Formal certification of game-based cryptographic proofs

Santiago Zanella-Béguelin

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°84 :
Sciences et technologies de l'information et de la communication

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Santiago José ZANELLA BÉGUELIN

le 9 décembre 2010

Certification Formelle de Preuves Cryptographiques Basées sur les Séquences de Jeux

---

Formal Certification of Game-Based Cryptographic Proofs

Directeur de thèse : **Gilles BARTHE**

T H È S E

MINES ParisTech
Centre de Mathématiques Appliquées
Rue Claude Daunesse B.P. 207, 06904 Sophia Antipolis Cedex, France

Résumé

Les séquences de jeux sont une méthodologie établie pour structurer les preuves cryptographiques. De telles preuves peuvent être formalisées rigoureusement en regardant les jeux comme des programmes probabilistes et en utilisant des méthodes de vérification de programmes. Cette thèse décrit CertiCrypt, un outil permettant la construction et vérification automatique de preuves basées sur les jeux. CertiCrypt est implementé dans l'assistant à la preuve Coq, et repose sur de nombreux domaines, en particulier les probabilités, la complexité, l'algèbre, et la sémantique des langages de programmation. CertiCrypt fournit des outils certifiés pour raisonner sur l'équivalence de programmes probabilistes, en particulier une logique de Hoare relationnelle, une théorie équationnelle pour l'équivalence observationnelle, une bibliothèque de transformations de programme, et des techniques propres aux preuves cryptographiques, permettant de raisonner sur les évènements. Nous validons l'outil en formalisant les preuves de sécurité de plusieurs exemples emblématiques, notamment le schéma de chiffrement OAEP et le schéma de signature FDH.

Abstract

The game-based approach is a popular methodology for structuring cryptographic proofs as sequences of games. Game-based proofs can be rigorously formalized by taking a code-centric view of games as probabilistic programs and relying on programming language techniques to justify proof steps. In this dissertation we present CertiCrypt, a framework that enables the machine-checked construction and verification of game-based cryptographic proofs. CertiCrypt is built upon the general-purpose proof assistant Coq, from which it inherits the ability to provide independently verifiable evidence that proofs are correct, and draws on many areas, including probability and complexity theory, algebra, and semantics of programming languages. The framework provides certified tools to reason about the equivalence of probabilistic programs, including a relational Hoare logic, a theory of observational equivalence, verified program transformations, and ad-hoc programming language techniques of particular interest in cryptographic proofs, such as reasoning about failure events. We validate our framework through the formalization of several significant case studies, including proofs of security of the Optimal Asymmetric Encryption Padding scheme against adaptive chosen-ciphertext attacks, and of existential unforgeability of Full-Domain Hash signatures.

Disclaimer

This dissertation builds on several published works that I co-authored. In chronological order,

- *Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt.*
  With Gilles Barthe, Benjamin Grégoire and Sylvain Heraud.
  In proceedings of the *5th International workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2009. Springer-Verlag.

- *Formal certification of code-based cryptographic proofs.*
  With Gilles Barthe and Benjamin Grégoire.
  In proceedings of the *36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2009*, pages 90–101, New York, 2009. ACM Press.

- *Formally certifying the security of digital signature schemes.*
  With Gilles Barthe, Benjamin Grégoire and Federico Olmedo.
  In proceedings of the *30th IEEE symposium on Security and Privacy, S&P 2009*, pages 237–250, Los Alamitos, California, 2009. IEEE Computer Society.

- *Programming language techniques for cryptographic proofs.*
  With Gilles Barthe and Benjamin Grégoire.
  In proceedings of the *1st International conference on Interactive Theorem Proving, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 115–130, Berlin, 2010. Springer-Verlag.

- *A machine-checked formalization of Sigma-protocols.*
  With Gilles Barthe, Benjamin Grégoire, Sylvain Heraud and Daniel Hedin.
  In proceedings of the *23rd IEEE Computer Security Foundations symposium, CSF 2010*, pages 246–260, Los Alamitos, California, 2010. IEEE Computer Society.

- *Beyond Provable Security. Verifiable IND-CCA Security of OAEP.*
  With Gilles Barthe, Benjamin Grégoire and Yassine Lakhnech.
  To appear in Topics in Cryptology – CT-RSA 2011, The Cryptographers' Track at the RSA Conference 2011.

I contributed in the elaboration of all of them, as well as in the development of the supporting machine-checked proofs.

A Coq development accompanying this dissertation is available upon request.

Contents

1 Introduction

DESIGNING secure cryptographic systems is a notoriously difficult task. Indeed, the history of modern cryptography is fraught with examples of cryptographic systems that had been thought secure for a long time before being broken and with flawed security proofs that stood unchallenged for years. Provable security [Goldwasser and Micali 1984; Stern 2003] is an approach that aims to establish the security of cryptographic systems through a rigorous analysis in the form of a mathematical proof, borrowing techniques from complexity theory. In a typical provable security argument, security is proved by reduction, showing that any attack against the security of the system would lead to an efficient way to solve some computationally hard problem.

Provable security holds the promise of delivering strong guarantees that cryptographic schemes meet their goals and is becoming unavoidable in the design and evaluation of new schemes. Yet provable security *per se* does not provide specific tools for managing the complexity of proofs and as a result several purported security arguments that followed the approach have been shown to be flawed. Consequently, the cryptographic community is increasingly aware of the necessity of developing methodologies that systematize the type of reasoning that pervade cryptographic proofs, and that guarantee that such reasoning is applied correctly.

One prominent method for achieving a high degree of confidence in cryptographic proofs is to cast security as a program verification problem: this is achieved by formulating goals and hypotheses in terms of probabilistic programs, and defining the adversarial model in terms of complexity classes, e.g. probabilistic polynomial-time programs. This code-centric view leads to statements that are unambiguous and amenable to formalization. However, standard methods to verify programs (e.g. in terms of program logics) are ineffective to directly address the kind of verification goals that arise from cryptographic statements. The game-based approach [Bellare and Rogaway 2006; Halevi 2005; Shoup 2004] is an alternative to standard program verification methods that establishes the verification goal through successive program transformations. In a nutshell, a game-based proof is structured as a sequence of transformations of the form $G, A \rightarrow^h G', A'$, where G and G' are probabilistic programs, A and A' are events, and h is a monotonic function such that $\Pr[G : A] \leq h(\Pr[G' : A'])$. When the security of a scheme is expressed as an inequality of the form $\Pr[G_0 : A_0] \leq p$, it can be proved by exhibiting a sequence of transformations

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

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

1.1 The CertiCrypt Framework

Whereas Bellare and Rogaway [2006] already observed that code-based proofs could be more easily amenable to machine-checking, Halevi [2005] argued that formal verification techniques should be used to improve trust in cryptographic proofs, and set up a program for building a tool that could be used by the cryptographic community to mechanize their proofs. We take a first step towards Halevi's ambitious program by presenting CertiCrypt [Barthe et al. 2009c], a fully machine-checked framework for constructing and verifying game-based cryptographic proofs. CertiCrypt builds on top of the Coq proof assistant [The Coq development team 2009] a broad set of reasoning principles used by cryptographers, drawing on program verification, algebraic reasoning, and probability and complexity theory. The most notable features of the framework are:

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

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

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

Powerful and automated reasoning methods. We formalize a relational Hoare logic and a theory of observational equivalence, and use them as stepping stones to support the main tools of code-based reasoning. In particular, we prove that many transformations used in code-based proofs, including common optimizations, are semantics-preserving. In addition, we mechanize reasoning patterns used ubiquitously in cryptographic proofs, such as reasoning about failure events (the so-called fundamental lemma of game-playing), and a logic for inter-procedural code-motion (used to justify the eager/lazy sampling of random values).

1.2 Organization of the Dissertation

The purpose of this dissertation is to provide a high-level description of the CertiCrypt framework, overview the case studies that have been formalized, and stir further interest in machine-checked cryptographic proofs. The rest of the dissertation is organized as follows:

- In the rest of this chapter we briefly discuss the motivation behind formal proofs and the features of modern proof assistants. We then present two introductory examples of game-based proofs, namely the semantic security of the ElGamal and Hashed ElGamal encryption schemes. These examples, although simple, nicely illustrate the kind of verification problem that we study and the techniques that we use to mechanize the construction of proofs.

- In Chapter 2 we overview the mathematical background behind our formalization and present the probabilistic language that we use to describe games and its semantics. We also discuss our model of adversaries and the notions of complexity and termination of programs in a probabilistic setting;
- In Chapter 3 we present the probabilistic relational Hoare logic that constitutes the core of the framework, and describe the formulation and mechanization of game transformations. We describe as well two ubiquitous reasoning patterns of cryptographic proofs and specializations of the relational Hoare logic that can be used to automate them;
- In Chapter 4 we give a detailed description of two different formalizations of the PRP/PRF switching lemma, an important result in cryptography that admits an elegant proof using games;
- In Chapter 5 we describe two different machine-checked proofs of the existential unforgeability against chosen-message attacks of the Full-Domain Hash digital signature scheme. We compare the resulting security bounds and discuss the practical importance of tight reductions and the role that exact security plays in choosing adequate parameters when instantiating schemes;
- In Chapter 6 we describe in some detail a machine-checked proof of the semantic security of the Optimal Asymmetric Encryption Padding scheme against chosen-plaintext attacks, and we report on a significantly more challenging proof of the security of the same scheme under adaptive chosen-ciphertext attacks;
- In Chapter 7 we overview a machine-checked theory of a large class of zero-knowledge protocols. We illustrate how to use this theory to obtain short proofs of several well-known zero-knowledge protocols from the literature;
- We conclude in Chapter 8 with a survey of related work in the area, a discussion of the lessons we learned while building CertiCrypt and perspectives to improve automation and further this line of research.

