G', A$, where G and G' behave identically unless a certain failure event bad occurs, and thus $h(p) = p + \Pr_G[\mathsf{bad}]$ (i.e. $\Pr_G[A] \leq \Pr_{G'}[A] + \Pr_G[\mathsf{bad}]$). The second technique uses interprocedural code motion to place upfront random choices made in games or, dually, to defer them until later in the game. These transformations, called *eager* and *lazy sampling* respectively, are widely used in proofs in the Random Oracle Model [**?**]. Both techniques are discussed in [**?**], but the present paper considerably extends their scope. Concretely, we complement the Fundamental Lemma of [**?**], that bounds the difference in the probability of events in two games that behave identically until failure, with a Failure Event Lemma that allows to bound the probability of failure. In order to establish the Failure Event Lemma, we introduce a notion of judgment that upper bounds the probability of an event after executing a program in terms of the probability of an event prior to execution. Moreover, we considerably clarify the eager/lazy sampling methodology using a logic for swapping program statements. The logic overcomes some limitations that hamper the application of the technique as it was described in [**?**].

Both extensions were required to prove the IND-CCA security of the OAEP padding scheme [**?**], for which the results of [**?**] did not suffice. As the complexity of this proof would prevent us from illustrating the techniques we used, we consider instead a simpler motivating example, namely the PRP/PRF Switching Lemma, a fundamental cryptographic result that bounds the probability of an

adversary to distinguish a family of pseudorandom functions from a family of pseudorandom permutations.

2 Motivating Example: the PRP/PRF Switching Lemma

Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that play a central role in the design of symmetric-key systems. Although the most natural assumption to make about a blockcipher is that it behaves as a pseudorandom permutation, most commonly the security of a system based on a blockcipher is analyzed by replacing the blockcipher with a perfectly random function. The PRP/PRF Switching Lemma [**?**,**?**] fills the gap: given a bound for the security of a blockcipher as a pseudorandom permutation, it gives a bound for its security as a pseudorandom function.

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} :
$\boldsymbol{L} \leftarrow [\,];\ b \leftarrow \mathcal{A}()$
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} :
$\boldsymbol{L} \leftarrow [\,];\ b \leftarrow \mathcal{A}()$
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 $\mathcal{O}$ implements an injective—and therefore bijective—function. Formally, the instruction $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$ may be implemented by repeatedly sampling a bitstring until the result does not belong to $\mathsf{ran}(\boldsymbol{L})$.

Lemma 1 (PRP/PRF switching lemma). *Let $\mathcal{A}$ be an adversary with black-box access to an oracle $\mathcal{O}$ implementing either a random permutation on $\{0,1\}^\ell$ as in game G_{RP} or a random function from $\{0,1\}^\ell$ to $\{0,1\}^\ell$ as in game G_{RF}. Suppose, in addition, that $\mathcal{A}$ makes at most $q > 0$ queries to its oracle. Then,*

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

The standard *proof* of the PRP/PRF Switching Lemma is due to Impagliazzo and Rudich [**?**, Theorem 5.1]. Bellare and Rogaway [**?**] report a subtle error in the reasoning of [**?**] and provide a counterexample. They give a game-based proof of the PRP/PRF Switching Lemma under the additional assumption that the adversary never asks the same oracle query twice. Their proof uses the Fundamental Lemma (Sec. 4) 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 [**?**, Section 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 who calls the adversary to get a query and who forwards it to the oracle. There is probably nothing wrong with this formulation, but we feel that it imposes unnecessary restrictions on the form of the adversary and hinders understanding.

The PRP/PRF Switching Lemma has been formalized previously. Affeldt, Tanaka and Marti [**?**] present a formalization of a game-based proof of the PRP/PRF Switching Lemma in Coq. What they prove in reality is a simplified variant that only holds for non-adaptive and deterministic adversaries. They formalize adversaries as purely deterministic mathematical functions that take a natural number and return an element in the domain of its oracle (a query). This implies that the queries the adversary makes do not depend on the responses to previous queries or on any random choices. The authors also report on a formalization in CertiCrypt [**?**]; Sec. 6 presents two significantly simplified proofs that use the programming language techniques developed in this paper.

3 A Language for Cryptographic Games

Games are formalized as programs in pWhile, a probabilistic imperative language with procedure calls. For the purpose of this exposition, we restrict random sampling to uniform distributions over a set $\mathcal{T}$ of base types. We let $\mathcal{V}$ be a set of variable identifiers[3] and $\mathcal{P}$ be a set of procedure identifiers. The set of commands is defined inductively by the clauses:

$$
\begin{array}{lll}
\mathcal{I} ::= \mathcal{V} \leftarrow \mathcal{E} & & \text{assignment} \\
\quad \mid \mathcal{V} \xleftarrow{\$} \mathcal{T} & & \text{random sampling} \\
\quad \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & & \text{conditional} \\
\quad \mid \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & & \text{while loop} \\
\quad \mid \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & & \text{procedure call} \\
\mathcal{C} ::= \mathsf{nil} & & \text{nop} \\
\quad \mid \mathcal{I};\ \mathcal{C} & & \text{sequence}
\end{array}
$$

[3] Variables are partitioned into local and global variables. We will sometimes ignore this distinction to avoid cluttering the presentation. We use a **bold face** to typeset global variables in games.

where we assume that the set $\mathcal{E}$ of expressions has been previously defined. Moreover, we assume given a function $\mathsf{fv} : \mathcal{E} \to \mathsf{list}\ \mathcal{V}$ that computes the set of free variables of an expression.

