Submitted for publication

Formal Certification of Code-Based Cryptographic Proofs

Gilles Barthe Benjamin Grégoire Romain Janvier Santiago Zanella Béguelin

INRIA Sophia Antipolis - Méditerranée, France Microsoft Research - INRIA Joint Centre, France

{Gilles.Barthe,Benjamin.Gregoire,Romain.Janvier,Santiago.Zanella}@sophia.inria.fr

Abstract

As cryptographic proofs have become essentially unverifiable, cryptographers have argued in favor of systematically structuring proofs as sequences of 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 steps in the proof—transitions between games. While these techniques contribute to increase confidence in the security of cryptographic systems, code-based proofs involve such a large palette of concepts from different fields that machine-verified proofs seem necessary to achieve the highest degree of confidence. Indeed, Halevi has convincingly argued that a tool assisting in the construction and verification of proofs is necessary to solve the crisis with cryptographic proofs. This article reports a first step towards the completion of Halevi's programme through the implementation of a fully formalized framework for code-based proofs built on top of the Coq proof assistant. The framework has been used to yield machine-checked proofs of the PRP/PRF switching lemma and semantic security of ElGamal and OAEP encryption schemes.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—*Compilers, Optimization*; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—*Logics of programs, Mechanical verification, Pre- and post-conditions*; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—*Denotational semantics, Program analysis*.

General Terms Languages, Security, Verification.

Keywords game-based cryptographic proofs, compiler transformations and optimizations, relational Hoare logic, the Coq proof assistant.

1. Introduction

Provable security, whose origins can be traced back to the pioneering work of Goldwasser and Micali [22], advocates a mathematical approach based on complexity theory in which the goals and requirements of cryptosystems are specified precisely, and where the security proof is carried rigorously and makes 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 general, provable security statements do not refer directly to the probability p of the adversary breaking security, but to its advantage $\mathsf{Adv} = |p - \hat{p}|$ over a *blind*, uninformed adversary. Typically, $\hat{p}$ can be easily computed and by providing an upper bound for p, one also provides an upper bound for Adv. In a similar fashion, security assumptions about cryptographic primitives bound the probability of polynomial algorithms to solve difficult 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. Many security proofs establish an asymptotic behavior for the adversaries, and show that the advantage of any effective adversary is negligible w.r.t. a security parameter, typically the length of keys or messages.

Although the adoption of provable security has significantly enhanced confidence in cryptographic systems, the cryptographic community is increasingly wary about security proofs: several published proofs have been found incorrect. Even for such basic results as the PRP/PRF switching lemma, subtle errors in proofs have made their way into publications [12].

The game-playing technique 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 game transformations that preserve security. In a typical game-based proof, one considers transitions of the form $G \rightarrow^h G'$. Denoting by p and p' the winning probability of the adversary in games G and G' respectively, we require h to be a monotonic function preserving negligibility and such that $p \leq h(p')$. By successively refining the initial game G_0 to be analyzed into an ideal game G_n where one can provide a bound for the probability p_n,

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

one can obtain an upper bound for p_0, namely $p_0 \leq h(p_n)$, for $h = h_1 \circ \cdots \circ h_n$, and conclude that the advantage $\mathsf{Adv}_0 = |p_0 - \hat{p}|$ of the adversary in the initial game is negligible provided that $|h(p_n) - \hat{p}|$ is negligible.

The game-playing technique is widely applicable, it supports reasoning in both the standard and the random oracle model of cryptography and has been extensively used for proving security properties of a variety of schemes and protocols. Code-based techniques (CBT) is an instance of the game-playing technique that has been used successfully to verify state-of-the-art cryptographic schemes, see e.g. [12]; its distinguishing feature is to take a code-centric view of games, security hypotheses and computational assumptions, that are expressed using (probabilistic, imperative, polynomial-time) programs. Under this view, game transformations become program transformations, and can be justified rigorously by semantic means; for instance, many transformations can be viewed as common program optimizations (e.g. constant propagation, common subexpression elimination), and are justified by proving that the original and transformed programs are equivalent w.r.t. indistinguishability. Although CBT proofs are easier to verify and are more easily amenable to machine-checking, they go far beyond established theories of program equivalence and exhibit a surprisingly rich and broad set of reasoning principles that draws from program verification, algebraic reasoning, and proba-

Please do not distribute this paper directly, but refer insted to the webpage `http://www-sop.inria.fr/everest/certicrypt` where the latest version can be obtained.]

bility and complexity theory. Thus, despite the beneficial effect of their underlying framework, CBT proofs remain inherently difficult to verify. In an inspiring paper, Halevi [26] argues that formal verification techniques are mandatory to improve trust in game-based proofs, going as far as describing an interface to a tool for computer-assisted code-based proofs. To the best of our knowledge, however, there is no tool currently available that meets the needs of cryptographers.

This article describes CertiCrypt, a framework to construct machine-checked code-based proofs in the Coq proof assistant [39]. CertiCrypt achieves many important goals of Halevi's ideal tool. At the same time, it brings a formal semanticist perspective on the design of the tool, and builds upon ideas of Foundational Proof Carrying Code [4] to achieve the highest guarantees with the smallest trusted base. The main characteristics of CertiCrypt are:

Direct and faithful encoding of code-based techniques. In order to take advantage of the generality of the CBT approach and to be readily accessible to cryptographers, we have chosen a formalism that is commonly used by cryptographers to describe games. Concretely, the lowest layer of CertiCrypt is a deep embedding in Coq of an imperative programming language with random assignments, structured datatypes, and procedure calls. The language semantics takes into account non-standard features such as complexity of programs, variable usage and calling policies, that are of paramount importance in cryptographic proofs. In addition, CertiCrypt provides a library for expressing common security properties, such as indistinguishability under chosen plain-text attacks (IND-CPA) for encryption primitives, and computational hypotheses, such as the Decisional Diffie-Hellmann (DDH) assumption.

Support for automated proofs. Automating proof steps is necessary to ensure an efficient use of any formal tool. We have therefore developed tactics for the most common transformations; all tactics are certified, in the sense that they are proved correct with respect to the operational semantics. Transformations fall into three main categories; 1) semantics preserving transformations, including compiler optimizations such as dead code elimination, code motion, constant propagation and common subexpression elimination; 2) transformations based on indistinguishability, i.e. a change that cannot be detected with a non-negligible probability; and 3) transformations based on failure events, where both games behave identically unless a certain *failure* occurs, and it is shown that this failure occurs with negligible probability.

Complete and independently verifiable proofs. CertiCrypt benefits from being developed on top of the Coq proof assistant to go beyond Halevi's vision in two respects. First, it supports the construction of full proofs, whereas Halevi mostly focuses on their "mundane parts". Second, it permits independent verifiability of proofs by third parties, which is an important motivation behind game-based proofs. Regarding full proofs, CertiCrypt requires that all (complexity-theoretic, group-theoretic, probabilistic) side conditions to apply transformations are justified within Coq, and also enables *ad hoc* reasoning, e.g. to conclude the proof in case of a sequence of games that ends in a non-trivial game [27]. Regarding verifiability, CertiCrypt inherits from Coq its ability to provide certificates, or proof objects, that are automatically verifiable with a small trusted core, namely the type-checker of Coq.

We have applied CertiCrypt to machine-check three famous examples of code-based proofs: the PRP/PRF switching lemma and semantic security of the ElGamal and OAEP encryption schemes. The first and second examples are basic applications of the main code-based techniques, but nonetheless nicely illustrate many aspects of our work. The third example is far more challenging, and involves a large number of transitions as well as some advanced game-based techniques that have not been formally justified before. The justification of some steps in the proof relies on program invariants which are proved using program verification techniques; for this purpose we have built a Hoare logic and an executable weakest precondition calculus for CertiCrypt underlying language. Other steps in the proof involve properties that are universally quantified over all well-formed adversaries. We have therefore modeled precisely the notion of well-formedness, which is very often left implicit in cryptographic proofs, and derived an induction principle that allows to reason about an unspecified well-formed adversary.

2. Basic Examples

The aim of this section is to illustrate the principles of CertiCrypt on two basic examples of game-based proofs: the PRP/PRF switching lemma and semantic security of ElGamal encryption. Description of the internals of CertiCrypt is deferred to later sections.

2.1 The ElGamal Encryption Scheme

ElGamal [20] is a widely used asymmetric encryption scheme, and an emblematic example of game-based proofs. The proof of its semantic security is very direct, see e.g. [36], but still embodies the most common techniques that arise in more complex proofs.