1.3 A Primer on Formal Proofs

Proof assistants are programs designed to support interactive construction and automatic verification of mathematical statements (understood in a broad sense). Initially developed by logicians to experiment with the expressive power of their foundational formalisms, proof assistants are now emerging as a mature technology that can be used effectively for verifying intricate mathematical proofs, such as the Four Color theorem [Gonthier 2008] or the Kepler conjecture [Hales 2008; Hales et al. 2010], or complex software systems, such as operating systems [Klein et al. 2009], virtual machines [Klein and Nipkow 2006] and optimizing compilers [Leroy 2006]. In the realm of cryptography, proof assistants have been used to formally verify secrecy and authenticity properties of protocols [Paulson 1998].

Proof assistants rely on expressive specification languages that allow formalizing arbitrary mathematical notions, and that provide a formal representation of proofs as proof objects. Their architecture is organized into two layers: a kernel, and a proof engine.

- The kernel is the cornerstone for correctness. Its central component is a checker for verifying the consistency of formal theories, including definitions and proofs. In particular, the checker guarantees that definitions and proofs are well-typed, that there are no missing cases or undefined notions in definitions, and that all proofs are built from valid elementary logical steps and make a correct use of assumptions.
- In contrast, the proof engine assists proof construction. The proof engine embraces a variety of tools. The primary tools are a set of pre-defined tactics; a language for writing user-defined tactics is usually provided. Tactics allow to reduce a proof goal to simpler ones. When invoked on a proof goal Q, a tactic will compute a new set of goals $P_1 \dots P_n$, and a proof that $P_1 \wedge \dots \wedge P_n \implies Q$. At the end of each demonstration, the proof engine outputs a proof object.

Proof objects are independently checked by the kernel. Therefore, the proof engine needs not be trusted, and the validity of a formal proof—beyond the accuracy of the statement itself—only depends on the correctness of the kernel. Pleasingly, kernels are extremely reliable programs with restricted functionalities and solid logical foundations.

As with any other mathematical activity, formal proofs strive for elegance and conciseness. In our experience, they also provide a natural setting for improving proofs—in the case of cryptography, improvement can be measured by comparing exact security bounds. Yet, what matters most about a formal proof is that it provides a nearly absolute degree of guarantee, without requiring expensive human verification.

1.4 Introductory Examples

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

Definition 1.1 (Asymmetric encryption scheme). *An asymmetric encryption scheme is composed of a triple of algorithms:*

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

Encryption*:*	*Given a public key pk and a plaintext m, the encryption algorithm* $\mathcal{E}(pk, m)$ *computes a ciphertext corresponding to the encryption of m under pk;*
Decryption:	*Given a secret key sk and a ciphertext c, the decryption algorithm* $\mathcal{D}(sk, c)$ *returns either the plaintext corresponding to the decryption of c, if it is a valid ciphertext, or a distinguished value* $\perp$ *otherwise.*

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

An asymmetric encryption scheme is said to be semantically secure if it is unfeasible to gain significant information about a plaintext given only a corresponding ciphertext and the public key. Goldwasser and Micali [1984] showed that semantic security is equivalent to the property of ciphertext indistinguishability under chosen-plaintext attacks (IND-CPA, for short). This property can be formally defined in terms of a game played between a challenger and an adversary $\mathcal{A}$, represented as a pair of procedures $(\mathcal{A}_1, \mathcal{A}_2)$ that may share state:

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

In this game, the challenger first generates a new key pair and gives the public key pk to the adversary, who returns two plaintexts m_0, m_1 of its choice. The challenger then tosses a fair coin b and gives the encryption of m_b back to the adversary, whose goal is to guess which message has been encrypted.

Definition 1.2 (IND-CPA security). *The advantage of an adversary* $\mathcal{A}$ *in the above experiment is defined as*

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

An encryption scheme is said to be IND-CPA secure if the advantage of any efficient adversary is a negligible function of the security parameter η*, i.e., the adversary cannot do much better than a blind guess.*

Definition 1.3 (Negligible function). *A function* $\nu : \mathbb{N} \to \mathbb{R}$ *is said to be negligible if it decreases asymptotically faster than the inverse of any polynomial:*

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

Note that in order to satisfy the above definition, an encryption scheme must necessarily be probabilistic, otherwise an adversary could trivially detect to which message corresponds the challenge ciphertext by simply encrypting one of the messages it has chosen and comparing the resulting ciphertext with the challenge ciphertext.

1.4.1 The ElGamal Encryption Scheme

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

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

We prove the IND-CPA security of ElGamal encryption under the assumption that the Decision Diffie-Hellman (DDH) problem is hard. Intuitively, for a family of finite cyclic groups, the DDH problem consists in distinguishing between triples of the form (g^x, g^y, g^{xy}) and triples of the form (g^x, g^y, g^z) , where the exponents x, y, z are uniformly sampled from $\mathbb{Z}_q$. One characteristic of game-based proofs is to formulate computational assumptions using games; the assumption that the DDH problem is hard is no exception and can be formulated as follows:

Definition 1.4 (Decision Diffie-Hellman assumption). *Consider the following games*

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

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

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

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

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

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

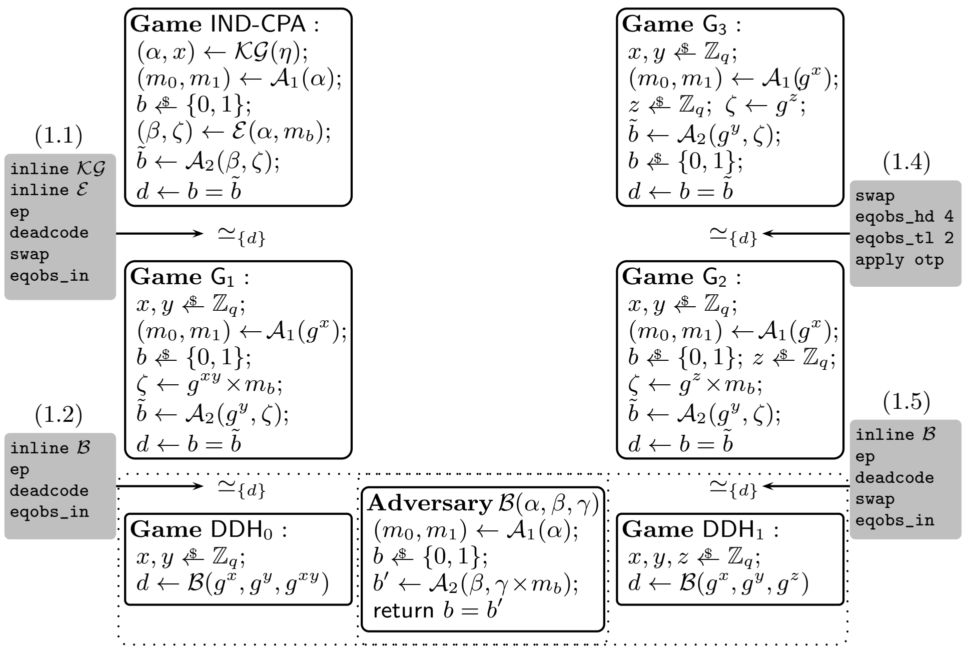
We prove that $\mathbf{Adv}_{\mathcal{B}}^{\mathsf{DDH}} = \mathbf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}$ for any given adversary $\mathcal{A}$. To conclude the proof (i.e. to show that the advantage of any efficient adversary $\mathcal{A}$ is negligible), we show that the reduction is efficient: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

Figure 1.1 gives a high-level view of the reduction: games appear inside white background boxes, whereas gray background boxes contain the actual proof scripts used to prove observational equivalence between consecutive games. A proof script is simply a sequence of tactics, each intermediate tactic transforms the current goal into a simpler one, whereas the last tactic in the script ultimately solves the goal. The tactics that appear in the figure hopefully have self-explanatory names, but are explained cursorily below and in more detail in Chapter 3.