A program (or game) 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[4]

$$\mathsf{declaration} \stackrel{\text{def}}{=} \{\mathsf{params} : \mathsf{list}\ \mathcal{V};\ \mathsf{body} : \mathcal{C};\ \mathsf{re} : \mathcal{E}\}$$

The language is deeply embedded in Coq, so one can define programs with holes by parametrizing them by program variables of type $\mathcal{C}$ (holes may appear in the main command or in the body of procedures in the environment). In particular, adversaries may be represented as procedures whose whole body is modeled as a variable of type $\mathcal{C}$.

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 the 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 $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$, where $\mathcal{F}$ is the set of procedures that the adversary may call, $\mathcal{RW}$ the set of variables that it may read and write, and $\mathcal{RO}$ the set of variables that it may only read. We say that an adversary $\mathcal{A}$ with interface $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$ is well-formed in an environment E if the judgment $\vdash_{\text{wf}} \mathcal{A}$ can be derived from the rules in Fig. 1. For convenience, we allow adversaries to call procedures outside $\mathcal{F}$, but these procedures must themselves respect the same interface. Note that the rules are generic, only making sure that the adversary makes a correct usage of variables and procedure calls. In particular, they do not aim to impose restrictions that are specific to a particular game, such as the number of calls that an adversary can make to an oracle, or the conditions under which an oracle can be called, or the computational complexity of the adversary. These additional assumptions on adversaries may be specified independently (e.g. by instrumenting games).

3.1 Semantics

Programs in pWhile are given a continuation-passing style semantics using the measure monad M, whose type constructor is defined as

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

[4] The formalization uses single-exit procedures. For readability, all examples are presented in a more traditional style, and use an explicit return statement.

$$I \vdash_{\mathrm{wf}} \mathsf{nil} : I \qquad \frac{I \vdash_{\mathrm{wf}} i : I' \quad I' \vdash_{\mathrm{wf}} c : O}{I \vdash_{\mathrm{wf}} i ; c : O} \qquad \frac{\mathsf{writable}(x) \quad \mathsf{fv}(e) \subseteq I}{I \vdash_{\mathrm{wf}} x \leftarrow e : I \cup \{x\}}$$

$$\frac{\mathsf{writable}(x)}{I \vdash_{\mathrm{wf}} x \xleftarrow{\$} T : I \cup \{x\}} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash_{\mathrm{wf}} c_i : O_i,\ i = 1, 2}{I \vdash_{\mathrm{wf}} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : O_1 \cap O_2} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash_{\mathrm{wf}} c : I}{I \vdash_{\mathrm{wf}} \mathsf{while}\ e\ \mathsf{do}\ c : I}$$

$$\frac{\mathsf{fv}(\boldsymbol{e}) \subseteq I \quad \mathsf{writable}(x) \quad p \in \mathcal{F}}{I \vdash_{\mathrm{wf}} x \leftarrow p(\boldsymbol{e}) : I \cup \{x\}} \qquad \frac{\mathsf{fv}(\boldsymbol{e}) \subseteq I \quad \mathsf{writable}(x) \quad p \notin \mathcal{F} \quad \vdash_{\mathrm{wf}} p}{I \vdash_{\mathrm{wf}} x \leftarrow p(\boldsymbol{e}) : I \cup \{x\}}$$

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

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

Fig. 1. Rules for well-formedness of an adversary against interface $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$. A judgment of the form $I \vdash_{\mathrm{wf}} c : O$ can be read 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 $I \vdash_{\mathrm{wf}} c : O$, then $I \subseteq O$.

The operators unit and bind of the monad are defined as follows:

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

Expressions are deterministic; an expression e of type T is interpreted by a function $[\![e]\!] : \mathcal{M} \to [\![T]\!]$, where $[\![T]\!]$ is the interpretation of T. The denotation of a game G is given by the function $[\![G]\!] : \mathcal{M} \to M(\mathcal{M})$, that relates an initial memory $m \in \mathcal{M}$ to the expectation operator of the (sub) probability distribution of final memories resulting from its execution. This allows to define the probability of an event A in a game G and an initial memory m in terms of its characteristic function $\mathbb{1}_A$, as $\Pr_{G,m}[A] \stackrel{\mathrm{def}}{=} [\![G]\!]\ m\ \mathbb{1}_A$. Thus, in this monadic semantics a probabilistic event is nothing but a continuation. We refer the interested reader to [**?**] for a more detailed account of the semantics.

3.2 Notations

Let X be a set of variables, $m_1, m_2 \in \mathcal{M}$ and $f_1, f_2 : \mathcal{M} \to [0, 1]$, we define

$$\begin{array}{rcl} m_1 =_X m_2 & \stackrel{\mathrm{def}}{=} & \forall x \in X.\ m_1(x) = m_2(x) \\ f =_X g & \stackrel{\mathrm{def}}{=} & \forall m_1\ m_2.\ m_1 =_X m_2 \implies f(m_1) = g(m_2) \end{array}$$

Let P be a predicate on X and let $\mu : M(X)$ be a distribution over X, then every value $x \in X$ with positive probability w.r.t μ satisfies P when

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

Our logics often use modify clauses; the statement $\mathsf{modify}(E, c, X)$ expresses that only variables in X are modified by the command c in environment E. Semantically,

$$\mathsf{modify}(E, c, X) \stackrel{\mathrm{def}}{=} \forall m.\ \mathsf{range}\ (\lambda m'.\ m =_{\mathcal{V} \setminus X} m')\ ([\![E, c]\!]\ m)$$

Finally, for a Boolean-valued expression e, we let $\langle e \rangle_i$ denote the binary relation $\lambda m_1\ m_2.\ [\![e]\!]\ m_i = \mathsf{true}$.