ElGamal inner workings ElGamal is a probabilistic public-key encryption scheme whose security relies on the assumption that computing discrete logarithms in certain cyclic groups is a hard problem. Given a cyclic group of order $q+1$ generated by g, to generate a new key pair one uniformly samples an integer x in the interval $[0..q]$ and takes x as the private key and $\alpha = g^x$ as the public one. The ciphertext for a given plaintext m (an element of the group) is $(\beta = g^y, \zeta = \alpha^y \times m)$ where y is uniformly sampled in $[0..q]$. Using the secret key x it is easy to recover the plaintext from a ciphertext (β, ζ) by computing $m = \zeta \times \beta^{-x}$. As an encryption scheme, ElGamal is composed of three algorithms:

- The key generator $\mathsf{KG}() \stackrel{\text{def}}{=} x \xleftarrow{\$} [0..q];\ \mathsf{return}\ (x, g^x)$
- The encryption algorithm $\mathsf{Enc}(\alpha, m) \stackrel{\text{def}}{=} y \xleftarrow{\$} [0..q];\ \mathsf{return}\ (g^y, \alpha^y \times m)$
- The decryption algorithm $\mathsf{Dec}(x, \beta, \zeta) \stackrel{\text{def}}{=} \mathsf{return}\ \zeta \times \beta^{-x}$.

Semantic security of ElGamal Fig. 1 presents a high level view of the proof that ElGamal is semantically secure (equivalently, IND-CPA secure) under the Decisional Diffie-Hellman (DDH) assumption. The objective is to prove that it is impossible to gain significant information about a plaintext given only a corresponding ciphertext and the public key. This is formally expressed using two games[1] $\mathsf{ElGamal}_0$ and $\mathsf{ElGamal}_1$. Each game begins by generating a fresh public key, which is given to a probabilistic polynomially bounded algorithm $\mathcal{A}$ that outputs a pair of messages (m_0, m_1). Depending on the game, either m_0 or m_1 is encrypted and the resulting ciphertext is given to another probabilistic polynomially bounded algorithm[2] $\mathcal{A}'$ which tries to guess which message has been encrypted by outputting a single bit d—if $d = 1$, the guess is m_1, otherwise m_0. We say that ElGamal is semantically secure iff for any pair of adversaries $\mathcal{A}$, $\mathcal{A}'$, the difference

[1] IND-CPA security can also be described using only one game; both descriptions are evenly used in the literature. We find that the description using two games eases the presentation.

[2] Both adversaries are allowed to share state via global variables and can thus be regarded as a single adversary (structuring games in this way is common in game-based proofs.)

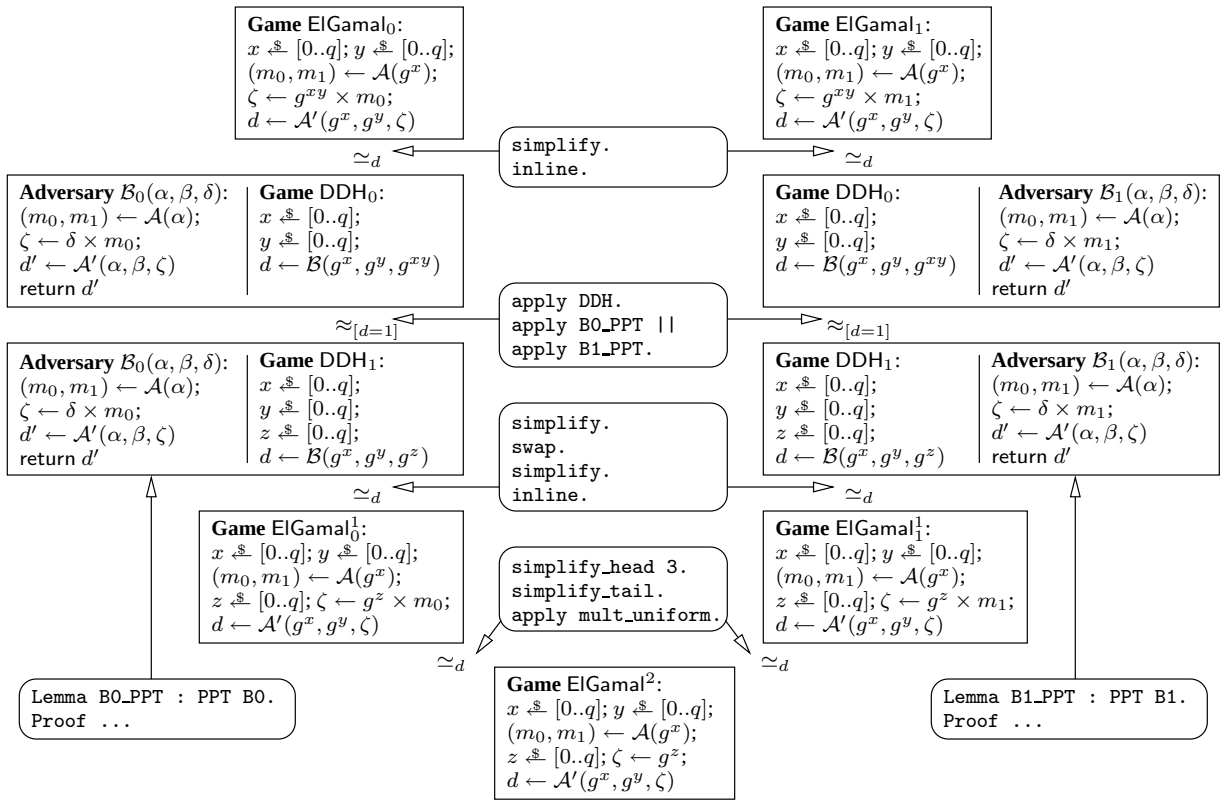

Figure 1. Game-based proof of ElGamal semantic security

in the probability of outputting $d = 1$ (equivalently, $d \neq 1$) in either game is a negligibly function of the security parameter:[3] $\mathsf{ElGamal}_0 \approx_{[d=1]} \mathsf{ElGamal}_1$.

The security of ElGamal relies on the DDH assumption [17], 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 and z are uniformly sampled in $[0..q]$. In our setting, DDH is formulated more precisely, stating that for any polynomial adversary $\mathcal{B}$, $\mathsf{DDH}_0 \approx_{[d=1]} \mathsf{DDH}_1$ (DDH_0 and DDH_1 are defined in Fig. 1).

Proof The proof is done by showing that both initial games are indistinguishable from a third one, namely $\mathsf{ElGamal}_2$. For $\mathsf{ElGamal}_0$ this is achieved using the following sequence of transformations:

$$\mathsf{ElGamal}_0 \simeq_d \mathsf{DDH}_0 \approx_{[d=1]} \mathsf{DDH}_1 \simeq_d \mathsf{ElGamal}_0^1 \simeq_d \mathsf{ElGamal}^2$$

The transition from $\mathsf{ElGamal}_0$ to $\mathsf{ElGamal}_0^1$ is justified by a reduction to the DDH assumption. The transition from $\mathsf{ElGamal}_0^1$ to $\mathsf{ElGamal}^2$ makes use of an algebraic property of cyclic groups: when multiplying a uniformly distributed element of the group by another element, the result is uniformly distributed. For $\mathsf{ElGamal}_1$ the sequence of transformations is completely symmetric.

In order to do the reduction to the DDH assumption it is necessary to construct an adversary $\mathcal{B}_0$ such that the distribution of the value of d after running games DDH_0 and $\mathsf{ElGamal}_0$ is exactly the same (and similarly for DDH_1 and $\mathsf{ElGamal}_0^1$). This is denoted by $\mathsf{ElGamal}_0 \simeq_d \mathsf{DDH}_0$ and is proved by applying semantics preserving transformations using the following tactics: `simplify`, that removes common context in both games, `swap`, that hoists instructions when possible in order to obtain a common prefix, and `inline`, that inlines a procedure call and optimizes the resulting code by performing dead code elimination and copy propagation. To justify the reduction we must also prove that the adversary $\mathcal{B}_0$ is polynomially bounded assuming so are $\mathcal{A}$ and $\mathcal{A}'$. This is easily done since $\mathcal{B}_0$ is just a *linear* extension of $\mathcal{A}$ and A'. Then, by appealing to the assumption $\mathsf{DDH}_0 \approx_{[d=1]} \mathsf{DDH}_1$ and the transitivity of the $\approx_{[d=1]}$ relation we conclude that $\mathsf{ElGamal}_0 \approx_{[d=1]} \mathsf{ElGamal}_0^1$.

In the last transition, we eliminate the common context in the two games with the exception of the instruction $z \xleftarrow{\$} [0..q]$, and then make use of the algebraic property mentioned above to prove that $z \xleftarrow{\$} [0..q];\ \zeta \leftarrow g^z \times m_0$ and $z \xleftarrow{\$} [0..q];\ \zeta \leftarrow g^z$ induce the same distribution on ζ, obtaining $\mathsf{ElGamal}_0^1 \simeq_d \mathsf{ElGamal}^2$. Together with the result of the previous paragraph, this allows to conclude that $\mathsf{ElGamal}_0 \approx_{[d=1]} \mathsf{ElGamal}^2$, and by a completely symmetric argument that $\mathsf{ElGamal}_1 \approx_{[d=1]} \mathsf{ElGamal}^2$, thus proving the desired result.