The proof proceeds by constructing an adversary $\mathcal{B}$ against DDH such that the distribution of $b = \tilde{b}$ (equivalently, d) after running the IND-CPA game for ElGamal is exactly the same as the distribution obtained by running game DDH_0. In addition, we show that the probability of d being `true` in DDH_1 is exactly 1/2 for the same adversary $\mathcal{B}$. The remaining gap between DDH_0 and DDH_1 is the DDH-advantage of $\mathcal{B}$. The reduction is summarized by the following equations:

$$\begin{aligned}
\left|\Pr\left[\mathsf{IND\text{-}CPA} : b = \tilde{b}\right] - 1/2\right| &= |\Pr[\mathsf{G}_1 : d] - 1/2| && (1.1)\\
&= |\Pr[\mathsf{DDH}_0 : d] - 1/2| && (1.2)\\
&= |\Pr[\mathsf{DDH}_0 : d] - \Pr[\mathsf{G}_3 : d]| && (1.3)\\
&= |\Pr[\mathsf{DDH}_0 : d] - \Pr[\mathsf{G}_2 : d]| && (1.4)\\
&= |\Pr[\mathsf{DDH}_0 : d] - \Pr[\mathsf{DDH}_1 : d]| && (1.5)
\end{aligned}$$

Equation (1.1) holds because games IND-CPA and G_1 induce the same distribution on d. We specify this as an observational equivalence judgment as $\mathsf{IND\text{-}CPA} \simeq_{\{d\}} \mathsf{G}_1$, and prove it using certified program transformations and decision procedures. A graphical representation of the sequence of tactics used to prove this judgment is shown in Figure 1.2. We first inline the procedure calls to $\mathcal{KG}$ and $\mathcal{E}$ in the IND-CPA game and simplify the resulting games by propagating assignments and eliminating dead code (tactics `ep`, `deadcode`). At this point we are left with two games almost identical, except that y is sampled later in one game than in the other. The tactic `swap` hoists instructions in one game whenever is possible in order to obtain a maximal common prefix with another game, and allows us to hoist the sampling of y in the program on the left hand side. We conclude the proof by applying the tactic `eqobs_in` that decides observational equivalence of a program with itself.



```
Game IND-CPA :
  $(\alpha, x) \leftarrow \mathcal{KG}(\eta)$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $(\beta, \zeta) \leftarrow \mathcal{E}(\alpha, m_b)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\beta, \zeta)$;
  $d \leftarrow b = \tilde{b}$
```

```
Game G1 :
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\zeta \leftarrow g^{xy} \times m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, \zeta)$;
  $d \leftarrow b = \tilde{b}$
```

```
Game DDH0 :
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^{xy})$
```

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

```
Game G3 :
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, \zeta)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $d \leftarrow b = \tilde{b}$
```

```
Game G2 :
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$; $z \xleftarrow{\$} \mathbb{Z}_q$;
  $\zeta \leftarrow g^z \times m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, \zeta)$;
  $d \leftarrow b = \tilde{b}$
```

```
Game DDH1 :
  $x, y, z \xleftarrow{\$} \mathbb{Z}_q$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^z)$
```

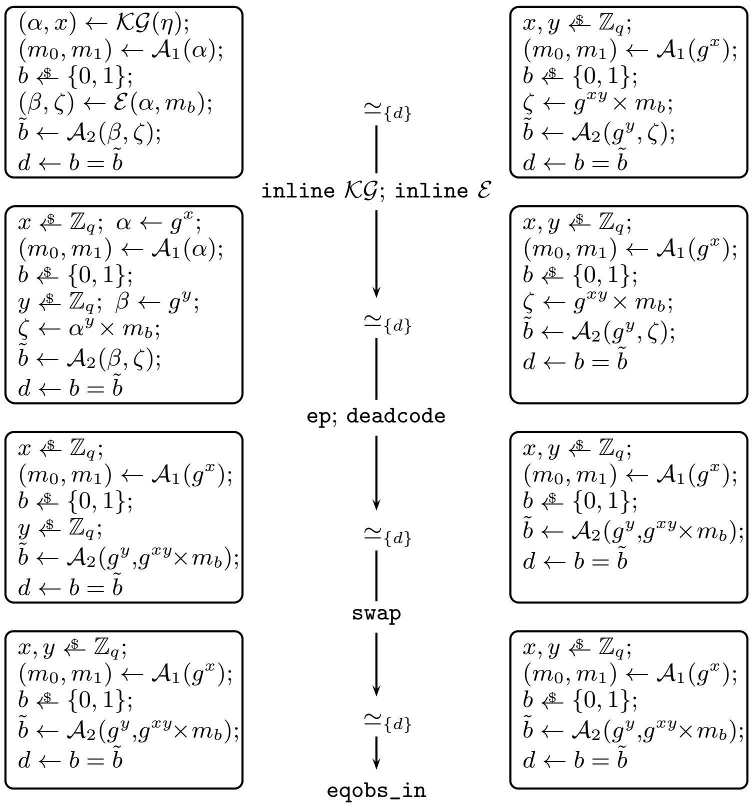
Fig. 1.1. Code-based proof of ElGamal semantic security.

Equations (1.2) and (1.5) are obtained similarly, while (1.3) is established by simple probabilistic reasoning: because in game G_3 the bit $\tilde{b}$ is independent from b, the probability of both bits being equal is exactly 1/2. Finally, to prove (1.4) we begin by removing the part the two games have in common with the exception of the instruction $z \xleftarrow{\$} \mathbb{Z}_q$ (`swap`, `eqobs_hd`, `eqobs_tl`) and then apply an algebraic property of cyclic groups that we have proved as a lemma (`otp`): if one applies the group operation to a uniformly distributed element of the group and some other constant element, the result is uniformly distributed—a random element acts as a one-time pad. This allows to prove that $z \xleftarrow{\$} \mathbb{Z}_q;\ \zeta \leftarrow g^z \times m_b$ and $z \xleftarrow{\$} \mathbb{Z}_q;\ \zeta \leftarrow g^z$ induce the same distribution on ζ, and thus remove the dependence of $\tilde{b}$ on b.

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

1.4.2 The Hashed ElGamal Encryption Scheme

Hashed ElGamal is a variant of the ElGamal public-key encryption scheme that does not require plaintexts to be members of the underlying group G. Instead, plaintexts in Hashed ElGamal are just bitstrings of a certain length ℓ and group elements are



```
$(\alpha, x) \leftarrow \mathcal{KG}(\eta);$                $x, y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha);$              $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$                        $b \xleftarrow{\$} \{0,1\};$
$(\beta, \zeta) \leftarrow \mathcal{E}(\alpha, m_b);$      $\simeq_{\{d\}}$     $\zeta \leftarrow g^{xy} \times m_b;$
$\tilde{b} \leftarrow \mathcal{A}_2(\beta, \zeta);$             $\tilde{b} \leftarrow \mathcal{A}_2(g^y, \zeta);$
$d \leftarrow b = \tilde{b}$                       $d \leftarrow b = \tilde{b}$
```

inline $\mathcal{KG}$; inline $\mathcal{E}$

```
$x \xleftarrow{\$} \mathbb{Z}_q;\ \alpha \leftarrow g^x;$          $x, y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha);$              $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$                        $b \xleftarrow{\$} \{0,1\};$
$y \xleftarrow{\$} \mathbb{Z}_q;\ \beta \leftarrow g^y;$    $\simeq_{\{d\}}$     $\zeta \leftarrow g^{xy} \times m_b;$
$\zeta \leftarrow \alpha^y \times m_b;$                  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, \zeta);$
$\tilde{b} \leftarrow \mathcal{A}_2(\beta, \zeta);$             $d \leftarrow b = \tilde{b}$
$d \leftarrow b = \tilde{b}$
```

ep; deadcode

```
$x \xleftarrow{\$} \mathbb{Z}_q;$                          $x, y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$                $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$                        $b \xleftarrow{\$} \{0,1\};$
$y \xleftarrow{\$} \mathbb{Z}_q;$                 $\simeq_{\{d\}}$     $\tilde{b} \leftarrow \mathcal{A}_2(g^y, g^{xy} \times m_b);$
$\tilde{b} \leftarrow \mathcal{A}_2(g^y, g^{xy} \times m_b);$   $d \leftarrow b = \tilde{b}$
$d \leftarrow b = \tilde{b}$
```

swap

```
$x, y \xleftarrow{\$} \mathbb{Z}_q;$                       $x, y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$                $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$                        $b \xleftarrow{\$} \{0,1\};$
$\tilde{b} \leftarrow \mathcal{A}_2(g^y, g^{xy} \times m_b);$  $\simeq_{\{d\}}$  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, g^{xy} \times m_b);$
$d \leftarrow b = \tilde{b}$                       $d \leftarrow b = \tilde{b}$
```

eqobs_in

Fig. 1.2. Sequence of transformations in the proof of $\mathsf{IND\text{-}CPA} \simeq_{\{d\}} \mathsf{G}_1$.

mapped into bitstrings using a hash function $H : G \to \{0,1\}^\ell$. Formally, the scheme is defined by the following triple of algorithms:

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

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

Definition 1.5 (Computational Diffie-Hellman assumption). *Consider the following game*

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

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

and define the success probability of $\mathcal{B}$ against CDH as follows

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

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

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

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

The proof is sketched in Figure 1.3. We follow the convention of typesetting global variables in boldface. The figure shows the sequence of games used to relate the success of the IND-CPA adversary in the original attack game to the success of the CDH adversary $\mathcal{B}$; the definition of the hash oracle is shown alongside each game. As in the proof of the semantic security of ElGamal, we begin by inlining the calls to $\mathcal{KG}$ and $\mathcal{E}$ in the IND-CPA game to obtain an observationally equivalent game G_1 such that

$$\Pr\left[\mathsf{IND\text{-}CPA} : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_1 : b = \tilde{b}\right] \tag{1.6}$$

We then fix the value $\hat{\boldsymbol{h}}$ that the hash oracle gives in response to g^{xy}. This is an instance of the *lazy sampling* transformation: any value that is randomly sampled at some point in a program can be sampled in advance, somewhere earlier in the program. This transformation is automated in CertiCrypt and is described in greater detail in Section 3.2.3. We get

$$\Pr\left[\mathsf{G}_1 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_2 : b = \tilde{b}\right] \tag{1.7}$$

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

```
Game IND-CPA :
  $L \leftarrow$ nil;
  $(\alpha, x) \leftarrow \mathcal{KG}(\eta)$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $(\beta, v) \leftarrow \mathcal{E}(\alpha, m_b)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\beta, v)$

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

$\simeq_{\{b,\tilde{b}\}}$

```
Game G_1 :
  $L \leftarrow$ nil;
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $h \leftarrow H(g^{xy})$;
  $v \leftarrow h \oplus m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, v)$