3.3 Observational Equivalence and Relational Hoare Logic

CertiCrypt formalizes an equational theory of observational equivalence that allows to prove that program fragments are semantically equivalent. We say that two games G_1, G_2 are observationally equivalent w.r.t. an input set of variables I and an output set of variables O, when

$$\begin{array}{l}\vdash G_1 \simeq_O^I G_2 \stackrel{\text{def}}{=} \\ \quad \forall m_1\ m_2.\ m_1 =_I m_2 \implies \forall f_1\ f_2.\ f_1 =_O f_2 \implies [\![G_1]\!]\ m_1\ f_1 = [\![G_2]\!]\ m_2\ f_2\end{array}$$

Observational equivalence provides a useful tool to reason about probabilities. Assume that A is an event (i.e. a map from memories to Booleans) whose value only depends on a set of variables O, i.e. $\mathbb{1}_A =_O \mathbb{1}_A$. If $\vdash G_1 \simeq_O^I G_2$, then for every pair of memories m_1 and m_2 such that $m_1 =_I m_2$, we have

$$\Pr_{G_1,m_1}[A] = \Pr_{G_2,m_2}[A] \tag{2}$$

When $I = O = \mathcal{V}$, $\vdash G_1 \simeq_O^I G_2$ boils down to the semantic equivalence of both games, which we write as $G_1 \equiv G_2$.

Observational equivalence, however, is not enough to justify some context-dependent program transformations. In order to prove the correctness of such transformations, we need to generalize observational equivalence to a full-fledged Relational Hoare Logic that considers arbitrary binary relations on memories (and not just equality on a subset of variables). This logic deals with judgments of the form

$$\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=} \forall m_1\ m_2.\ m_1\ \Psi\ m_2 \implies [\![G_1]\!]\ m_1 \sim_\Phi [\![G_2]\!]\ m_2$$

where the relation $\sim_\Phi$ is a lifting of relation Φ to distributions, defined as:

$$\mu_1 \sim_\Phi \mu_2 \stackrel{\text{def}}{=} \exists \mu.\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{range}\ \Phi\ \mu$$

where π_1 and π_2 are the projections that map a distribution over $A \times B$ to a distribution over A and B, respectively:

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

For an overview of the rules of the relational logic we refer the reader to [**?**].

4 Failure Events

One common technique to justify a *lossy* transformation $G, A \to G', A$, where $\Pr_G[A] \neq \Pr_{G'}[A]$ is to annotate both games with a fresh Boolean flag bad that

is set whenever the code of the games differ. Consider for example the following two program snippets and their annotated versions:

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

If we ignore the variable bad, s and s_{bad}, and s' and s'_{bad}, respectively, are observationally equivalent. Moreover, s_{bad} and s'_{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 s' instead can be bounded by the probability of bad being set in either s_{bad} or s'_{bad}, provided variable bad is initially set to false.

Lemma 2 (Fundamental Lemma). *For any pair of games G, G' and events A, A' and F:*

$$\Pr_G[A \wedge \neg F] = \Pr_{G'}[A' \wedge \neg F] \implies |\Pr_G[A] - \Pr_{G'}[A']| \leq \max(\Pr_G[F], \Pr_{G'}[F])$$

To apply the Fundamental Lemma, we developed a syntactic criterion to discharge its hypothesis for the particular case where $A = A'$ and $F = \mathsf{bad}$. 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, then $\Pr_G[\mathsf{bad}] = \Pr_{G'}[\mathsf{bad}]$, and that if, for instance, game G' terminates with probability 1, it must be the case that $\Pr_G[\mathsf{bad}] \leq \Pr_{G'}[\mathsf{bad}]$.

4.1 A Logic for Bounding the Probability of Failure

Many steps in game-based proofs require to provide an upper bound for the expectation 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 $\mathbb{1}_{\mathsf{bad}}$). An upper bound for a function $(\lambda m.[\![c]\!]\ m\ g)$ is a function f such that $\forall m.[\![c]\!]\ m\ g \leq f\ m$. We note this as a triple $[\![c]\!]g \preceq f$,

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

Figure 2 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 that she may modify 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 Hoare Logic. However, there are some subtle differences. For example, the premises of the rules for branching statements do not consider guards. The rule

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

$$\vdash [\![\mathsf{nil}]\!] f \preceq f \quad \frac{f = \lambda m.\ g(m\{x := [\![e]\!]\ m\})}{\vdash [\![x \leftarrow e]\!] g \preceq f} \quad \frac{f = \lambda m. |[\![T]\!]|^{-1} \sum_{t \in [\![T]\!]}\ g(m\{x := t\})}{\vdash [\![x \xleftarrow{\$} T]\!] g \preceq f}$$

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

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

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

$$\frac{f =_I f \quad \vdash c \simeq_O^I c' \quad g =_O g \quad \vdash [\![c']\!] g \preceq f}{\vdash [\![c]\!] g \preceq f}$$

Fig. 2. Selected rules of a logic for bounding events.

where $f_{|e}$ is defined as $(\lambda m.\ \text{if } [\![e]\!]m \text{ then } f(m) \text{ else } 0)$ can be derived from the rule for conditionals statements by two simple applications of the "rule of consequence". Moreover, the rule for conditional statements 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. More generally, It seems possible to make the logic complete, at the cost of considering more complex statements with preconditions on memories.