Comparison with other works ElGamal is a standard example for security proofs, and has been used by several authors to validate their work. We briefly comment on three proofs that are closely related to ours. The most recent, and closely related is a formalization in Coq of a game-based proof of ElGamal semantic security by Nowak [31]. While we opt for a deep embedding, Nowak uses a shallow one, modeling adversaries directly as Coq functions. This implies that the resulting framework can only provide limited support for proof automation: because there is no special syntax for writing games, mechanizing syntactic transformations becomes very difficult. All in all, the resulting proof is elegant but ignores

[3] The security parameter, implicit in this presentation, determines a cyclic group of order q with generator g by indexing a family of groups where the DDH problem is believed intractable.

complexity issues.[4] An earlier work by Barthe, Cederquist and Tarento [9] provides a formal proof of security of (signed) ElGamal in Coq. The proof is not completely formalized, only the "mathematical" arguments are proved. Moreover the proof relies on idealized models (generic model and random oracle model) while the proof presented here is done in the standard model of cryptography. Corin and den Hartog [19] developed a Hoare-style proof system for game-based cryptographic proofs. The formalism is not sufficiently powerful to express precisely the security goals: notions such as negligible advantage or effective adversary are not modeled. Moreover, there is currently no computer assistance for reasoning using this logic.

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 states that it is indeed justified to do such replacement in a game without significantly changing the advantage of a polynomial adversary. The intuition is that the probability of a game outputting a given value is the same if a PRP is replaced by a PRF but no collisions are observed, but as explained in [12] the proof is not trivial. Nonetheless, using Lemma 2 presented in Sec. 4, the proof can be easily done in CertiCrypt (see Fig. 2 for a high level view of the proof). The goal is to prove that the difference in the probability that an adversary outputs 1 when given oracle access to a PRP and when given oracle access to a PRF is negligible. This statement is encoded in CertiCrypt as $G_{\mathrm{PRP}} \approx_{[d=1]} G_{\mathrm{PRF}}$ (these games are defined in Fig. 2), and proved by means of the sequence of games:

$$G_{\mathrm{PRP}} \simeq_d G'_{\mathrm{PRP}} \approx_{[d=1]} G'_{\mathrm{PRF}} \simeq_d G_{\mathrm{PRF}}$$

The first and last transitions in the sequence are semantics preserving transformations used to reformulate the statement in the form required to apply Lemma 2. For doing so, we introduce a variable bad that is set to true whenever a collision is found in G'_{PRF}, and we reformulate G'_{PRP} accordingly to be syntactically equal until bad is set. Then, by applying Lemma 2 and proving that the probability that bad is set to true is negligible, we can conclude. It is important to note that, in CertiCrypt, we explicitly distinguish the global variables that can be accessed by the adversary, such as the security parameter η, that can be read but not written, and the global variables that it cannot even read, such as the association list L or the variable bad, while this is usually left unspecified in the literature.

3. The pWHILE Language

To define games we use a probabilistic imperative language with procedure calls; all probabilistic features are encapsulated in a set $\mathcal{BI}$ of basic instructions, which is left unspecified in large part of the development. We assume given a set $\mathcal{V}$ of variable identifiers, a partition $(\mathcal{V}_{\mathsf{glb}}, \mathcal{V}_{\mathsf{loc}})$ of $\mathcal{V}$ of identifiers for global and local variables respectively, a set $\mathcal{P}$ of procedure identifiers, and a set $\mathcal{E}$ of expressions. We define inductively the set of commands by the following clauses:

$\mathcal{I}$	::=	$\mathcal{BI}$	basic instruction
	|	if $\mathcal{E}$ then $\mathcal{C}$ else $\mathcal{C}$	conditional
	|	while $\mathcal{E}$ do $\mathcal{C}$	while loop
	|	$\mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E})$	procedure call
$\mathcal{C}$	::=	nil	nop
	|	$\mathcal{I}$; $\mathcal{C}$	sequence

[4] See [18] for preliminary treatment of complexity in a shallow-embedding setting.

In the rest of this paper the metavariables $c,\ c_i$ range over $\mathcal{C}$; $x,\ x_i$ over $\mathcal{V}$; $e,\ e_i$ over $\mathcal{E}$; and p, p_i over $\mathcal{P}$. Since sequencing is associative we reuse the notation $c_1;\ c_2$ for denoting also the sequence of two commands. To the purpose of this presentation, we instantiate the set of basic instructions $\mathcal{BI}$ as follows:

$\mathcal{BI}$	::=	$\mathcal{V} \leftarrow \mathcal{E}$	deterministic assignment
	|	$\mathcal{V} \xleftarrow{\$} [0..\mathcal{E}]$	uniform sampling (integer interval)
	|	$\mathcal{V} \xleftarrow{\$} \{0,1\}^{\mathcal{E}} \backslash \mathcal{E}$	uniform sampling (bitstrings)

In the following, $x \xleftarrow{\$} \{0,1\}^e$ will be used as a shorthand for $x \xleftarrow{\$} \{0,1\}^e \backslash \emptyset$.

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 specifying games, though),*

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

In the the remainder of this section we assume the existence of an implicit environment E.

In the actual development we instantiate values in the language to include booleans, bitstrings, natural numbers, and group elements as base types and pairs and lists as structured types; expressions are instantiated accordingly to include common operations. However, to ease the presentation, in this section we leave them unspecified and assume the existence of a function $[\![\cdot]\!]$, that evaluates an expression in a given memory (a mapping from local and global variables to values). We give meaning to programs by means of a small-step semantics, using a frame stack to deal with procedure calls. The small-step semantics relates a deterministic state to a (sub-)probability distribution over deterministic states (Fig. 3). Following Paulin and Audebaud [5] formulation, (sub-)probability distributions over a set X are represented using the continuation monad

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

The unit and bind operators of the monad, satisfying the usual properties, are defined as

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

A deterministic state is a triple consisting of a command $(c : \mathcal{C})$, a memory $(m : \mathcal{M})$, and a frame stack $(F : \mathsf{list\ frame})$. Upon a call (3rd rule in Fig. 3), 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 callee. 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 (2nd rule) 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 callee; 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 has terminated and thus nothing is done except embedding the final state into the monad using the unit operator.

The one step execution relation $\rightsquigarrow$ defines a semantic function $[\![\cdot]\!]^1 : \mathcal{S} \to \mathcal{D}(\mathcal{S})$; the continuation monad allows us to compose this semantic function with itself to obtain an n-step execution function $[\![\cdot]\!]_n$:

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

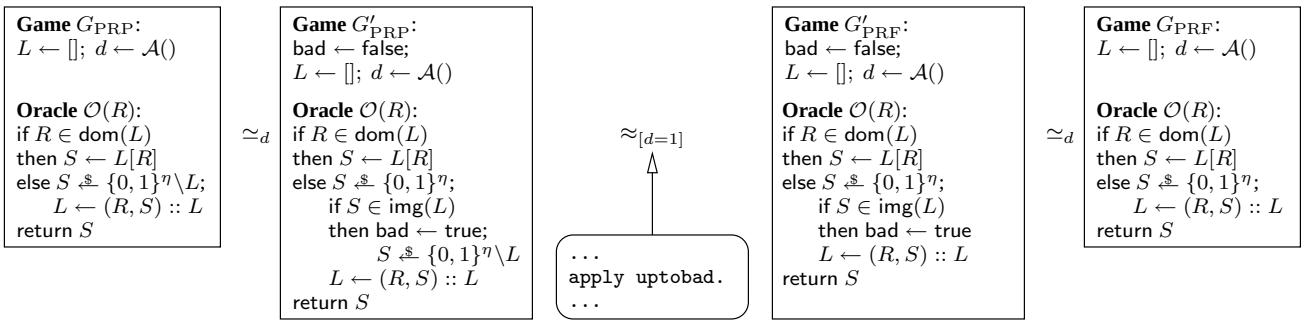

Figure 2. Code-based proof of the PRP/PRF switching lemma

$$
\begin{array}{rcll}
(\mathsf{nil}, m, \emptyset) & \rightsquigarrow & \mathsf{unit}\ (\mathsf{nil}, m, \emptyset) & \\
(\mathsf{nil}, m, (x, e, c, l) :: F) & \rightsquigarrow & \mathsf{unit}\ (c, (l, m.\mathsf{glb})\{x \leftarrow [\![e]\!]m\}, F) & \\
(x \leftarrow p(\vec{e});\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (E(p).\mathsf{body}, (\{E(p).\mathsf{params} \leftarrow [\![\vec{e}]\!]m\}, m.\mathsf{glb}), (x, E(p).\mathsf{re}, c, m.\mathsf{loc}) :: F) & \\
(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c_1;\ c, m, F) & \text{if } [\![e]\!]m = \mathsf{true} \\
(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c_2;\ c, m, F) & \text{if } [\![e]\!]m = \mathsf{false} \\
(\mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \rightsquigarrow & \mathsf{unit}\ (c;\ \mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \text{if } [\![e]\!]m = \mathsf{true} \\
(\mathsf{while}\ e\ \mathsf{do}\ c;\ c', m, F) & \rightsquigarrow & \mathsf{unit}\ (c', m, F) & \text{if } [\![e]\!]m = \mathsf{false} \\
 & & & \\
(x \leftarrow e;\ c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c, m\{x \leftarrow [\![e]\!]m\}, F) & \\
(x \xleftarrow{\$} [0..e];\ c, m, F) & \rightsquigarrow & \lambda f.\ \sum_{i=0}^{n} \frac{1}{1+n} f(c, m\{x \leftarrow i\}, F) & \text{where } n = [\![e]\!]m \\
(x \xleftarrow{\$} \{0,1\}^e \backslash e_L;\ c, m, F) & \rightsquigarrow & \lambda f.\ \sum_{bs \in \{0,1\}^n \backslash L} \frac{1}{2^n - |L|} f(c, m\{x \leftarrow bs\}, F) & \text{where } n = [\![e]\!]m \text{ and } L = [\![e_L]\!]m
\end{array}
$$