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

$\simeq_{\{b,\tilde{b}\}}$

```
Game G_2 :
  $\hat{h} \xleftarrow{\$} \{0,1\}^\ell$;
  $L \leftarrow$ nil;
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $\Lambda \leftarrow g^{xy}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $h \leftarrow H(\Lambda)$;
  $v \leftarrow h \oplus m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, v)$

Oracle $H(\lambda)$ :
  if $\lambda \notin$ dom($L$) then
    if $\lambda = \Lambda$ then
      $h \leftarrow \hat{h}$
    else $h \xleftarrow{\$} \{0,1\}^\ell$
    $L \leftarrow (\lambda, h) :: L$
  else $h \leftarrow L[\lambda]$
  return $h$
```

$\sim_{\{b,\tilde{b}\} \wedge \phi_{23}}$

```
Game G_3 :
  $\hat{h} \xleftarrow{\$} \{0,1\}^\ell$;
  $L \leftarrow$ nil;
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $\Lambda \leftarrow g^{xy}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $h \leftarrow \hat{h}$;
  $v \leftarrow h \oplus m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, v)$

Oracle $H(\lambda)$ :
  if $\lambda = \Lambda$ then
    $h \leftarrow \hat{h}$
  else
    if $\lambda \notin$ dom($L$) then
      $h \xleftarrow{\$} \{0,1\}^\ell$
      $L \leftarrow (\lambda, h) :: L$
    else $h \leftarrow L[\lambda]$
  return $h$
```

```
Game G_4 $\boxed{G_5}$ :
  bad $\leftarrow$ false;
  $\hat{h} \xleftarrow{\$} \{0,1\}^\ell$;
  $L \leftarrow$ nil;
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $\Lambda \leftarrow g^{xy}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $v \leftarrow \hat{h} \oplus m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, v)$

Oracle $H(\lambda)$ :
  if $\lambda \notin$ dom($L$) then
    if $\lambda = \Lambda$ then
      bad $\leftarrow$ true;
      $h \leftarrow \hat{h}$
      $\boxed{h \xleftarrow{\$} \{0,1\}^\ell}$
    else $h \xleftarrow{\$} \{0,1\}^\ell$
    $L \leftarrow (\lambda, h) :: L$
  else $h \leftarrow L[\lambda]$
  return $h$
```