Discussion. The differences between the above triples and those of Hoare logic are inherent to their definition, which is tailored to provide an upper bound for the probability of an event after executing a command. Nevertheless, the validity of a Hoare triple $\{A\}c\{B\}$ (in which pre and postconditions are Boolean-valued predicates) is equivalent to the validity of the triple $[\![c]\!] \mathbb{1}_{\neg B} \preceq \mathbb{1}_{\neg A}$.

There exists a dual notion of triple $[\![c]\!] g \succeq f$ whose validity is defined as:

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

This dual notion allows to express termination of a program as $[\![c]\!] \mathbb{1}_{\mathsf{true}} \succeq \mathbb{1}_{\mathsf{true}}$. Moreover, there exists an embedding of Hoare triples, mapping $\{P\}c\{Q\}$ to $[\![c]\!] \mathbb{1}_Q \succeq \mathbb{1}_P$. The 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 not $[\![c]\!] \mathbb{1}_{\mathsf{false}} \succeq \mathbb{1}_{\mathsf{true}}$ because for every $m \in \mathcal{M}$, we have $[\![c]\!]\ m\ \mathbb{1}_{\mathsf{false}} = 0$ and $\mathbb{1}_{\mathsf{true}}(m) = 1$.

4.2 Automation

In most applications of Lemma 2, the failure event 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.

Lemma 3 (Failure Event Lemma). *Consider a game G that gives adversaries access to an oracle $\mathcal{O}$. Let P, F be predicates over memories, and let $h : \mathbb{N} \to [0,1]$ be such that F does not depend on variables that can be written outside $\mathcal{O}$, and for any memory m,*

$$\begin{aligned} P(m) &\implies \mathsf{range}\ (\llbracket \mathcal{O}.\mathit{body} \rrbracket\ m)\ (\lambda m'.\llbracket \mathsf{cntr} \rrbracket\ m < \llbracket \mathsf{cntr} \rrbracket\ m') \\ \neg P(m) &\implies \mathsf{range}\ (\llbracket \mathcal{O}.\mathit{body} \rrbracket\ m)\ (\lambda m'.F\ m' = F\ m \wedge \llbracket \mathsf{cntr} \rrbracket\ m = \llbracket \mathsf{cntr} \rrbracket\ m') \\ \neg F(m) &\implies \Pr_{\mathcal{O}.\mathit{body},m}[F] \leq h(\llbracket \mathsf{cntr} \rrbracket\ m) \end{aligned}$$

Intuitively, P indicates whether a call would increment the counter, failure F only occurs in calls incrementing the counter, and h bounds the probability of failure in a single call.

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

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

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

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

We show that $\llbracket G \rrbracket f \preceq f$ 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 $\neg P(m)$, because we have

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

When $P(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m \geq q$, this is trivial too, because $\mathcal{O}.\mathsf{body}$ increments cntr and the left hand side becomes 0. We are left with the case where $P(m)$ 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

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

Using the rules in Fig. 2, we can then extend this result to adversary calls and to the rest of the game, showing that $[\![G]\!]f \preceq f$.

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

$$\Pr_{G,m}[F \wedge \mathsf{cntr} \leq q] \leq [\![G]\!]\ m\ f \leq f(m) = \sum_{i=0}^{q-1} h(i) \qquad \square$$

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 $F = \mathsf{bad}$ and defining h suitably. In most practical applications (e.g. security of OAEP) h is a constant function; the proof of Lemma 1 given in Sec. 6.2 is an exception for which the full generality of the lemma is needed.

5 Eager and Lazy Sampling

Game-based proofs commonly include bridging steps in which one performs a semantics-preserving reordering of instructions. On most occasions, the reordering is intraprocedural. However, proofs in the random oracle model (see $\mathcal{O}_{\mathrm{lazy}}$ in Fig. 4 for an example of a random oracle) often use interprocedural code motion, in which sampling statements are moved from an oracle to the main command of the game or, conversely, from the main command to an oracle. The first transformation, called eager sampling, is useful for moving random choices upfront: a systematic application of eager sampling allows to transform a probabilistic game G that samples at most 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 might still make their own random choices.[5] The second, dual, transformation, called lazy sampling, is useful to postpone sampling of random values until these values are actually used for the first time.

CertiCrypt features tactics that allow to perform and justify both intra and interprocedural code motion. The tactic for intraprocedural code motion is described in [**?**]. 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, and overcomes many limitations of our earlier lemma reported in [**?**]. A first limitation of our earlier lemma is that it only allowed to swap one random sampling at the time, whereas some applications, including the PRP/PRF Switching Lemma, require swapping a sequence of random samplings. Another limitation of our earlier method is that it could not be used for proving that some queries to a random oracle $\mathcal{O}$ are uniformly distributed and independent from the view of the adversary, as needed in the proof of IND-CCA of OAEP.