Figure 3. Probabilistic semantics of pWHILE programs

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

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

where $f|_{\mathsf{final}} : \mathcal{M} \to [0, 1]$ 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 $[\![(c, m, \emptyset)]\!]_n\ f|_{\mathsf{final}}$ is increasing and upper bounded by 1, this least upper bound always exists and corresponds to the limit of the sequence. As an example, it is left as an exercise to the reader to verify that the denotation of $x \xleftarrow{\$} [0..1];\ y \xleftarrow{\$} [0..1]$ in m is

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

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

Probabilistic termination The semantics is sufficiently expressive to characterize different notions of termination. For instance, one can characterize the class of always-terminating—*lossless*—programs as the programs satisfying the condition

$$\mathsf{Lossless}(c) \stackrel{\text{def}}{=} \forall m.\ [\![c]\!]\ m\ \mathbb{1}_{\mathsf{true}} = 1$$

The probability that a given program does not terminate starting from the initial memory m is $1 - [\![c]\!]\ m\ \mathbb{1}_{\mathsf{true}}$.

In game-based cryptographic proofs we are frequently interested in notions of asymptotic termination depending on a security parameter η. To define these notions we have extended our semantics to take into account the cost incurred in executing a program by extending deterministic states with an extra parameter representing this cost, i.e. we take distributions over $\mathcal{S} \times \mathbb{N}$ instead of simply over $\mathcal{S}$.

Definition 2 (Probabilistic polynomial time termination[5])**.** *A command c parametrized by η is said to be PPT, denoted $\mathsf{PPT}(c, \eta)$ if there exists a polynomial π such that for every polynomially bounded memory m,*

$$[\![c]\!]\ m\ (\lambda\ (m', n).\ \mathbf{if}\ n \leq \pi(\eta)\ \mathbf{then}\ 1\ \mathbf{else}\ 0) = 1$$

Observe that we cannot simply quantify over every initial memory in the above definition, because by doing so we will rule out most programs from the definition—a single instruction operating on a non-polynomially bounded value may take an exponential time to execute. Instead, we quantify over every initial memories m associating only polynomially-bounded values to variables (denoted $\mathsf{poly}(m, \eta)$).

4. Computational Indistinguishability

In CBT proofs it is usually needed to show that two games are computationally indistinguishable w.r.t. some observable event. This is expressed by saying that the difference between the probability of the event occurring in each game is a negligible function of a security parameter η. Formally, a function $\nu : \mathbb{N} \to \mathbb{R}$ is negligible

[5] This notion is not what is known as expected PPT, it is strictly stronger. Nevertheless both definitions are interchangeable in most cryptographic proofs.

iff

$$\mathsf{negl}(\nu) \stackrel{\text{def}}{=} \forall\, c.\, \exists\, n_c.\, \forall\, n.\, n \geq n_c \Rightarrow |\nu(n)| \leq n^{-c}$$

Given an indexed family of memories $M : \mathbb{N} \to \mathcal{M}$, the difference in the probability of an event A between two games G_1 and G_2 is said to be negligible iff

$$M \vDash G_1 \approx_A G_2 \stackrel{\text{def}}{=} \mathsf{negl}(\lambda\eta.\, |\Pr_{G_1,M(\eta)}[A] - \Pr_{G_2,M(\eta)}[A]|)$$

Fundamental lemma of game-based proofs A technique very often used for proving two games indistinguishable 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 1 (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 $\Pr_{G_1,m}[A \wedge \neg F] = \Pr_{G_2,m}[B \wedge \neg F]$, then*

$$|\Pr_{G_1,m}[A] - \Pr_{G_2,m}[B]| \;\leq\; \Pr_{G_i,m}[F] \qquad i = 1, 2$$

In most code-based proofs, the failure condition is 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'_{PRP} and G'_{PRF} in Fig. 2 satisfy this condition. We have used this syntactic criterion to implement in Coq a specialization of the fundamental lemma for game-based proofs.

Lemma 2 (Fundamental lemma—based on reflection)**.**

$$\begin{array}{l}\forall\, G_1\; G_2\; A.\\ \quad \mathsf{uptobad}(G_1, G_2) \wedge \mathsf{Lossless}(G_1) \wedge \mathsf{Lossless}(G_2) \Rightarrow\\ \quad |\Pr_{G_1,m}[A] - \Pr_{G_2,m}[A]| \leq \Pr_{G_i,m}[\mathsf{bad} = \mathsf{true}] \quad i = 1, 2\end{array}$$

To prove that two games are computationally indistinguishable one can apply this lemma and then show that the probability of the failure event is negligible in one of the games. The hypotheses in the lemma may be proved by reflection in the Coq implementation, since we provide syntactic criteria for proving Lossless whenever it is syntactically provable (this is the case for programs without loops or recursive functions). Observe that we need to ensure that the termination behavior of both games is the same after setting bad to true, requiring both games to be Lossless is sufficient but not necessary.

5. Computational Equivalence

In this section, we present a general notion of program equivalence for pWHILE programs.

5.1 Probabilistic Relational Hoare Logic (pRHL)

A particular but useful way to prove that the difference in the probability of occurrence of an event A in two games is negligible is to prove that this probability does not change at all, i.e the two programs are equivalent w.r.t. the event A:

$$G_1 \simeq_A G_2 \stackrel{\text{def}}{=} \forall m, \Pr_{G_1,m}[A] = \Pr_{G_2,m}[A]$$

Unfortunately this property is not contextual, the transition between G_1 and G_2 is generally a local replacement of a part P by a part P' in the main code or the code of a procedure, and P and P' are not necessarily observationally equivalent, they need only to be equivalent in the context where the replacement is done. However, it is often sufficient to characterize the context where the replacement is valid by a precondition and a postcondition over the memories before and after the evaluation of the replaced part.

A judgment $\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi$ in our probabilistic relational Hoare Logic relates the evaluation of a program G_1 to the evaluation of a program G_2. If the two programs are deterministic, it 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 and G_2 starting from m_1 and m_2 terminate with final memories m'_1 and m'_2 respectively, then $m'_1\ \Phi\ m'_2$ holds. So, restricted to deterministic programs, our equivalence relation corresponds to the Relational Hoare Logic of Benton [13]. Unfortunately, the probabilistic case is more difficult since the semantics maps programs and initial memories to distributions over memories and thus one needs to consider relations over distributions instead of simply relations over memories.

Definition 3 (Meaning of pRHL judgments)**.**

- *Two functions are equivalent w.r.t. a predicate Φ iff:*
$$f \sim_\Phi g \stackrel{\text{def}}{=} \forall\, m_1\ m_2.\, m_1\ \Phi\ m_2 \Rightarrow f(m_1) = g(m_2)\ ;$$
- *This notion is extended to distributions as follows:*
$$d_1 \sim_\Phi d_2 \stackrel{\text{def}}{=} \forall\, f\ g.\, f \sim_\Phi g \Rightarrow d_1\ f = d_2\ g\ ;$$
- *Two programs G_1 and G_2 are equivalent w.r.t. a precondition Ψ and a postcondition Φ iff:*
$$\begin{array}{l}\vDash G_1 \sim G_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=}\\ \qquad \forall\, m_1\ m_2.\, m_1\ \Psi\ m_2 \Rightarrow [\![G_1]\!]\ m_1 \sim_\Phi [\![G_2]\!]\ m_2\ .\end{array}$$

Using this definition we can derive the system in Fig. 4, corresponding to the inference rules of RHL in [13]. The major difference with RHL is that pRHL judgments talk about probabilistic programs instead of deterministic ones, and that pre and postconditions are not restricted to a particular syntax, they can be any relation over memories expressible in Coq.

The [R-Rand] and [R-Case] rules do not appear in RHL. The former deals with random assignments, where d_1 and d_2 stand for expressions denoting distributions. We require that in a given memory they evaluate to the same distribution in $\mathcal{D}(\mathsf{val})$, and that the postcondition holds no matter which value in the support of the distribution is used to update the memories in each program. The latter rule permits to do a case analysis on the evaluation of an arbitrary relation in the initial memories. Together with simple rules in the spirit of

$$\frac{m_1\ \Psi'\ m_2 \stackrel{\text{def}}{=} m_1\ \Psi\ m_2 \wedge [\![e]\!]m_1 \quad \vdash c_1 \sim c : \Psi' \Rightarrow \Phi}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sim c : \Psi' \Rightarrow \Phi}$$

it subsumes [R-Cond] and allows to derive a judgment showing the equivalence of $(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$ and $(\mathsf{if}\ \neg e\ \mathsf{then}\ c_2\ \mathsf{else}\ c_1)$, which is otherwise not possible.