$\sim_{\{L,\Lambda,b,\tilde{b}\} \wedge (\mathbf{bad} \Longrightarrow \Lambda \in \mathsf{dom}(L))} \langle 1 \rangle$

```
Game G_6 :
  $L \leftarrow$ nil;
  $x, y \xleftarrow{\$} \mathbb{Z}_q$;
  $\Lambda \leftarrow g^{xy}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $v \xleftarrow{\$} \{0,1\}^\ell$;
  $\hat{h} \leftarrow v \oplus m_b$;
  $\tilde{b} \leftarrow \mathcal{A}_2(g^y, v)$

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

$\simeq_{\{L,x,y\}}$

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

Adversary $\mathcal{B}(\alpha, \beta)$
  $L \leftarrow$ nil;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha)$;
  $v \xleftarrow{\$} \{0,1\}^\ell$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\beta, v)$;
  $\gamma \xleftarrow{\$}$ dom($L$);
  return $\gamma$

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

Fig. 1.3. Game-based proof of semantic security of Hashed ElGamal encryption in the Random Oracle Model.

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

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

$$\Pr\left[\mathsf{G}_2 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_3 : b = \tilde{b}\right] \tag{1.8}$$

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

$$\Pr\left[\mathsf{G}_3 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_4 : b = \tilde{b}\right] \tag{1.9}$$

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

$$\left|\Pr\left[\mathsf{G}_4 : b = \tilde{b}\right] - \Pr\left[\mathsf{G}_5 : b = \tilde{b}\right]\right| \leq \Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] \tag{1.10}$$

This form of reasoning is pervasive in game-based cryptographic proofs and is an instance of the so-called Fundamental Lemma that we discuss in detail in Section 3.3. In addition, we establish that $\mathbf{bad} \implies \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})$ is a post-condition of game G_5 and thus

$$\Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_5 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] \tag{1.11}$$

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

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

The semantic equivalence of these two program fragments can be proved using the probabilistic relational Hoare logic presented in Section 3.1—a proof is given in Section 3.2.2. Hence,

$$\Pr\left[\mathsf{G}_5 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_6 : b = \tilde{b}\right] \tag{1.12}$$

and

$$\Pr\left[\mathsf{G}_5 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] = \Pr\left[\mathsf{G}_6 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] \tag{1.13}$$

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

$$\Pr\left[\mathsf{G}_6 : b = \tilde{b}\right] = \frac{1}{2} \tag{1.14}$$

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

$$\Pr\left[\mathsf{G}_6 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right] = \Pr\left[\mathsf{G}_6 : g^{xy} \in \mathsf{dom}(\boldsymbol{L})\right] \leq q_{\mathsf{H}} \Pr\left[\mathsf{CDH} : \gamma = g^{xy}\right] \tag{1.15}$$

To summarize, from (1.6)—(1.15) we obtain

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

For any adversary $\mathcal{A}$ that executes in polynomial time, we can assume that the bound q_{H} on the number of queries is polynomial on the security parameter. Under the CDH assumption, the $\mathsf{IND\text{-}CPA}$ advantage of adversary $\mathcal{A}$ must then be negligible. Otherwise, the adversary $\mathcal{B}$ that we constructed would solve CDH with non-negligible probability, contradicting our computational assumption. To see this, we need to verify that adversary $\mathcal{B}$ runs in probabilistic polynomial time, but this is the case because procedures $\mathcal{A}_1, \mathcal{A}_2$ do, and $\mathcal{B}$ does not perform any additional costly computations. As in the previous example, the proof of this latter fact is completely automated in CertiCrypt.

We note that Hashed ElGamal can also be proved semantically secure in the standard model, but under the stronger DDH assumption. A game-based proof appears in [Barthe et al. 2009a]. The security reduction can be made under the hypothesis that the family of hash functions H is *entropy smoothing*—such a family of hash functions can be built without additional assumptions using the Leftover Hash Lemma [Håstad et al. 1999].

2 A Language for Describing Games

S

We have tried so far to be as rigorous as possible in our treatment of cryptographic proofs. We argued that a game-based approach can lead to tidier, more understandable proofs that help eliminate conspicuous errors and make clear the reasoning behind each step in a proof. We moved one step forward, taking a language-based approach and regarding games as programs. But still, our understanding of what a game means remains purely intuitive. In this chapter we will make this intuition precise by defining formally the probabilistic language we use to describe games and its semantics. This semanticist perspective allows a precise specification of the interaction between an adversary and the challenger in a game, and to readily answer questions that often arise in proofs, such as: Which oracles does the adversary have access to? Can the adversary read/write this variable? How many queries the adversary can make to a given oracle? What is the type/length of a value returned by the adversary? Can the adversary repeat a query? Furthermore, the framework enables us to give very precise definitions of fundamental notions such as probabilistic polynomial-time complexity or termination which are of paramount importance in the specification of security definitions and computational hardness assumptions.

2.1 Mathematical Preliminaries

2.1.1 The Unit Interval

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

Addition:	$(x, y) \mapsto \min(x + y, 1)$, where $+$ denotes addition over reals;
Inversion:	$x \mapsto 1 - x$, where $-$ denotes subtraction over reals;

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

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

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

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

More generally, for any complete partially ordered set D, we use $\sup f$ to denote the least upper bound of a monotonic sequence $f : \mathbb{N} \to D$. Note that a cpo structure on D induces a cpo structure in the function space $A \to D$ by taking

$$\begin{array}{lcl} f \leq_{A \to D} g & \stackrel{\text{def}}{=} & \forall x : A.\ f(x) \leq_D g(x) \\ 0_{A \to D} & \stackrel{\text{def}}{=} & \lambda x.\ 0_D \\ \sup_{A \to D} f & \stackrel{\text{def}}{=} & \lambda x.\ \sup_D(f(x)) \end{array}$$

2.1.2 The Measure Monad

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

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

However, it is convenient to take a more general approach and represent instead a distribution over a set A as a probability measure, by giving a function that maps a $[0, 1]$-valued random variable (a function in $A \to [0, 1]$) to its expected value, i.e. the integral of the random variable with respect to the probability measure. This view of distributions eliminates the need of cluttered definitions and proofs involving summations, and allows us to give a continuation-passing style semantics to programs by defining a suitable monadic structure on distributions. Formally, we represent a distribution on A as a function μ of type

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

satisfying the following properties:

Monotonicity: $\qquad f \leq g \implies \mu\ f \leq \mu\ g;$

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

We do not restrict our attention only to distributions with probability mass of 1, but we consider as well sub-probability distributions, that may have a total mass strictly less than 1. As we will see, this is key to give semantics to *non-terminating* programs (i.e. programs that do not terminate with probability 1).

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

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

These operators satisfy the usual monadic laws

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

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

2.1.3 Lifting Predicates and Relations to Distributions

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

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

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

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

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

Definition 2.1 (Lifting predicates to distributions). *Let μ be a distribution on a set A, and P be a predicate on A. We define the lifting of P to μ as follows:*

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

This rather contrived definition is necessary because we consider sub-probability distributions whose total measure may be less than 1; equivalently we could have stated it as:

$$\mathsf{range}\ P\ \mu \ \overset{\text{def}}{=}\ \forall f.\ (\forall x.\ P\ x \implies f\ x = 1) \implies \mu\ f = \mu\ \mathbb{1}$$

Note also that due to the way distributions are formalized, the above definition is strictly stronger than the following, more intuitive definition:

$$\mathsf{range}\ P\ \mu \ \overset{\text{def}}{=}\ \forall x \in \mathsf{support}(\mu).\ 0 < \mu(\mathbb{I}_{\{x\}}) \implies P\ x$$

This latter definition makes sense only for distributions with countable support, for which it can be proved equivalent to the above definitions.

Definition 2.2 (Lifting relations to distributions). *Let μ_1 be a probability distribution on a set A and μ_2 a probability distribution on a set B. We define the lifting of a relation $R \subseteq A \times B$ to μ_1 and μ_2 as follows:*

$$\mu_1\ R^{\#}\ \mu_2 \ \overset{\text{def}}{=}\ \exists \mu : \mathcal{D}(A \times B).\ \pi_1(\mu) = \mu_1\ \wedge\ \pi_2(\mu) = \mu_2\ \wedge\ \mathsf{range}\ R\ \mu \qquad (2.1)$$

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

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

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

Let us give an example that conveys a better intuition; suppose one wants to prove $\mathcal{U}_A\ R^{\#}\ \mathcal{U}_B$, where $\mathcal{U}_X$ stands for the uniform probability distribution on a finite set X. When A and B have the same size, proving this is equivalent to exhibiting a bijection $f : A \to B$ such that for every $a \in A$, $R(a, f(a))$ holds. Indeed, using such f it is easy to build a distribution μ on $A \times B$ that satisfies the condition in (2.1):

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

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

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

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

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

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

$$\mu_1 \; \Phi^{\#} \; \mu_2 \; \wedge \; f =_{\Phi} g \quad \Longrightarrow \quad \mu_1 \; f = \mu_2 \; g$$

and analogously.

$$\mu_1 \; \Phi^{\#} \; \mu_2 \; \wedge \; f \leq_{\Phi} g \quad \Longrightarrow \quad \mu_1 \; f \leq \mu_2 \; g$$

We use these properties to prove rules relating observational equivalence to probability in Section 3.1.

It can be shown that lifting preserves the reflexivity and symmetry of the lifted relation, but proving that it preserves transitivity is not as straightforward. Ideally, one would like to have for probability measures $\mu_1 : \mathcal{D}(A)$, $\mu_2 : \mathcal{D}(B)$, $\mu_3 : \mathcal{D}(C)$ and relations $\Psi \subseteq A \times B$, $\Phi \subseteq B \times C$

$$\mu_1 \; \Psi^{\#} \; \mu_2 \; \wedge \; \mu_2 \; \Phi^{\#} \; \mu_3 \quad \Longrightarrow \quad \mu_1 \; (\Psi \circ \Phi)^{\#} \; \mu_3$$

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

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

Lemma 2.3. *Consider $d_1 : \mathcal{D}(A)$, $d_2 : \mathcal{D}(B)$, $d_3 : \mathcal{D}(C)$ such that d_2 has countable support. Suppose there exist distributions $\mu_{12} : \mathcal{D}(A \times B)$ and $\mu_{23} : \mathcal{D}(B \times C)$ that make $\mu_1 \; \Psi^{\#} \; \mu_2$ and $\mu_2 \; \Phi^{\#} \; \mu_3$ hold. Then, the following distribution over $A \times C$ is a witness for the existential in $\mu_1 \; (\Psi \circ \Phi)^{\#} \; \mu_3$:*

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

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

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

□

2.2 The pWhile Language

We describe games as programs in the pWhile language, a probabilistic extension of an imperative language with procedure calls. This language can be regarded as a mild generalization of the language proposed by Bellare and Rogaway [2006], in that our language allows while loops whereas theirs only allow bounded for loops. The formalization of pWhile is carefully crafted to exploit key features of Coq: it uses modules to support an extensible expression language that can be adapted according to the verification goal, dependent types to ensure that programs are well-typed and have a total semantics, and monads to give semantics to probabilistic programs and capture the cost of executing them.

We formalize programs in a deep-embedding style, i.e. the syntax of the language is encoded within the proof assistant. Deep embeddings offer one tremendous advantage over shallow embeddings, in which the language used to represent programs is the same as the underlying language of the proof assistant. Namely, a deep embedding allows to manipulate programs as syntactic objects. This permits to achieve a high level of automation in reasoning about programs through certified tactics that implement syntactic program transformations. Additionally, a deep embedding allows to formalize complexity issues neatly and to reason about programs by induction on the structure of their code.

The semantics of programs is given by an interpretation function that takes a program p—an element of the type of programs—and an initial state s, and returns the result of executing p starting from s. In a deterministic case, both s and the result of executing p starting from s would be deterministic states, i.e. memories mapping variables to values. In the case of pWhile programs, the denotation of a program is instead a function mapping an initial state to a (sub)-probability measure over final states. We use the measure monad described in 2.1 to define the denotation of programs.

2.2.1 Syntax

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

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

The inductive definition of the language suffices to understand the rest of the presentation and the reader may prefer to retain it for further reference. In practice, however, variable and procedure identifiers are annotated with their types, and the syntax of programs is dependently-typed. Thus, $x \leftarrow e$ is well-formed only if the types of x and e coincide, and if e then c_1 else c_2 is well-formed only if e is a Boolean expression and c_1 and c_2 are themselves well-formed. An immediate benefit of using dependent types is that the type system of Coq ensures for free the well-typedness of expressions and commands.

In the remainder of this section we describe in detail the formalization of the syntax and semantics of the language. Most readers, particularly those not familiar with Coq, can skim through this section without hindering the understanding of the rest of the dissertation.

Background on the Coq proof assistant

We built our framework on top of Coq, a general purpose proof assistant that has been used for over two decades to formalize results in mathematics and computer science [The Coq development team 2009]. Coq provides an expressive specification language based on the Calculus of Inductive Constructions, a higher-order dependently-typed λ-calculus in which mathematical notions can be formalized conveniently. The Coq logic distinguishes between types, of type Type, which represent sets, and propositions, of type Prop, which represent formulae: thus, $a : A$ is interpreted as "a is an element of type A" if A is a set, and as "a is a proof of A" if A is a proposition. In the latter case, we say that a is a proof object. Types can either be introduced by standard constructions, e.g. (generalized) function space and products, or by inductive definitions. Most common inductive types are predefined, including the type $\mathbb{N}$ of natural numbers, the type $\mathbb{B}$ of Boolean values, and sum and product types. We will also use the inductively defined types of homogeneous and heterogeneous lists. Homogeneous lists are composed of elements of the same type. The polymorphic inductive type of homogeneous lists is defined as follows:

$$\begin{array}{l}\textbf{Inductive}\ \mathsf{list}\ A : \mathsf{Type} := \\ \mid \mathsf{nil}\quad : \mathsf{nil} \\ \mid \mathsf{cons} : A \to \mathsf{list}\ A \to \mathsf{list}\ A\end{array}$$

The list constructor cons is usually represented using an infix notation as the operator "::". Thus, the list composed of the natural numbers 1, 2 and 3, in that order, has type list $\mathbb{N}$ and could be represented as $(1 :: 2 :: 3 :: \mathsf{nil})$. Heterogeneous lists are composed of elements whose type may depend on a value. Given a type A and a type-valued function $P : A \to \mathsf{Type}$, the inductive type of heterogeneous lists is defined as follows:

$$\begin{array}{l}\textbf{Inductive}\ \mathsf{hlist}\ A\ (P : A \to \mathsf{Type}) : \mathsf{list}\ A \to \mathsf{Type} := \\ \mid \mathsf{dnil}\quad : \mathsf{hlist}\ \mathsf{nil} \\ \mid \mathsf{dcons} : \forall a\ l,\ \ P\ a \to \mathsf{hlist}\ l \to \mathsf{hlist}\ (a :: l)\end{array}$$

We will use $A^\star$ to denote the type of A-lists (i.e. list A), and $P_l^\star$ to denote the type of heterogeneous P-lists over a list of values l (i.e. hlist $P\ l$).

Types

We formalize a dependently-typed syntax, and use the underlying type system of Coq to ensure for free that expressions and commands are well-formed. In our experience, the typed syntax provides particularly useful feedback when debugging proofs and makes proofs easier by restricting reasoning about programs to reasoning about their meaningful behaviors.

The types and expressions of the language are defined on top of a module that contains the declaration of user-defined types and operators. Formally, the set $\mathcal{T}$ of types is defined as:

$$\begin{array}{l}\textbf{Inductive}\ \mathcal{T} : \mathsf{Type} := \\ \mid \mathsf{User}\quad : \mathcal{T}_{\mathsf{user}} \to \mathcal{T} \\ \mid \mathsf{Nat}\quad : \mathcal{T} \\ \mid \mathsf{Bool}\quad : \mathcal{T} \\ \mid \mathsf{List}\quad : \mathcal{T} \to \mathcal{T} \\ \mid \mathsf{Pair}\quad : \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\ \mid \mathsf{Sum}\quad : \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\ \mid \mathsf{Option} : \mathcal{T} \to \mathcal{T}\end{array}$$

where $\mathcal{T}_{\mathsf{user}}$ denotes the set of user-defined types. This set can be given different definitions according to the cryptographic system under verification.

The interpretation of types—and of programs in general—depends on a security parameter η (a natural number),

$$
\begin{array}{l}
\textbf{Definition}\ \mathsf{interp}\ (\eta : \mathbb{N})\ (t : \mathcal{T}) := \\
\quad \mathsf{match}\ t\ \mathsf{with} \\
\quad \mid \mathsf{User}\ ut \quad \Rightarrow \mathsf{interp}_{\mathsf{user}}\ \eta\ ut \\
\quad \mid \mathsf{Nat} \qquad \Rightarrow \mathbb{N} \\
\quad \mid \mathsf{Bool} \qquad \Rightarrow \mathbb{B} \\
\quad \mid \mathsf{List}\ t \quad \Rightarrow \mathsf{list}\ (\mathsf{interp}\ \eta\ t) \\
\quad \mid \mathsf{Pair}\ t_1\ t_2 \Rightarrow (\mathsf{interp}\ \eta\ t_1) \times (\mathsf{interp}\ \eta\ t_2) \\
\quad \mid \mathsf{Sum}\ t_1\ t_2 \Rightarrow (\mathsf{interp}\ \eta\ t_1) + (\mathsf{interp}\ \eta\ t_2) \\
\quad \mathsf{end}
\end{array}
$$

For instance, to introduce a type of bitstrings of a certain length $\ell(\eta)$ depending on the security parameter, one would define $\mathcal{T}_{\mathsf{user}}$ as follows,

$$
\begin{array}{l}
\textbf{Inductive}\ \mathcal{T}_{\mathsf{user}} : \mathsf{Type} := \\
\mid \mathsf{Bitstring} : \mathcal{T}_{\mathsf{user}}
\end{array}
$$

and let the interpretation of Bistring for a value η of the security parameter be some representation of the set $\{0,1\}^{\ell(\eta)}$ in Coq (the type of bitvectors defined in the standard library of Coq provides a convenient representation).

Expressions

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{Inductive}\ \mathcal{E} : \mathcal{T} \to \mathsf{Type} :=} \\
\mid \mathsf{Enat} & :> & \mathbb{N} \to \mathcal{E}_{\mathsf{Nat}} \\
\mid \mathsf{Ebool} & :> & \mathbb{B} \to \mathcal{E}_{\mathsf{Bool}} \\
\mid \mathsf{Evar} & :> & \forall t,\ \mathcal{V}_t \to \mathcal{E}_t \\
\mid \mathsf{Eop} & : & \forall op,\ \mathcal{E}^{\star}_{(\mathsf{targs}\ op)} \to \mathcal{E}_{(\mathsf{tres}\ op)} \\
\mid \mathsf{Eforall} & : & \forall t,\ \mathcal{V}_t \to \mathcal{E}_{\mathsf{Bool}} \to \mathcal{E}_{(\mathsf{List}\ t)} \to \mathcal{E}_{\mathsf{Bool}} \\
\mid \mathsf{Eexists} & : & \forall t,\ \mathcal{V}_t \to \mathcal{E}_{\mathsf{Bool}} \to \mathcal{E}_{(\mathsf{List}\ t)} \to \mathcal{E}_{\mathsf{Bool}} \\
\mid \mathsf{Efind} & : & \forall t,\ \mathcal{V}_t \to \mathcal{E}_{\mathsf{Bool}} \to \mathcal{E}_{(\mathsf{List}\ t)} \to \mathcal{E}_t
\end{array}
$$