[5] Making adversaries deterministic is the goal of the *coin fixing* technique, as described in [**?**]; formalizing this technique is left for future work.

5.1 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

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

In most cases, the logic will be applied with S being a sequence of (guarded) sampling statements; however, the logic does not constrain S, and merely requires that S satisfies three basic properties:

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

for some sets of variables X and I. Some rules of the logic are given in Fig. 3; for the sake of readability, all rules are specialized to $\equiv$, although we formalized more general versions of the rules, e.g. for conditional statements

$$\frac{\begin{array}{c}\vdash E, c_1; S \sim E', S; c_1') : P \wedge \langle e \rangle_1 \Rightarrow Q \qquad \vdash E, c_2; S \sim E', S; c_2' : P \wedge \langle \neg e \rangle_1 \Rightarrow Q \\ P \Rightarrow \langle e \rangle_1 = \langle e' \rangle_2 \qquad \vdash E', S \sim E', S : = \wedge \langle e' \rangle_1 \Rightarrow = \wedge \langle e' \rangle_1\end{array}}{\vdash E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \sim E', (S; \mathsf{if}\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2') : P \Rightarrow Q}$$

which is used in the application considered in the next section.

5.2 Application

Consider the games $\mathsf{G}_{\mathsf{lazy}}$ and $\mathsf{G}_{\mathsf{eager}}$ in Fig. 4. Suppose that our goal is to provide an upper bound of the probability of an event A in game $\mathsf{G}_{\mathsf{lazy}}$ and that the proof proceeds by eagerly sampling the value that the oracle $\mathcal{O}$ returns in response for a particular query $\hat{\boldsymbol{x}}$. Define

$$S_{\hat{\boldsymbol{y}}} \stackrel{\text{def}}{=} \mathsf{if}\ \hat{\boldsymbol{x}} \notin \mathsf{dom}(\boldsymbol{L})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} T\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{L}(\hat{\boldsymbol{x}})$$

and take $I = \{\hat{\boldsymbol{x}}, \boldsymbol{L}\}$ and $X = \{\hat{\boldsymbol{y}}\}$. We have that

$$\vdash E_{\mathsf{lazy}},\ c; S_{\hat{\boldsymbol{y}}} \ \equiv\ E_{\mathsf{eager}},\ S_{\hat{\boldsymbol{y}}}; c \quad \Longrightarrow \quad \Pr_{\mathsf{G}_{\mathsf{lazy}}}[A] = \Pr_{\mathsf{G}_{\mathsf{eager}}}[A]$$

Thus, in order to move from game $\mathsf{G}_{\mathsf{lazy}}$ to game $\mathsf{G}_{\mathsf{eager}}$, it is enough to prove the commutation property on the left of the implication. This, in turn, requires showing that

$$\vdash E_{\mathsf{lazy}}, \mathcal{O}_{\mathsf{lazy}}.\mathsf{body}; S_{\hat{\boldsymbol{y}}} \ \equiv\ E_{\mathsf{eager}}, S_{\hat{\boldsymbol{y}}}; \mathcal{O}_{\mathsf{eager}}.\mathsf{body}$$

which can be achieved by applying the rules of the logic for swapping statements and the relational Hoare logic.

$$\frac{x \notin I \cup X \qquad \mathsf{fv}(e) \cap X = \emptyset}{\vdash E, (x \leftarrow e; S) \;\equiv\; E', (S; x \leftarrow e)} \qquad \frac{x \notin I \cup X}{\vdash E, (x \xleftarrow{\$} T; S) \;\equiv\; E', (S; x \xleftarrow{\$} T)}$$

$$\frac{\vdash E, (c_1; S) \;\equiv\; E', (S; c_1') \quad \vdash E, (c_2; S) \;\equiv\; E', (S; c_2')}{\vdash E, (c_1; c_2; S) \;\equiv\; E', (S; c_1'; c_2')}$$

$$\frac{\vdash E, (c_1; S) \;\equiv\; E', (S; c_1') \qquad \vdash E, (c_2; S) \;\equiv\; E', (S; c_2') \qquad \mathsf{fv}(e) \cup X = \emptyset}{\vdash E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \;\equiv\; E', (S; \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2')}$$

$$\frac{\vdash E, (c; S) \;\equiv\; E', (S; c') \quad \mathsf{fv}(e) \cup X = \emptyset}{\vdash E, (\mathsf{while}\ e\ \mathsf{do}\ c; S) \;\equiv\; E', (S; \mathsf{while}\ e\ \mathsf{do}\ c')}$$

$$\frac{\begin{array}{c}\vdash E, (f_E.\mathsf{body}; S) \;\equiv\; E', (S; f_{E'}.\mathsf{body}) \quad f_E.\mathsf{params} = f_{E'}.\mathsf{params} \quad f_E.\mathsf{re} = f_{E'}.\mathsf{re} \\ \mathsf{fv}(f_E.\mathsf{re}) \cap X = \emptyset \quad x \notin I \cup X \quad \mathsf{fv}(\boldsymbol{e}) \cap X = \emptyset\end{array}}{\vdash E, (x \leftarrow f(\boldsymbol{e}); S) \;\equiv\; E', (S; x \leftarrow f(\boldsymbol{e}))}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{wf}} \mathcal{A} \quad X \cap (\mathcal{RW} \cup \mathcal{RO}) = \emptyset \quad I \cap \mathcal{RW} = \emptyset \quad \forall f \notin \mathcal{F}.\ f_E = f_{E'} \\ \forall f \in \mathcal{F}.\ f_E.\mathsf{params} = f_{E'}.\mathsf{params} \wedge f_E.\mathsf{re} = f_{E'}.\mathsf{re} \ \wedge \\ \vdash E, (f_E.\mathsf{body}; S) \;\equiv\; E', (S; f_{E'}.\mathsf{body})\end{array}}{\vdash E, (x \leftarrow \mathcal{A}(\boldsymbol{e}); S) \;\equiv\; E', (S; x \leftarrow \mathcal{A}(\boldsymbol{e}))}$$