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 reveals insufficient, thus avoiding completeness issues.

5.2 Mechanizing Equivalence Proofs

Hand building derivations in pRHL to prove program equivalence is tedious. However, by restricting pre and postconditions to relations based on equality over a subset of program variables, proving equivalence becomes a semi-decidable problem.

To prove $G_1 \simeq_A G_2$ it is sufficient to determine the set of variables O on which the event A depends and find another set

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

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

$$\frac{m_1\ \Psi\ m_2 \stackrel{\text{def}}{=} [\![d_1]\!]m_1 = [\![d_2]\!]m_2\ \wedge\ \forall\, v \in \mathsf{supp}([\![d_1]\!]m_1).\ (m_1\{x_1 \leftarrow v\})\ \Phi\ (m_2\{x_2 \leftarrow v\})}{\vdash E_1, x_1 \stackrel{\$}{\leftarrow} d_1 \sim E_2, x_2 \stackrel{\$}{\leftarrow} d_2 : \Psi \Rightarrow \Phi} \text{[R-Rand]}$$

$$\frac{\begin{array}{c}\vdash E_1, c_1 \sim E_2, c_2 : (\lambda\, m_1\, m_2.\ m_1\ \Psi\ m_2 \wedge [\![e_1]\!]m_1 \wedge [\![e_2]\!]m_2) \Rightarrow \Phi \\ \vdash E_1, c_1' \sim E_2, c_2' : (\lambda\, m_1\, m_2.\ m_1\ \Psi\ m_2 \wedge \neg[\![e_1]\!]m_1 \wedge \neg[\![e_2]\!]m_2) \Rightarrow \Phi\end{array}}{\vdash E_1, \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim E_2, \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' : (\lambda\, m_1\, m_2.\ m_1\ \Psi\ m_2 \wedge [\![e_1]\!]m_1 = [\![e_2]\!]m_2) \Rightarrow \Phi} \text{[R-Cond]}$$

$$\frac{m_1\ \Phi'\ m_2 \stackrel{\text{def}}{=} m_1\ \Phi\ m_2 \wedge [\![e_1]\!]m_1 = [\![e_2]\!]m_2 \quad \vdash E_1, c_1 \sim E_2, c_2 : (\lambda m_1\, m_2.\ m_1\ \Phi\ m_2 \wedge [\![e_1]\!]m_1 \wedge [\![e_2]\!]m_2) \Rightarrow \Phi'}{\vdash E_1, \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim E_2, \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Phi' \Rightarrow (\lambda m_1\, m_2.\ m_1\ \Phi\ m_2 \wedge \neg[\![e_1]\!]m_1 \wedge \neg[\![e_2]\!]m_2)} \text{[R-Whl]}$$

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

$$\frac{\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi \quad \mathsf{SYM}(\Psi) \quad \mathsf{SYM}(\Phi)}{\vdash G_2 \sim G_1 : \Psi \Rightarrow \Phi} \text{[R-Sym]} \qquad \frac{\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi \quad \vdash G_2 \sim G_3 : \Psi \Rightarrow \Phi \quad \mathsf{PER}(\Psi) \quad \mathsf{PER}(\Phi)}{\vdash G_1 \sim G_3 : \Psi \Rightarrow \Phi} \text{[R-Tr]}$$

$$\frac{\vdash G_1 \sim G_2 : (\lambda m_1\, m_2.\ m_1\ \Psi\ m_2 \wedge m_1\ \Psi'\ m_2) \Rightarrow \Phi \quad \vdash G_1 \sim G_2 : (\lambda m_1\, m_2.\ m_1\ \Psi\ m_2 \wedge \neg(m_1\ \Psi'\ m_2)) \Rightarrow \Phi}{\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi} \text{[R-Case]}$$

Figure 4. Selection of derived rules for pRHL

I of variables such that the following holds:

$$\vDash G_1 \simeq_O^I G_2 \stackrel{\text{def}}{=} \vDash G_1 \sim G_2 : =_I \Rightarrow =_O ,$$

where $m_1 =_X m_2 \stackrel{\text{def}}{=} \forall\, x \in X.\ m_1\ x = m_2\ x$. Unfortunately this is too restrictive as showed by the example:

$$p(w) : \mathsf{if}\ w < 5\ \mathsf{then}\ r \leftarrow w + x\ \mathsf{else}\ r \leftarrow w;\ \mathsf{return}\ r$$
$$p(w) : \mathsf{return}\ w$$
$$main() : x \leftarrow 0;\ c$$

Under the hypothesis that global variable x is equal to 0 at the beginning of the body of p, it is easy to prove that both versions of the procedure are equivalent (use constant propagation on x, copy propagation and dead code elimination). Therefore, if the code c preserves the invariant $(x = 0)$, one can show that the two programs are equivalent. To that end, we first define the following predicate[6]

$$m_1 =_{X,\varphi} m_2 \stackrel{\text{def}}{=} m_1 =_X m_2 \wedge \varphi(m_1) \wedge \varphi(m_2) ,$$

and then specialize our equivalence relation as follows

$$\vDash_\varphi G_1 \simeq_O^I G_2 \stackrel{\text{def}}{=} \vDash G_1 \sim G_2 : =_{I,\varphi} \Rightarrow =_{O,\varphi} .$$

In general, the global invariant is not satisfied by the initial memories of the games, it is only established after reaching a certain point in the execution of the programs. To show that the invariant holds we have implemented in Coq a certified weakest precondition calculus for probabilistic programs (without loops).

Note that the instantiation of the previous judgment with the same command ($\vdash_\varphi E_1, c \simeq_O^I E_2, c$) is a generalization of the type system for secure information flow of Volpano and Smith [37]. Given a set of input variables I, it is semi-decidable to find a set of output variables for which the previous judgment is sound—it can be done using a calculus of variable dependencies. The converse problem is also semi-decidable. We have constructed (partial) functions called eqobsIn (eqobsOut) that given a set O (I) of output (input) variables compute a set I (O) of input (output) variables which is sufficient to ensure $\vDash_\varphi E_1, c \simeq_O^I E_2, c$ holds.

Interprocedural analysis The main difficulty in implementing the algorithms described in the previous paragraph is due to the presence of procedure calls in our language. Each time it crosses a procedure call, the algorithm should be able to compute the input (output, resp.) set of variables whose equality is guaranteed by the call as well as telling whether the call preserves the invariant. Without recursive calls, this can be done by a recursive inspection of the procedure bodies, with the exception of adversaries (whose code is unknown). To resolve the problem we assume given a partial function providing information about the code of the procedures in both programs. This function can be seen as an environment in a type system. The correctness of our algorithms rely on the correctness of this function. For each procedure p, we store three different kinds of information about its declaration in both environments: whether the procedure is lossless, which global variables it does not modify, and a triple composed of a subset I_1 of the formal parameters, and two sets I_p, O_p of global variables. The validity of this triple is semantically expressed as:

$$\begin{array}{c}\exists\, O_1.\ \vDash_\varphi E_1, E_1(p).\mathsf{body} \simeq_{O_p \cup O_1}^{I_p \cup I_1} E_2, E_2(p).\mathsf{body} \\ \wedge\ [\![E_1(p).\mathsf{re}]\!] \sim_{=_{O_p \cup O_1}} [\![E_2(p).\mathsf{re}]\!]\end{array}$$

Loosely speaking, the above formula states that the invariant is preserved, and that equality over I_p and I_1 is sufficient to guarantee that both versions of p return the same value and result in memories that are equal w.r.t. O_p. In Sec. 6 we describe a method for deriving this information for an unknown adversary from the information of its oracles.

Transformations based on variable dependencies The eqobsIn and eqobsOut functions are key stones of a variety of other transformations that we have certified. A transformation that allows to eliminate a common context when proving two programs equivalent is based on the following rule, which is easily derivable from

[6] Since we are interested in doing interprocedural optimizations, we consider invariants φ depending only on global variables.

rule [R-Seq]:

$$\frac{\vdash_\varphi E_1, c_1 \simeq^{I}_{I'} E_2, c_1 \quad \vdash_\varphi E_1, c_3 \simeq^{O'}_{O} E_2, c_3 \quad \vdash_\varphi E_1, c_2 \simeq^{I'}_{O'} E_2, c_2'}{\vdash_\varphi E_1, c_1; c_2; c_3 \simeq^{I}_{O} E_2, c_1; c_2'; c_3}$$

The main difficulty to apply this rule is to find I' and O'. Given c and c', the common context c_1, c_3, and sets I' and O' satisfying the premises can be found automatically. This leads to the simplified rule:

$$\frac{\mathsf{context}(I, c, c', O) = (I', c_2, c_2', O') \;\; \vdash_\varphi E_1, c_2 \simeq^{I'}_{O'} E_2, c_2'}{\vdash_\varphi E_1, c \simeq^{I}_{O} E_2, c'}$$

where context is a certified algorithm built using eqobsIn and eqobsOut. Using the same idea, we have constructed certified algorithms for removing only a common prefix or suffix.