The first three clauses declare constructors as coercions; thanks to this mechanism it is possible to view an element of their domain as an element of their codomain, e.g. a natural number as an expression of type Nat and a variable of type t as an expression of type t. The fourth clause corresponds to the standard rule for operators; the rule requires that the types of the arguments be compatible with the declaration of the operator, as enforced by the type $\mathcal{E}^{\star}_{(\mathsf{targs}\ op)}$ of heterogeneous lists of expressions. In this clause, op is universally quantified over an inductive type that contains a fixed set of operators for base types and user-defined operators.

The last three clauses introduce operations on lists that are commonly used in cryptographic proofs: they take as parameters a variable x of type t, a Boolean valued expression e that may depend on x, and an expression l of type $\mathsf{List}\ t$, and respectively

($\mathsf{Eforall}\ x\ e\ l$)	checks whether every element a in l verifies e when substituting a for x. We note it as $(\forall x \in l.\ e)$;
($\mathsf{Eexists}\ x\ e\ l$)	checks whether some element a in l verifies e when substituting a for x. We note it as $(\exists x \in l.\ e)$;
($\mathsf{Efind}\ x\ e\ l$)	evaluates to the first element a in the list l that verifies e when substituting a for x, or to a designated default element of type t if no such element is found. We usually do not write this operator explicitly, instead we assume that an expression $(\exists x \in l, e)$ implicitly assigns to the variable x the value of ($\mathsf{Efind}\ x\ e\ l$).

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

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

In addition to the set of deterministic expressions defined above, to encode random assignments we use a set of type-indexed distribution expressions $\mathcal{DE}$. An element of $\mathcal{DE}_t$ denotes some discrete distribution over values of type t. The core language includes expressions denoting the uniform distribution on natural intervals of the form $[0..n]$, and on Boolean values. Again, the set of distribution expressions of the core language can be extended by the user,

$$\begin{array}{l}
\textbf{Inductive}\ \mathcal{DE} : \mathcal{T} \to \mathsf{Type} := \\
\mid \mathsf{Dnat}\ \ : \mathcal{E}_{\mathsf{Nat}} \to \mathcal{DE}_{\mathsf{Nat}} \\
\mid \mathsf{Dbool} : \mathcal{DE}_{\mathsf{Bool}} \\
\mid \mathsf{Duser} : \forall t,\ \mathcal{DE}_{\mathsf{user}}\ t \to \mathcal{DE}_t
\end{array}$$

Programs

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

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

For instance, $b \xleftarrow{\$} \{0,1\}$ (a shorthand for $\mathsf{Rand}\ b\ \mathsf{Dbool}$) is an instruction that samples a random bit with uniform probability and assigns it to variable b. Note that the above syntax lacks a construct for sequential composition; instead, we use lists to represent sequences of instructions.

Definition 2.4 (Program). *A program is a pair consisting of a command $c \in \mathcal{C}$ and an environment $E : \forall l\ t,\ \mathcal{P}_{(l,t)} \to \mathsf{decl}_{(l,t)}$, which maps procedure identifiers to their declaration. The declaration of a procedure $p \in \mathcal{P}_{(l,t)}$ consists of its formal parameters, its body, and a return expression,*

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

An environment specifies the type of the parameters and the return expression of procedures, so that procedure calls are always well-typed. In a typical formalization, the environment will map procedures to closed commands, with the exception of adversaries whose code is unknown, and thus modeled by variables of type $\mathcal{C}$. This is a standard trick to deal with uninterpreted functions in a deep embedding.

We frequently make no distinction between a game $G = (c, E)$ and its main command c when the environment either has no relevance or is clear from the context. In the remainder, we revert to a more natural notation to specify games: we rely on standard notation as in [Barthe et al. 2009a,c]. In particular, we write procedures that might have multiple exit points and use explicit return instructions instead of specifying a single return expression.

2.2.2 Semantics

The semantics of commands and expressions depends on a natural number representing the security parameter. As we have seen, the interpretation of types and operators may depend on this parameter, but for the sake of readability we omit it most of the time. The denotation of a command is defined relative to an initial memory, mapping variables to values of their respective types. Since variables are partitioned into local and global variables, we will sometimes represent a memory m as a pair of mappings ($m.\mathsf{loc}, m.\mathsf{glob}$) for local and global variables, respectively. We let $\mathcal{M}$ denote the type of memories and $\varnothing$ denote a mapping associating variables to default values of their corresponding types.

Expressions are deterministic; their semantics is standard and given by a function

$$[\![e : \mathcal{E}_t]\!]_{\mathcal{E}} : \mathcal{M} \to \mathsf{interp}\ t$$

that evaluates an expression in a given memory and returns a value of the right type. The semantics of distribution expressions is given by a function a

$$[\![d : \mathcal{DE}_t]\!]_{\mathcal{DE}} : \mathcal{M} \to \mathcal{D}(\mathsf{interp}\ t)$$

that given a distribution expression d of type t and a memory m, returns a measure over values of type t. For instance, in Section 1.4.2 we have used $\{0,1\}^\ell$ to denote the uniform distribution on bitstrings of a certain length ℓ; formally, we have

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

Observe that distribution expressions are not restricted to constant distributions. Indeed, for any expression of type $e \in \mathcal{E}_{\mathsf{Nat}}$, the semantics of the uniform distribution on the natural interval $[0..e]$ depends on the evaluation of e,

$$[\![\ [0..e]\ ]\!]_{\mathcal{DE}}\ m : \mathcal{D}(\mathbb{N}) \ \stackrel{\text{def}}{=}\ \lambda f.\ \sum_{i=0}^{n} \frac{1}{n+1}\ f(i) \quad \text{where } n = [\![e]\!]_{\mathcal{E}}\ m$$

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

The (small-step) semantics of commands relates an initial deterministic state to a sub-probability distribution over final deterministic states. It uses a frame stack to deal with procedure calls. Formally, a deterministic state is a triple consisting of the current command $c : \mathcal{C}$, a memory $m : \mathcal{M}$, and a frame stack $F : \mathsf{frame}^\star$. We let $\mathcal{S}$ denote the set of deterministic states,

$$\mathcal{S} \ \stackrel{\text{def}}{=}\ \mathcal{C} \times \mathcal{M} \times \mathsf{frame}^\star$$

One step execution $[\![\cdot]\!]_1 : \mathcal{S} \to \mathcal{D}(\mathcal{S})$ is defined by the rules of Figure 2.1; in the figure, we use $a \rightsquigarrow b$ as a notation for $[\![a]\!]_1 = b$.

We briefly comment on the transition rules for calling a procedure (5th rule) and returning from a call (2nd rule). Upon a call, a new frame is appended to the stack, containing the destination variable, the return expression of the called procedure, the continuation to the call, and the local memory of the caller. The state resulting from the call contains the body of the called procedure, the global part of the memory, a local memory initialized to map the formal parameters to the value of the actual parameters just before the call, and the updated stack. When returning from a call with a non-empty stack, the top frame is popped, the return expression is evaluated and the resulting value is assigned to the destination variable after previously restoring the local memory of the caller; the continuation taken from the frame becomes the current command. If the stack is empty when returning from a call, the execution of the program terminates and the final state is embedded into the monad using the unit operator.

We then define an n-step execution function $[\![\cdot]\!]^n$ as follows:

$$\begin{array}{lcl} [\![s]\!]^0 & \stackrel{\text{def}}{=} & \mathsf{unit}\ s \\ [\![s]\!]^{n+1} & \stackrel{\text{def}}{=} & \mathsf{bind}\ [\![s]\!]^n\ [\![\cdot]\!]_1 \end{array}$$

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

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

where $f_{|\mathsf{final}}$ is a function that when applied to a state (c, m, F) equals $f(m)$ if the state is a final state and 0 otherwise, i.e.

$$\begin{array}{rcl}
(\mathsf{skip}, m, \mathsf{nil}) & \rightsquigarrow & \mathsf{unit}\ (\mathsf{skip}, m, \mathsf{nil}) \\
(\mathsf{skip}, m, (x, e, c, l) :: F) & \rightsquigarrow & \mathsf{unit}\ (c, (l, m.\mathsf{glob})\{[\![e]\!]_{\mathcal{E}}\ m/x\}, F) \\
(x \leftarrow e; c, m, F) & \rightsquigarrow & \mathsf{unit}\ (c, m\{[\![e]\!]_{\mathcal{E}}\ m/x\}, F) \\
(x \overset{\$}{\leftarrow} d; c, m, F) & \rightsquigarrow & \mathsf{bind}\ ([\![d]\!]_{\mathcal{DE}}\ m)\ (\lambda v.\ \mathsf{unit}\ (c, m\{v/x\}, F)) \\
(x \leftarrow p(\mathbf{e}); c, m, F) & \rightsquigarrow & \begin{array}{l}\mathsf{unit}\ (p.\mathsf{body}, (\varnothing\{[\![\mathbf{e}]\!]_{\mathcal{E}}\ m/p.\mathsf{args}\}, m.\mathsf{glob}), \\ \qquad (x, p.\mathsf{re}, c, m.\mathsf{loc}) :: F)\end{array} \\
(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c, m, F) & \rightsquigarrow & \begin{cases}\mathsf{unit}\ (c_1; c, m, F) & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{true} \\ \mathsf{unit}\ (c_2; c, m, F) & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{false}\end{cases} \\
(\mathsf{while}\ e\ \mathsf{do}\ c; c', m, F) & \rightsquigarrow & \begin{cases}\mathsf{unit}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c; c', m, F) & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{true} \\ \mathsf{unit}\ (c', m, F) & \text{if } [\![e]\!]_{\mathcal{E}}\ m = \mathsf{false}\end{cases}
\end{array}$$

Fig. 2.1. Probabilistic one-step semantics of pWhile programs.

$$\begin{array}{rcl}
f_{|\mathsf{final}}\ (c, m, F) & : & \mathcal{S} \to [0,1] \\
f_{|\mathsf{final}}\ (c, m, F) & \stackrel{\text{def}}{=} & \begin{cases} f(m) & \text{if } c = \mathsf{skip} \wedge F = \mathsf{nil} \\ 0 & \text{otherwise}\end{cases}
\end{array}$$

The set of final states grows monotonically as the number of execution steps increases, which implies that the sequence $[\![(c, m, \mathsf{nil})]\!]^n\ f_{|\mathsf{final}}$ is increasing because f is non-negative. Because, in addition, this sequence is upper bounded by 1, the least upper bound in the definition of the denotation of a command always exists and corresponds to the limit of the sequence.

Figure 2.2 summarizes the denotational semantics of commands as equations following from the above limit construction. The denotation of a program relates an initial memory to a (sub-)probability distribution over memories using the measure monad presented in the previous section:

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

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

$$\begin{array}{rcl}
[\![c]\!]^{\#} & : & \mathcal{D}(\mathcal{M}) \to \mathcal{D}(\mathcal{M}) \\
[\![c]\!]^{\#} & \stackrel{\text{def}}{=} & \lambda\mu.\ \mathsf{bind}\ \mu\ [\![c]\!]
\end{array}$$

We have shown that the semantics of programs maps memories to discrete distributions, provided expressions in $\mathcal{DE}$ evaluate to distributions with countable support. We use this together with Lemma 2.3 to prove the soundness of some relational Hoare logic rules (namely, [Comp] and [Trans]) in Section 3.1.

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

Fig. 2.2. Denotational semantics of pWHILE programs.

Computing probabilities

The advantage of using this monadic semantics is that, if we use an arbitrary function as a continuation to the denotation of a program, what we get (for free) as a result is its expected value w.r.t. the distribution of final memories. In particular, we can compute the probability of an event A (represented as a function in $\mathcal{M} \to \mathbb{B}$) in the distribution obtained after executing a command c in an initial memory m by measuring its characteristic function $\mathbb{1}_A$:

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

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

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

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

In what follows, when writing probabilities we sometimes omit the initial memory m; in that case one may safely assume that the memory is initially $\varnothing$, which maps variables to default values of the right type.

2.3 Probabilistic Polynomial-Time Programs

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

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

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

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

Definition 2.5 (Termination). *The probability that a program c terminates starting from an initial memory m is* $\Pr[c, m : \mathsf{true}] = [\![c]\!]\ m\ \mathbb{1}$. *We say that a program c is absolutely terminating, and note it* $\mathsf{lossless}(c)$, *iff it terminates with probability 1 in any initial memory,*

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

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

- We postulate the execution time of each operator, in the form of a function that depends on the inputs of the operator—which corresponds to the so-called functional time model;
- We postulate for each datatype a size measure, in the form of a function that assigns to each value its memory footprint.

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

Definition 2.6 (Polynomially bounded distribution). *We say that a family of distributions $\{\mu_\eta : \mathcal{D}(\mathcal{M} \times \mathbb{N})\}$ is (p, q)-bounded, where p and q are polynomials,* *whenever for every value of the security parameter η and any pair (m, n) occurring with non-zero probability in μ_η, the size of values in m is bounded by $p(\eta)$ and the cost n is bounded by $q(\eta)$. This notion can be formally defined by means of the* range *predicate introduced in Section 2.1.3:*

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

Definition 2.7 (Strict probabilistic polynomial-time program). *We say that a program c is strict probabilistic polynomial-time (PPT) iff it terminates absolutely, and there exist polynomial transformers F, G such that for every (p, q)-bounded distribution family μ_η, $(\mathsf{bind}\ \mu_\eta\ [\![c]\!])$ is $(F(p), q + G(p))$-bounded.*

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

Although our formalization of termination and efficiency relies on semantic definitions, it is not necessary for users to reason directly about the semantics of a program to prove it meets those definitions. CertiCrypt implements a certified algorithm showing that every program without loops and recursive calls terminates absolutely.[1] We also provide another algorithm that, together with the first, establishes that a program is PPT provided that, additionally, the program does not contain expressions that might generate values of super-polynomial size or take a super-polynomial time when evaluated in a polynomially bounded memory.

Exact bounds on execution time

Extracting an exact security result from a reductionist game-based proof requires to lower bound the success probability of the reduction and to upper bound the overhead incurred in execution time. Computing a bound on the success probability is what takes most of the effort since it requires examining the whole sequence of games and a careful bookkeeping of the probability of events. On the other hand, bounding the overhead of a reduction only requires examining the last game in the sequence. While we have put a great effort in automating the computation of probability bounds and we developed an automated method to obtain asymptotic polynomial bounds on the execution time of reductions, we did not bother to provide a method to compute exact time bounds. To do so, we would need an alternative

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

cost-instrumented semantics that does not take into account the time spent in evaluating calls to oracles, but instead just records the number of queries that have been made. Assume that an adversary $\mathcal{A}$ executes within time t (without taking into account oracle calls) and makes at most $q_{\mathcal{O}_i}$ queries to oracle $\mathcal{O}_i$. Suppose we have a reduction where an adversary $\mathcal{B}$ uses $\mathcal{A}$ as a sub-procedure; assume wlog that $\mathcal{B}$ only calls $\mathcal{A}$ once and does not make any additional oracle calls. Then, we can argue that if $\mathcal{B}$ executes within time t' without taking into account the cost of evaluating calls to $\mathcal{A}$ (this could easily be computed by considering $\mathcal{A}$ as an oracle for $\mathcal{B}$), then $\mathcal{B}$ executes within time

$$t + t' + \sum_i q_{\mathcal{O}_i}\ t_{\mathcal{O}_i}$$

where $t_{\mathcal{O}_i}$ upper bounds the cost one query to oracle $\mathcal{O}_i$.

2.4 Adversaries

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

Formally, the interface of an adversary consists of a triple $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$, where $\mathcal{O}$ is the set of procedures that the adversary may call, $\mathcal{RW}$ the set of variables that it may read and write, and $\mathcal{R}$ the set of variables that it may only read. We say that an adversary $\mathcal{A}$ with interface $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$ is well-formed if the judgment $\vdash_{\mathrm{wf}} \mathcal{A}$ can be derived from the rules in Figure 2.3. Note that the rules are generic, only making sure that the adversary makes a correct use of variables and procedures. These rules guarantee that a well-formed adversary always initializes local variables before using them, only writes global variables in $\mathcal{RW}$, and only reads global variables in $\mathcal{RW} \cup \mathcal{R}$. For convenience, we allow adversaries to call procedures outside $\mathcal{O}$, but these procedures must themselves respect the same interface.

Additional constraints may be imposed on adversaries. For example, exact security proofs usually impose an upper bound to the number of calls adversaries can make to a given oracle, while some properties, such as IND-CCA2 (see §2.5.2 below), restrict the parameters with which oracles may be called at different stages in an experiment. Likewise, some proofs impose extra conditions such as forbidding repeated or malformed queries. These kinds of properties can be formalized using global variables that record calls to oracles and verifying as post-condition that all calls were legitimate.

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

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

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

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

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

Fig. 2.3. Rules for well-formedness of an adversary against interface $(\mathcal{O}, \mathcal{RW}, \mathcal{R})$. A judgment of the form $I \vdash c : O$ can be interpreted as follows: assuming variables in I may be read, the adversarial code fragment c respects the interface and after its execution variables in O may be read. Thus, if $I \vdash c : O$, then $I \subseteq O$.

2.5 Making Security Properties Precise

Before going any further in the formalization of cryptographic proofs, we need to be sure that the results that we prove are meaningful. Security definitions in cryptography have so many subtleties that it is not clear that the whole cryptographic community agrees even on the most fundamental of these definitions. To illustrate this point, let us analyze in detail two pervasive definitions that we use in subsequent chapters: the security of a signature scheme against existential forgery under adaptive chosen-message attacks (EF-CMA security), and the indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2 security) of an encryption scheme.

2.5.1 EF-CMA Security

We start by recalling the definition of digital signature schemes.

Definition 2.8 (Digital signature scheme). *A digital signature scheme is composed of a triple of algorithms:*

Key generation:	*Given a security parameter η, the key generation algorithm $\mathcal{KG}(\eta)$ returns a public/secret key pair (pk, sk);*
Signing:	*Given a secret key sk and a message m, the signing algorithm* $\mathsf{Sign}(sk, m)$ *produces a signature of m under sk;*
Verification:	*Given a public key pk, a message m, and a purported signature σ for m, the verification algorithm* $\mathsf{Verify}(pk, m, \sigma)$ *returns a Boolean value indicating whether the signature is valid or not.*

Key generation and signing may be probabilistic, while the verification algorithm is deterministic. We require that verification always succeeds for authentic signatures:

for every pair of keys (pk, sk) *that can be output by the key generation algorithm, and every message* m*, it must be the case that* $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = \mathsf{true}$.

We informally describe the way in which EF-CMA security is typically defined. The experiment begins by choosing a public verification key pk and a secret signing key sk, using the key-generation algorithm of the signature scheme. The public key is given to the forger, who can ask for the signature of messages of its choice to a signing oracle and eventually halts and outputs a message m together with a purported signature σ. The forger wins when the signature σ verifies. We say that the scheme is secure when the winning probability of any probabilistic polynomial-time forger is negligible. Since the forger could trivially win by asking for the signature of m, the forger is not allowed to query m to the signing oracle. Figure 2.4 depicts this experiment as a game.