Fig. 3. Selected rules of a logic for swapping statements.

6 Proofs of the PRP/PRF Switching Lemma

We have formalized two proofs of the Switching lemma: both 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 to bound the probability of failure, whereas the second one relies on the Failure Event Lemma.

We begin by introducing in Fig. 5 annotated versions $\mathsf{G}_{\mathsf{RP}}^{\mathsf{bad}}$ and $\mathsf{G}_{\mathsf{RF}}^{\mathsf{bad}}$ of the games G_{RP} and G_{RF} defined in Sec. 2. From Lemma 2, we readily have

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

6.1 A Proof Based on Eager Sampling

We make a first remark: the probability of bad being set in game $\mathsf{G}_{\mathsf{RF}}^{\mathsf{bad}}$ 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 $\mathsf{bad} \implies \mathsf{col}(\boldsymbol{L})$ is an invariant of the game by means of the mechanized relational logic.

Using the logic for swapping statements, we modify the oracle in $\mathsf{G}_{\mathsf{RF}}^{\mathsf{bad}}$ 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. 5. 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.

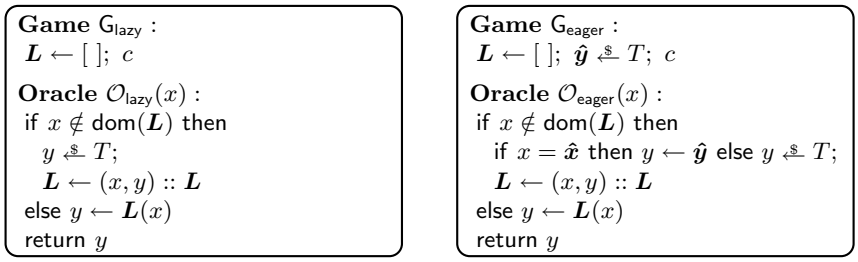
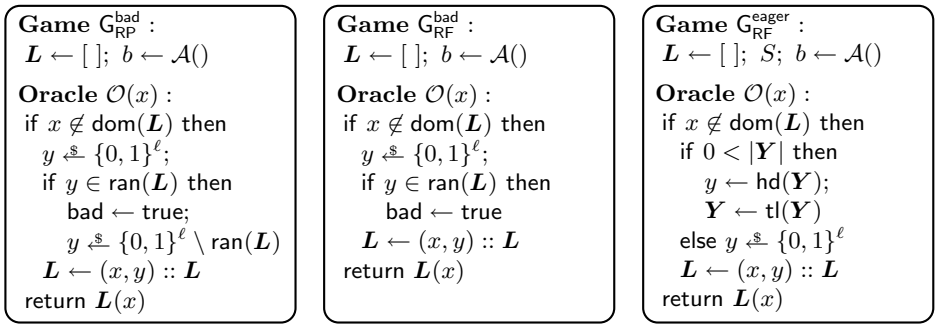


Fig. 4. An example of eager sampling using interprocedural code motion.



$S \stackrel{\text{def}}{=} \boldsymbol{Y} \leftarrow [\,];\ \mathsf{while}\ |\boldsymbol{Y}| < q\ \mathsf{do}\ y \xleftarrow{\$} \{0,1\}^{\ell};\ \boldsymbol{Y} \leftarrow \boldsymbol{Y} +\!\!+ [y]$

Fig. 5. Games used in the proofs of the PRP/PRF Switching Lemma.

By using the rules of the logic for swapping statements, we show that the call $b \leftarrow \mathcal{A}()$ swaps with S. Since the initialization code S terminates and does not modify $\boldsymbol{L}$, we can conclude that

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

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 Eq. (1) holds by analyzing the loop in S. Observe that if there are no collisions in $\boldsymbol{Y}$ in a memory m, the probability of sampling a colliding value in the remaining loop iterations is

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

This is proved by induction on $(q - |\boldsymbol{Y}|)$.

Remark The proof reported in [**?**] uses a similar sequence of games. The sole difference is in the application of eager sampling. Here we do it in one step,

whereas the earlier version uses induction. The new proof is both simpler and shorter (about 400 lines of Coq compared to the 900 lines of the former proof).