In many steps in game-based proofs one faces the problem of proving the equivalence of two games which are syntactically equal up to code-motion. We have constructed a function testing whether a sequence of two pieces of code in a program can be swapped. Using this function we have constructed a certified algorithm that given two commands, repeatedly hoists common instructions to obtain a maximal common prefix, which can then be eliminated using the previous rule. Its correctness is based on the rule:

$$\frac{\begin{array}{c}\vdash E_1, c_1 \simeq^{I_1}_{O_1} E_2, c_1 \quad \vdash E_1, c_2 \simeq^{I_2}_{O_2} E_2, c_2 \\ O_1 \cap O_2 = \emptyset \quad O_1 \cap I_2 = \emptyset \quad O_2 \cap I_1 = \emptyset \\ \mathsf{NotModify}(E_i, c_1, I_2 \cup O_2)\ \mathsf{NotModify}(E_i, c_2, I_1 \cup O_1)\ i = 1, 2\end{array}}{\vdash E_1, c_1;\ c_2 \simeq^{I_1 \cup I_2}_{O_1 \cup O_2} E_2, c_2;\ c_1}$$

In the rule above, $\mathsf{NotModify}(E, c, X)$ is a predicate expressing that variables in X are not modified by the command c in the environment E. It is semantically defined as follows:

$$\mathsf{NotModify}(E, c, X) \stackrel{\text{def}}{=} \forall\, f\ m.\ [\![E, c]\!]\ m\ f = [\![E, c]\!]\ m\ (\lambda\, m'.f\{X \leftarrow m\})$$

For mechanizing the application of the rules, we have implemented an algorithm computing a sound under approximation of the variables in a given set that are not modified by a portion of code (this is done by traversing the code and removing assigned variables from the set, except for self-assignments).

Our algorithm for performing dead code elimination removes portions of code that do not affect the output variables considered, so it actually behaves more like an slicing algorithm, but performs at the same time other transformations: branch predicting, self-assignment elimination and branch coalescing (replacing if e then c else c by c). Its correctness relies on the rule

$$\frac{\mathsf{NotModify}(E_1, c, X) \quad \mathsf{Lossless}(E_1, c) \quad \mathsf{fv}(\varphi) \subseteq X}{\vdash_\varphi E_1, c \simeq^{X}_{X} E_2, \mathsf{nil}}$$

We have also constructed certified algorithms for inlining procedure calls and performing a simple variant of register allocation. Together with the optimizations presented in the next paragraph, this results in a powerful tool for proving program transformations.

Transformations based on program analyses Many of the transformations appearing in game-based cryptographic proofs can be performed using common compiler optimizations, we have implemented and proved correct in Coq the most common ones: copy propagation, common subexpression elimination, constant propagation. As usual, program optimizations are done in two stages: first, an analysis of the program collects abstract values representing the compile time approximation of the different expressions contained in the program; then an optimizer uses this information to transform the program. All optimizations have been implemented in a generic way using the module system of Coq. 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), the functor automatically constructs the certified optimization function

$$\mathsf{optimize} : \mathcal{C} \to D \to \mathcal{C} \times D\ .$$

When given a command c and an element $d \in D$, this function transforms c into its optimized version c' assuming the validity of d. In addition, it returns an element $d' \in D$ which is valid after executing c (or c') and allows to recursively apply the optimization.

The technique used to prove the correctness of the optimizer is a mixture of the techniques presented in [29, 14] and [13]. For proving the correctness of an optimization, we first need to express the validity of the information contained in the analysis domain, i.e a predicate $\mathsf{Valid}(d, m)$ expressing the agreement between the compile time abstract values in d and the runtime memory m. Then, the correctness of the optimizer is expressed in terms of a pRHL judgment:

$$\forall d\ c.\ \textbf{let}\ (c', d') := \mathsf{optimize}(c, d)\ \textbf{in} \vdash E, c \sim E, c' : \asymp_d \Rightarrow \asymp_{d'}$$

where $m_1 \asymp_d m_2 \stackrel{\text{def}}{=} m_1 = m_2 \wedge \mathsf{Valid}(d, m_1)$. The following useful rule can be derived:

$$\frac{\forall m_1\ m_2.\ m_1\ \Psi\ m_2 \Rightarrow \mathsf{Valid}(d, m_1) \quad \mathsf{optimize}(c_1, d) = (c_1', d') \quad \vdash E_1, c_1' \sim E_2, c_2 : \Psi \Rightarrow \Phi}{\vdash E_1, c_1 \sim E_2, c_2 : \Psi \Rightarrow \Phi}$$

When implementing a classical analysis (not an inter-procedural analysis), the analysis loses all information about the value of global variables after a function call. In our context it is important to keep this information. We do so by reusing the same information that is used for transformations based on variable dependencies as presented above.

5.3 Building Automatic Tactics

The meta theory of the Coq theorem prover is based on the Calculus of Inductive Constructions, a functional programming language with dependent types. In this type system type programs are compared up to β-equivalence. In practice, this allows to replace deduction by computation. The idea is the following, to prove $\mathsf{prime}(17)$ one can first write a program test dividing its input n by all numbers between 2 and $n-1$, and returning true iff no division is exact, second prove the correctness of the test:

$$\forall n.\ \mathsf{test}(n) = \mathsf{true} \Rightarrow \mathsf{prime}(n)$$

Last, to prove $\mathsf{prime}(17)$, apply the correctness lemma to 17 and to a proof of the proposition $\mathsf{test}(17) = \mathsf{true}$. Since $\mathsf{test}(17)$ reduces to true, the proposition is β-equivalent to the proposition $\mathsf{true} = \mathsf{true}$ which is trivial. This proof technique is called proof by reflection and has been used for many applications [23, 24, 25]. CertiCrypt is strongly based on this proof technique. Most of the tactics provided to the user boil down to the application of the correctness lemmas of one or more of the certified algorithms we have constructed.

6. Reasoning about Adversaries

The notion of adversary is generally left implicit in cryptographic proofs. In the literature, most of the time the only condition imposed to adversaries is that their (expected) running time is bounded by a polynomial on the security parameter. In some cases, extra conditions forbidding repeated or malformed queries to oracles are imposed. In any case, little care is taken to explicitly state the access rights of adversaries to global variables or to procedures. However, these issues cannot be ignored if one wants to formally justify security; for instance, if security relies on a secret shared

among the oracles using a global variable, any adversary with the right to access this variable may trivially break security.

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 in the type system:

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

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

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

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

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

where $\mathsf{Writable}(x) \stackrel{\text{def}}{=} \mathsf{Local}(x) \vee x \in \mathcal{G}_{\mathcal{A}}$, and $\mathcal{A}_E$ and o_E stand for $E(\mathcal{A})$ and $E(o)$ respectively. The rules above 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 the adversary has the right to read or a local variable previously initialized. In every derivable judgment of the form $I \vdash c : O$, it is the case that $I \subseteq O$.

In order to apply the mechanized tactics described in the previous sections in the presence of unknown adversaries, we need to provide information about the adversaries in the same way as we do for other procedures. For oracles, and in the absence of recursive calls, this information can be constructed incrementally and in an automatic way by reusing the functions presented in Sec. 5.2. Once the information for the oracles in $\mathcal{O}$ is computed, we provide a function that computes the needed information for a well-formed adversary (the code of the adversary should be the same in both environments). This is achieved by taking $\bigcap_{o \in \mathcal{O}} \mathsf{NotModify}(o) \setminus \mathcal{G}_{\mathcal{A}}$ as the set of variables known to be not modified by the adversary. In addition, we build a triple $(I_1, I_{\mathcal{A}}, O_{\mathcal{A}})$ satisfying the specification given in Sec. 5. We take I_1 as the whole set of $\mathcal{A}$'s actual parameters. The needed input global variables $I_{\mathcal{A}}$ should contain at least $\mathcal{G}_{\mathcal{A}} \cup \mathcal{G}_{\mathrm{ro}}$ (the adversary may depend on them) as well as the set of needed input global variables of the oracles that may be called by the adversary, since the result of the oracle—and thus of the adversary—may depend on these global variables. In summary, it suffices to take

$$I_{\mathcal{A}} \stackrel{\text{def}}{=} \bigcup_{o \in \mathcal{O}} I_o \cup \mathcal{G}_{\mathcal{A}} \cup \mathcal{G}_{\mathrm{ro}}$$

Note that, because we make absolutely no assumptions about the order in which the adversary calls the oracles, for each variable $x \in I_{\mathcal{A}}$, we cannot do better than requiring every oracle to either not modify x, or to ensure equality on x after its execution in both environments. This way, we can guarantee that after an arbitrary sequence of calls to the oracles, equality over $I_{\mathcal{A}}$ is preserved.

When quantifying over an adversary $\mathcal{A}$ in a cryptographic statement, we therefore assume its well-formedness as well as that its running time is polynomial in the security parameter under the hypothesis that the oracles in the environment are so.