6.2 A Proof Based on Bounding Triples

The bound in Eq. 1 follows from a direct application of Lemma 3. It suffices to take $P = x \notin \mathsf{dom}(\boldsymbol{L})$, $F = \mathsf{bad}$, $h(i) = i\,2^{-\ell}$, and $\mathsf{cntr} = |\boldsymbol{L}|$. If bad is initially set to false in memory m, we have

$$\begin{aligned}\Pr_{\mathsf{G}_{\mathsf{RF}}^{\mathsf{bad}},m}[\mathsf{bad}] &= \Pr_{b\leftarrow\mathcal{A}(),m\{\boldsymbol{L}:=[]\}}[\mathsf{bad}] \\ &= \Pr_{b\leftarrow\mathcal{A}(),m\{\boldsymbol{L}:=[]\}}[\mathsf{bad} \wedge |\boldsymbol{L}| \leq q] \leq \sum_{i=0}^{q-1} h(i) = \frac{q(q-1)}{2^{\ell+1}}\end{aligned}$$

The second equation holds because $\mathcal{A}$ does not make more than q queries to oracle $\mathcal{O}$; the last inequality is obtained from Lemma 3. We use the logic in Fig. 2 to bound the probability of bad being set in one call to the oracle by $|\boldsymbol{L}|2^{-\ell}$, as required by the Failure Event Lemma. The resulting proof is considerably shorter compared to the one presented in the previous section (only about 100 lines of Coq).

7 Related Work

We refer to [**?**] for an overview of papers that apply proof assistants to cryptography, and that focus on programming language techniques akin to those described in this paper. The first line of work is concerned with reasoning about probabilistic programs. Several program logics have been studied in the literature, see e.g. [**?**,**?**], and some authors have developed machine-checked frameworks to reason about randomized algorithms. Hurd et al [**?**] report on a formalization of the logic of [**?**], and on several applications. Audebaud and Paulin [**?**] present another framework, which provides the library of probabilities upon which CertiCrypt is built. The second line of work is concerned with certified program transformations. Over the last few years, certified optimizing compilers have become a reality, see e.g. [**?**]. In the course of these efforts, many program transformations have been certified, including lazy code motion [**?**]. There is a similarity between lazy code sampling and rematerialization [**?**]—an optimization that recomputes a value instead of loading it from memory—and it would be interesting to see whether the method developed in this paper could prove useful to build a translation validator for the former.

8 Conclusion

The game-based approach to cryptographic proofs routinely uses a number of unusual programming language techniques. In this paper we report on the certification and automation of two such techniques, namely failure events, and

eager/lazy sampling. Both techniques have been used extensively to successfully provide a machine-checked proof of IND-CCA security of the OAEP padding scheme. Our ultimate goal is to provide a comprehensive coverage of the techniques used by cryptographers, and to turn CertiCrypt into an effective platform for verifying a wide range of cryptographic proofs.

References

1. Stern, J.: Why provable security matters? In: Advances in Cryptology – EUROCRYPT'03. Volume 2656 of Lecture Notes in Computer Science., Springer-Verlag (2003) 449–461
2. Bellare, M., Rogaway, P.: 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., Springer-Verlag (2006) 409–426
3. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
4. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th ACM Symposium on Principles of Programming Languages, ACM Press (2009) 90–101
5. Zanella Béguelin, S., Grégoire, B., Barthe, G., Olmedo, F.: Formally certifying the security of digital signature schemes. In: 30th IEEE Symposium on Security and Privacy, S&P 2009. (2009) 237–250
6. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: 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, Springer-Verlag (2008)
7. The Coq development team: The Coq Proof Assistant Reference Manual v8.2 (2008) [Online]. Available: `http://coq.inria.fr`.
8. Bellare, M., Rogaway, P.: Optimal asymmetric encryption – How to encrypt with RSA. In: Advances in Cryptology – EUROCRYPT'94. Volume 950 of Lecture Notes in Computer Science., Springer-Verlag (1995) 92–111
9. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. Journal of Cryptology **17**(2) (2004) 81–104
10. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In Johnson, D.S., ed.: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 15-17 May 1989, Seattle, Washington, USA, ACM (1989) 44–61
11. Jonsson, B., Larsen, K.G., Yi, W.: Probabilistic extensions of process algebras. In: Handbook of Process Algebra. Elsevier (2001) 685–711
12. Bellare, M., Rogaway, P.: 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., Springer (2006) 409–426

A Proof Sketch of an Application of Lazy Sampling

The following is a proof sketch of the implication:

$$\dfrac{\dfrac{\mathsf{modify}(E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}}, \{\hat{\boldsymbol{y}}\})}{\vdash G_{\mathrm{l}} \simeq^{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}}_{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}} E_{\mathrm{l}}, \boldsymbol{L} \leftarrow [\,]; c; S_{\hat{\boldsymbol{y}}}}\ [3] \quad \dfrac{\vdash E_{\mathrm{l}}, c; S_{\hat{\boldsymbol{y}}} \ \equiv\ E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}}; c}{\vdash E_{\mathrm{l}}, \boldsymbol{L} \leftarrow [\,]; c; S_{\hat{\boldsymbol{y}}} \simeq^{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}}_{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}} G_{\mathrm{e}}}\ [4]}{\vdash G_{\mathrm{l}} \simeq^{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}}_{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}} G_{\mathrm{e}}}\ [2]}{\Pr_{G_{\mathrm{l}}}[A] = \Pr_{G_{\mathrm{e}}}[A]}$$

The proof uses Equation 2 to go back from an equality of probabilities to observational equivalence. Step [2] is justified by the transitivity of the $\simeq$ relation; [3] is justified by dead code elimination, while [4] follows from first applying the rule for sequential statements of the Relational Hoare Logic, and then propagating the initial assignment to $\boldsymbol{L}$ to the conditional in $S_{\hat{\boldsymbol{y}}}$ and removing the dead branch.