7. A More Elaborated Example: OAEP

The most advanced application of CertiCrypt is a proof of security of OAEP, a widely used padding scheme whose history perfectly illustrates the difficulty in achieving a correct proof. Indeed, it was initially believed that OAEP was IND-CCA2 secure [11], but it was later discovered it was only IND-CCA1 secure [35], a weaker security notion. However, in combination with a well chosen encryption scheme, it is possible to recover IND-CCA2 security, as it is the case for RSA-OAEP [21].

OAEP is parametrized by an encryption scheme whose key generator $\mathcal{F}$ is also used as key generator for OAEP. This encryption scheme must act as a permutation. To achieve such a high level of security, OAEP adds randomness into the plaintext and uses two functions G and H to mask it before applying a non-invertible encryption scheme such as RSA.

We consider the proof that OAEP is IND-CPA secure in the random oracle model [10]. In this model, one-way functions are simulated by oracles that log their responses to previous queries, and when given a new query answer with a randomly sampled value. The two functions G and H used in OAEP are represented by such random oracles.

In [12] it is shown that, if the probability of inverting a randomly sampled encryption key f is negligible, then OAEP is IND-CPA secure. This means that, if the underlying encryption primitive used by OAEP ensures that without knowing the decryption key, one cannot obtain the whole plaintext given its ciphertext, then using OAEP it is unfeasible to obtain information even about portions of the plaintext.

Theorem 1 (OAEP semantic security)**.**

$$\begin{aligned}\forall \mathcal{A}\ \mathcal{A}'\ M.\ & E_{\mathit{OAEP}} \vdash_{\mathrm{wf}} \mathcal{A} \wedge E_{\mathit{OAEP}} \vdash_{\mathrm{wf}} \mathcal{A}' \wedge \\ & \mathsf{PPT}(\mathcal{A}, \eta) \wedge \mathsf{PPT}(\mathcal{A}', \eta) \\ & \Rightarrow M \vdash \mathit{OAEP}_0 \approx_{[d=1]} \mathit{OAEP}_1\end{aligned}$$

where $M : \mathbb{N} \to \mathcal{M}$ is a family of polynomially bounded memories indexed by the security parameter η, and E_{OAEP} is the environment of the two initial games OAEP_0 and OAEP_1 (not shown in this paper.)

Proof This proof exemplifies nearly all the game transitions presented in [12] and is more involved than most proofs appearing in the literature. The whole proof consists of a main sequence of nearly 30 transitions, some of them being justified by another sequence of transitions of their own. Due to space constraints, we do not fully present the proof here, but we instead explain in detail two non-trivial transformations. At some point in the proof, after introducing a failure event and applying Lemma 2, we become interested in proving that the probability of the event $(\mathsf{bad} = \mathsf{true})$ in a certain game $init_{15}$ is negligible. To prove this, it is convenient to first transform the game into a simpler one, were it is easier to bound this probability. Figure 5 illustrates the proof of the first simplifying transition, where we prove the equivalence w.r.t. bad of the games $init_{15}$ and $init_{16}$. The overall objective of the transition is to remove (S', HS') from the initialization of L_H in the main code, and to modify accordingly the code of H to preserve the same behavior by returning directly HS' upon a query S'. In the justification of this transition most transformations are done in the oracle H; we present its various versions in the figure. Although, the bad flag is only set in the G oracle, we do not present it here because it remains unmodified in every step of the sequence. Global variables S' and HS' correspond to a query and its response, respectively, made to oracle H at the beginning of the game. L_H is a global variable representing a log of the queries made to H and their corresponding answers, and hence in the two games the value of L_H is not necessarily the same. In particular, if the adversary never makes a query S' to H, the pair (S', HS') will not appear in L_H in game $init_{16}$.

In order to prove the transition, we first introduce an intermediate game containing a *ghost* variable L'_H, representing the variable L_H in $(init_{15}, H_{15})$. In the second transformation we replace occurrences of L_H by L'_H using the invariant $L_H = L'_H$. Once

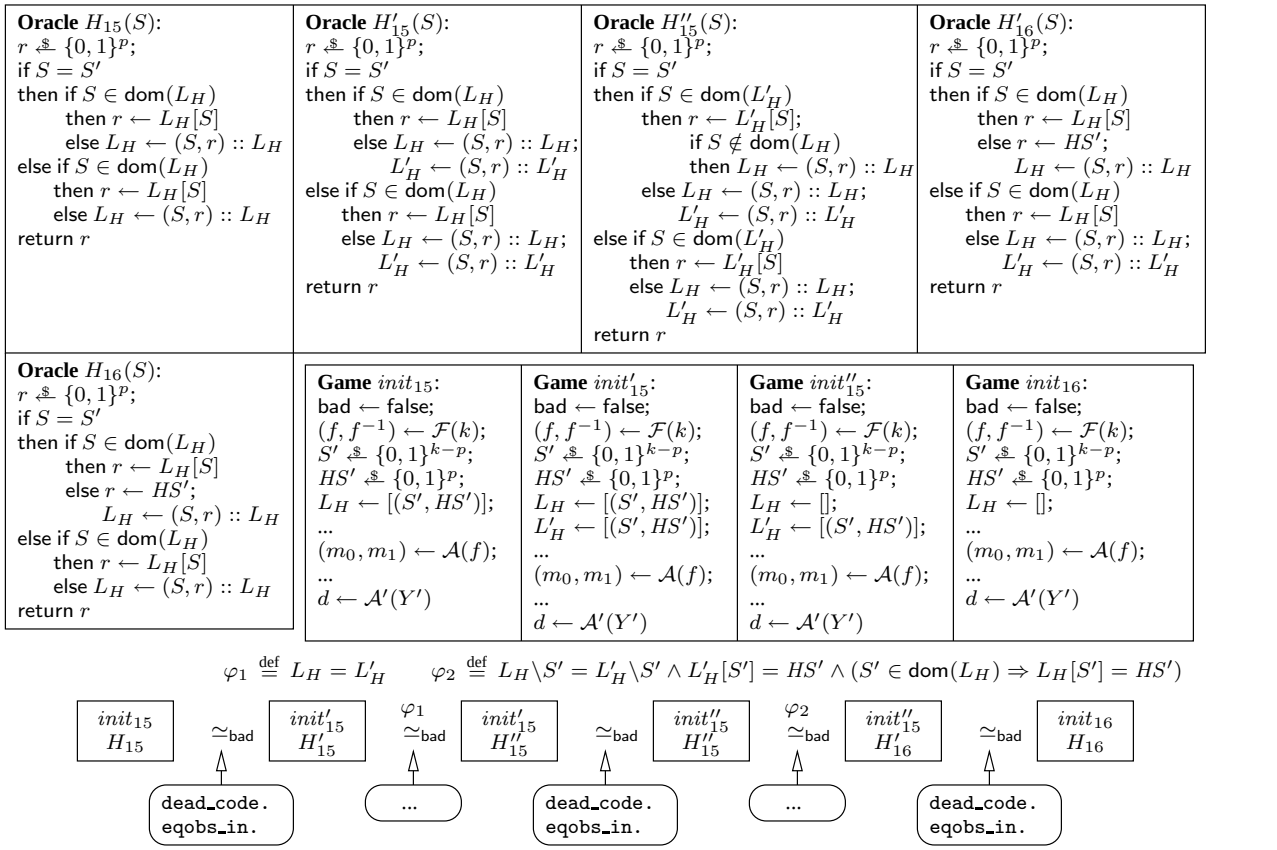

Figure 5. The proof of a transition for OAEP

this is done, we are able to modify the initialization of L_H in $(init''_{15}, H''_{15})$ while still preserving the overall behavior, since the output of H is now independent from L_H. We then prove the invariant φ_2 defined in Fig. 5, which states that L_H and L'_H map queries to the same values except maybe for S'. This allows us to replace occurrences of L'_H by L_H and modify oracle H accordingly to preserve its behavior. The purpose of the last transition is simply to *clean* the game by removing the ghost variable L'_H.

Although we have described in CertiCrypt all the transitions in the proof of OAEP semantic security, we have not proved all of them: two transitions involving interprocedural code-motion techniques not yet implemented in CertiCrypt remain unproved. Nevertheless, we believe that the effort needed to implement them in CertiCrypt is not significant and we expect to do it soon.

8. Related Work

For clarity, we distinguish between verification tools and methods for cryptographic proofs, and relevant formalizations that have not been developed on purpose for cryptographic proofs.

Dedicated tools Most dedicated tools focus on verifying protocols in the symbolic model, there are only a few tools that provide guarantees w.r.t. the computational model.

CryptoVerif [15] is a dedicated tool developed by Blanchet to support game-based proofs; it was initially developed for protocol verification but was later applied to proofs of cryptographic schemes, e.g. the unforgeability of the FDH signature scheme assuming the existence of one-way permutations [16]. CryptoVerif performs an heuristic-based search on a library of (user-provided and predefined) transformations to try to generate a sequence of games for a proof, but gives little attention to whether the transformations are computationally sound, while our framework relies on the user to supply the sequence of games but instead put the emphasis on verifying that the whole proof is computationally sound. We believe the two approaches are complementary and can benefit from each other: compiling CryptoVerif sequences of games to CertiCrypt is an interesting research direction.