B Proof of Swapping Statement

The (backwards) proof starts by an application of the rule for the conditional with P and Q instantiated to $=_{\mathcal{V}\setminus\{\hat{\boldsymbol{y}}\}}$ and e and e' instantiated to $x \notin \mathsf{dom}(\boldsymbol{L})$. There are four proof obligations:

1. $P \rightarrow \langle e\rangle_1 = \langle e'\rangle_2$. It is trivial.
2. $\vdash E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}} \sim E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}} : = \wedge\langle e'\rangle_1 \Rightarrow = \wedge\langle e'\rangle_1$. It follows from the fact that $S_{\hat{\boldsymbol{y}}}$ does not modify the free variables of e'.
3. $\vdash E_{\mathrm{l}}, y \leftarrow \boldsymbol{L}[x]; S_{\hat{\boldsymbol{y}}} \sim E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}}; y \leftarrow \boldsymbol{L}[x] : =_{\mathcal{V}} \wedge\neg\langle x \notin \mathsf{dom}(\boldsymbol{L})\rangle_1 \Rightarrow =_{\mathcal{V}}$. This proof obligation corresponds to showing that the code in the else branch commute with $S_{\hat{\boldsymbol{y}}}$. It can be discharged with a direct application of the rule for the assignment.
4. $\vdash E_{\mathrm{l}}, y \xleftarrow{\$} T; \boldsymbol{L}_+; S_{\hat{\boldsymbol{y}}} \sim E_{\mathrm{e}}, S_{\hat{\boldsymbol{y}}}; \mathsf{if}\ x = \hat{\boldsymbol{x}}\ \mathsf{then}\ y \leftarrow \hat{\boldsymbol{y}}\ \mathsf{else}\ y \xleftarrow{\$} T; \boldsymbol{L}_+ : =_{\mathcal{V}} \wedge\langle x \notin \mathsf{dom}(\boldsymbol{L})\rangle_1 \Rightarrow =_{\mathcal{V}}$
   where $\boldsymbol{L}_+$ is a shorthand for $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$. This proof obligation corresponds to showing that the code of the then branch commutes with $S_{\hat{\boldsymbol{y}}}$. The obligation is proved by case analysis on $x = \hat{\boldsymbol{x}}$.

   (a) If $x = \hat{\boldsymbol{x}}$, certified program transformations can be invoked to reduce the goal—using the precondition $x \notin \mathsf{dom}(\boldsymbol{L})$ and the hypothesis $x = \hat{\boldsymbol{x}}$—to

   $$\vdash E_{\mathrm{l}}, y \xleftarrow{\$} T; \hat{\boldsymbol{y}} \leftarrow y; \boldsymbol{L}_+ \sim E_{\mathrm{e}}, \hat{\boldsymbol{y}} \xleftarrow{\$} T; y \leftarrow \hat{\boldsymbol{y}}; \boldsymbol{L}_+ : =_{\mathcal{V}} \wedge\langle x \notin \mathsf{dom}(\boldsymbol{L})\rangle_1 \Rightarrow =_{\mathcal{V}}$$

   This latter equivalence is easy to establish.

   (b) If $x \neq \hat{\boldsymbol{x}}$, the equivalence is proved easily using a combination of certified program transformations and relational Hoare logic.

C Using Swapping Statements for OAEP

Consider an environment E with a procedure $\mathcal{O}$ that acts as an extended random oracle:

Oracle $\mathcal{O}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} \{0,1\}^{\ell}$;
  if caller $= \mathcal{A}$ then $\boldsymbol{L} \leftarrow (x,(y,\mathsf{true})) :: \boldsymbol{L}$
    else $\boldsymbol{L} \leftarrow (x,(y,\mathsf{false})) :: \boldsymbol{L}$
else
  $(y,b) \leftarrow \boldsymbol{L}(x)$;
  if caller $= \mathcal{A}$ then $\boldsymbol{L}(x) \leftarrow (y,\mathsf{true})$
return y

where the test caller $= \mathcal{A}$ indicates whether the caller is the adversary—we gloss over the encoding. Consider now the environment E' that differs from E only in the definition of $\mathcal{O}$, which becomes

Oracle $\mathcal{O}'(x)$:
foreach $(x,(y,b)) \in \boldsymbol{L}$ do
  if $b = \mathsf{false}$ then $y \xleftarrow{\$} \{0,1\}^{\ell}$; $\boldsymbol{L}(x) \leftarrow (y,\mathsf{false})$;
[code of $\mathcal{O}(x)$]

where, again, we gloss over the encoding of the foreach statement that traverses the list $\boldsymbol{L}$.

The equivalence $(E,c) \equiv (E',c)$ captures the fact that for tuples in $\boldsymbol{L}$ of the form $(x,(y,\mathsf{false}))$, the value of y is independent of the view of the adversary. Such equivalences are instrumental in many cryptographic proofs. Indeed, our (ongoing) proof of IND-CCA security of OAEP relies on this. We prove for specific games (E,c) that arise in the proof that the aforementioned equivalence holds using the logic for swapping statements. Performing program transformations and applying Lemma 3 in (E',c) instead that in (E,c) becomes much easier because freshly sampled elements in oracle $\mathcal{O}'$ are readily known to be independent of the execution history.