Backes and Laud [7] have developed a dedicated tool to mechanically analyze protocols in the Backes-Pfitzmann-Waidner cryptographic library [32, 8] against simulatability-based security conditions. They use type-based program analyses for a process algebra inspired from the spi calculus [1] and prove automatically the correctness of a number of protocols from the literature. One important difference with CertiCrypt (and CryptoVerif) is that the tool of Backes and Laud provides a proof of security in a symbolic rather than computational model; however, one can derive a complexity-theoretical guarantee by appealing to the soundness of the symbolic model. Similarly, several formalizations [38, 6] of the symbolic model of BPW cryptographic library have been achieved in various theorem provers, but without proving computational soundness.

Methods In addition to the above tools, there are recent proposals of formalisms to prove security of cryptographic protocols, so far lacking tool support. Roy, Datta, Derek and Mitchell [33] have identified conditions under which trace properties ensure computational guarantees for secrecy and authentication properties. With this approach, proofs can be done by induction. The authors report that their technique is powerful enough to apply it to Kerberos and

IKE. Corin and den Hartog [19] developed a probabilistic Hoare logic, which is more expressive than the simple (not relational) Hoare logic we use to reason about programs, and used it to prove ElGamal semantic security; no tool support nor further examples have been presented since then. As said in Sec. 2, their logic by itself is not sufficient to carry arbitrary code-based proofs.

Machine-checked libraries CertiCrypt relies on diverse mathematical concepts and theories that have been modeled for their own sake, including probabilities, group theory, polynomials, programming languages semantics, program equivalence, Hoare logics and generation of verification conditions, and compiler optimizations. It is not possible to review all the relevant work here, but we focus on the most directly related formalizations. The most remarkable, is the Coq library for representing (sub-)probability distributions of Paulin-Mohring and Audebaud [5] which we reuse to construct our framework. Hurd et al. [28] developed a mechanized theory in the HOL theorem prover for reasoning about pGCL programs, a probabilistic extension of Dijkstra's guarded command language presented in McIver and Morgan [30]. Their work focuses on using a weakest (liberal) precondition calculus for proving partial correctness properties expressed as Hoare triples in a probabilistic logic. Despite having very strong theoretical foundations, the framework is less appealing for carrying program transformations that require a relational Hoare logic rather than a standard one.

9. Conclusion and Future work

We have developed CertiCrypt, a fully formalized framework for machine-checked game-based proofs, and applied it to prove the PRF/PRP switching lemma, and IND-CPA security of ElGamal and OAEP encryption schemes. About 6000 lines of Coq in the development are dedicated to formalize the semantics and deriving the lemmas corresponding to the rules of pRHL; roughly 10000 lines to defining and proving correct the reflection-based tactics; about 700 to the proof of ElGamal IND-CPA security; and 5000 lines for the proof of security of OAEP. It is worth noticing that in these two last proofs, more than one third of the lines are spent just in defining the sequence of games. The complete Coq development is available at `http://www-sop.inria.fr/everest/certicrypt`.

CertiCrypt is the most advanced tool of its kind, and constitutes a significant first step towards the completion of Halevi's programme. Nevertheless, numerous research directions remain to be explored.

Code-based game proofs Our most immediate priority is to enhance proof automation: while our framework already provides automated support for semantics-preserving transformations, and the fundamental lemma of game-playing, we still need to develop automated tactics to compute the complexity of a program and bound the probability of an event in a final game. In order to assess the benefits of the tactics we develop, we intend to machine-check the proof of an exact bound for the security of 3DES [12].

One of our priorities in making CertiCrypt more user-friendly is to add an extensive library of lemmas for proving transformations based on algebraic properties (e.g. equational theory for cyclic groups), but also more complex transformations, such as transformations strongly related to the random oracle model (e.g. those presented in Sec. 7). For some specific transformations in our framework, we need to introduce intermediate games. We believe that these games could be constructed automatically in most cases. In the long term, a (minimalist) interface to ease the writing of games and their corresponding proofs should be developed. Based on a user-defined sequence of games, this interface should be able to automatically generate the skeleton of a proof since the proof of each transition follows the same schema: build the information for the procedures in the environments, prove its correctness, and then prove the transition using automated tactics.

In parallel, we are currently working on integrating arrays within the framework. Dealing with arrays involves a major technical hurdle: inference of frame conditions, i.e. the parts of the memory that are modified by a program, becomes more difficult to achieve—precise and automated inference of frame conditions is essential for ensuring an appropriate level of automation. The extension to arrays, and its application to Halevi and Rogaway's tweakable enciphering scheme, are currently under development, and will be reported elsewhere.

Computational soundness of symbolic cryptography The focus of our work is to use a general purpose proof assistant to verify cryptographic proofs in the computational model. Alternatively, one can develop dedicated tools to perform proofs automatically; to the exception of CryptoVerif, such tools operate on the symbolic model, that abstracts from the computational model by assuming perfect cryptography, i.e. in the case of encryption, that it is not possible to extract a plaintext from a ciphertext without the decryption key. The symbolic model disposes of effective decision procedures to reason about protocols, and has been proven sound w.r.t. the computational model under the assumption that the cryptographic primitives are sufficiently secure. A last objective for future work is to machine-check soundness proofs, both because they are complex and error-prone, and also because a machine-checked soundness proof could be used in conjunction with a reflective implementation of decision procedures at the symbolic level to generate correctness proofs of protocols in the computational model. A first step in this direction has been taken recently by Corin [18], who provides a machine-checked proof in Coq of the soundness result in the seminal work of Abadi and Rogaway [2].

Language-based security Language based security is an active field of research that attempts to achieve security and counter application-level attacks at the level of programming languages. Language-based security advocates a rigorous definition of the security goal based on the semantics of programs, and commonly focuses on non-interference [34], an information flow property that guarantees the absence of illicit information leakage through program execution. A common means to enforce non-interference is through an information flow type system, and many such systems have been developed for complex calculi and languages. However, non-interference is too strong a requirement in practice, as many applications intentionally release information. Computational language based security is an extension of language based security that studies information flow properties in presence of cryptographic primitives and reconciles programming language security with the computational model used by cryptographers. Its focus is computational non-interference, a generalization of non-interference that allows to leak secret sensitive data after it has been encrypted, provided the underlying encryption scheme is secure. As with non-interference, type systems are the prominent means to enforce computational information flow. We believe that our framework is sufficiently rich to yield machine-checked proofs of soundness for the information flow type system of Smith and Alpízar [3].

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

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

[3] R. Alpízar and G. Smith. Secure information flow with random assignment and encryption. In *Proceedings of the 4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–44, 2006.

[4] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 243–253, 2000.

[5] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. In *Mathematics of Program Construction*, volume 4014 of *LNCS*, 2006.

[6] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *20th Symposium on Theoretical Aspects of Computer Science*, pages 675–686, 2003.

[7] 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, 2006.

[8] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *ACM Conference on Computer and Communications Security*, pages 220–230, 2003.

[9] 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, 2004.

[10] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[11] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT'94, Proc.*, pages 92–111, 1994.

[12] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Proceedings of the 25th International Cryptology Conference*, volume 4004 of *LNCS*, pages 409–426, 2006.

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

[14] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *International Types for Proofs and Programsi, Workshop*, LNCS, pages 66–81, 2006.

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

[16] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Proceeding of the 26th Annual International Cryptology Conference*, pages 537–554, 2006.

[17] D. Boneh. The decision Diffie-Hellman problem. In *Proccedings of the 3rd Algorithmic Number Theory Symposium*, volume 1423 of *LNCS*, pages 48–63. Springer, 1998.

[18] R. Corin. Computational soundness of formal encryption in Coq. In *Informal Proceedings of the 3rd Workshop on Formal and Computational Cryptography*, 2007. To appear.

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

[20] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of the 4th Annual International Cryptology Conference*, pages 10–18, 1985.

[21] 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.

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

[23] G. Gonthier. A computer-checked proof of the Four Colour Theorem. `http://research.microsoft.com/~gonthier`.

[24] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 98–113, 2005.

[25] B. Grégoire, L. Thery, and B. Werner. A computational approach to Pocklington certificates in type theory. In *Proceedings of the 8th Symposium on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 97–113, 2006.

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

[27] S. Halevi and P. Rogaway. A tweakable enciphering model. In *Proceeding of the 23th Annual International Cryptology Conference*, pages 482–499, 2003.

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

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

[30] A. McIver and C. Morgan. *Abstraction, Refinement, and Proof for Probabilistic Systems*. Springer, 2005.

[31] D. Nowak. A framework for game-based security proofs. Cryptology ePrint Archive, Report 2007/199, 2007.

[32] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *ACM Conference on Computer and Communications Security*, pages 245–254, 2000.

[33] A. Roy, A. Datta, A. Derek, and J. Mitchell. Inductive proof method for computational secrecy. Cryptology ePrint Archive, Report 2007/165, 2007.

[34] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, pages 5–19, 2003.

[35] V. Shoup. OAEP reconsidered. In *Proceeding of the 21th Annual International Cryptology Conference*, pages 239–259, 2001.

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

[37] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 355–364, 1998.

[38] C. Sprenger, M. Backes, D. A. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 153–166, 2006.

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