```
**Game** $\mathsf{G}_{\mathsf{EF}}^{\mathcal{A}}$ :
  $\boldsymbol{S} \leftarrow \mathsf{nil}$;
  $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}(\eta)$;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$
```

```
**Oracle** $\mathsf{Sign}_{\mathcal{A}}(m)$ :
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}$;
  $\sigma \leftarrow \mathsf{Sign}(\boldsymbol{sk}, m)$;
  return $\sigma$
```

Fig. 2.4. The EF-CMA experiment; the signing oracle is instrumented to record the queries made by the forger.

The above definition seems unambiguous at first sight. There are however two ways of forbidding the adversary from querying m to the signing oracle. The first is simply to reject adversaries that may query m with non-zero probability; this amounts to restrict the quantification over adversaries—instead of considering all efficient forgers one considers only those forgers that do not query the message they output to the signing oracle. The second way is to test a posteriori whether the adversary queried m to its oracle, and to declare that it lost in this case. Following [Bellare et al. 2009], we call the former definitional style the *exclusion* ("E") style and the latter the *penalty* ("P") style. Both styles are common and used interchangeably in the literature; for instance [Bellare and Rogaway 1996; Katz and Wang 2003] use the penalty style while [Bellare and Rogaway 1993] uses the exclusion style. The question is whether the two styles result in equivalent definitions or not.

It should be clear that security in the penalty style implies security in the exclusion style. To see this, consider any efficient forger $\mathcal{A}$ valid in the exclusion style definition. The forger is also valid in the penalty style definition and it achieves the same success probability. Since the probability of $\mathcal{A}$ ever querying the message m whose signature it forges is 0, we have

$$\Pr\left[\mathsf{G}_{\mathsf{EF}}^{\mathcal{A}} : \mathsf{Verify}(pk, m, \sigma)\right] = \Pr\left[\mathsf{G}_{\mathsf{EF}}^{\mathcal{A}} : \mathsf{Verify}(pk, m, \sigma) \land m \notin \boldsymbol{S}\right]$$

Implication in the other direction (i.e., that security in the exclusion style implies security in the penalty style definition) is not as evident. Given an adversary $\mathcal{A}$

that forges a signature for a fresh message m with non-negligible probability, but that may ask the signature of the message m to the signing oracle with non-zero probability, can we construct an adversary that never queries m to the signing oracle and achieves comparable success in forging a signature? The theorem we present next proves that we can and, what is more, without any probability loss.

Theorem 2.9. *If a signature scheme is $\mathsf{EF\text{-}CMA}$ secure according to the exclusion style definition, then it is secure according to the penalty style definition.*

Proof. Let $\mathcal{A}$ be an adversary against the $\mathsf{EF\text{-}CMA}$ security of the scheme in the penalty style definition. We exhibit an adversary $\mathcal{B}$ that is valid in the exclusion style definition and outputs a successful forgery with at least the same probability as $\mathcal{A}$:

```
Adversary $\mathcal{B}(pk)$ :
  $\overline{\boldsymbol{S}} \leftarrow \mathsf{nil}$;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  if $m \in \overline{\boldsymbol{S}}$ then $m \overset{\$}{\leftarrow} \{0,1\}^k \backslash \overline{\boldsymbol{S}}$
  return $(m, \sigma)$

Oracle $\mathsf{Sign}_{\mathcal{A}}(m)$ :
  $\overline{\boldsymbol{S}} \leftarrow m :: \overline{\boldsymbol{S}}$;
  $\sigma \leftarrow \mathsf{Sign}_{\mathcal{B}}(m)$;
  return $\sigma$
```

The forger $\mathcal{B}$ uses $\mathcal{A}$ as a subroutine; it intercepts the signing queries that $\mathcal{A}$ makes and answers them using oracle $\mathsf{Sign}_{\mathcal{A}}$. This oracle just records the message queried and forwards it to the original signing oracle. When $\mathcal{A}$ outputs a purported forgery (m, σ), $\mathcal{B}$ checks if m ever appeared in a signing query and if it is the case, replaces m with a fresh message. We have that $m \notin \overline{\boldsymbol{S}}$ and $\overline{\boldsymbol{S}} = \boldsymbol{S}$ are post-conditions of game $\mathsf{G}_{\mathsf{EF}}^{\mathcal{B}}$, which implies that $\mathcal{B}$ is a valid adversary according to the exclusion style definition. In addition,

$$\Pr\left[\mathsf{G}_{\mathsf{EF}}^{\mathcal{A}} : \mathsf{Verify}(pk, m, \sigma) \wedge m \notin \boldsymbol{S}\right] \leq \Pr\left[\mathsf{G}_{\mathsf{EF}}^{\mathcal{B}} : \mathsf{Verify}(pk, m, \sigma)\right]$$

□

For a matter of taste and definitional clarity, we define $\mathsf{EF\text{-}CMA}$ security using the penalty style.

Definition 2.10 (EF-CMA security). *A signature scheme $(\mathcal{KG}, \mathsf{Sign}, \mathsf{Verify})$ is secure against existential forgeries under chosen-message attacks if the probability*

$$\Pr\left[\mathsf{G}_{\mathsf{EF}}^{\mathcal{A}} : \mathsf{Verify}(pk, m, \sigma) \wedge m \notin \boldsymbol{S}\right]$$

is negligible for any probabilistic polynomial-time adversary $\mathcal{A}$.

The penalty and exclusion style definitions of $\mathsf{EF\text{-}CMA}$ security turned out to be perfectly equivalent. Indeed, this equivalence can be regarded as folklore. This may lead us to think that there is no point in analyzing this kind of subtle differences in security definitions. But such a way of thinking is perilous. We have been lucky that both formulations of $\mathsf{EF\text{-}CMA}$ security are equivalent. We will see in the next section that being sloppy can sometimes lead to consider as equivalent definitions that in reality are not.

2.5.2 IND-CCA2 Security

The notion of IND-CCA2 security for an encryption scheme is defined relative to a two-phase experiment where the adversary has access to a decryption oracle. The experiment begins by generating a pair of keys (pk, sk) and giving the public key pk to the adversary. In the first phase the adversary chooses two messages m_0 and m_1. The challenger then tosses a fair coin b, encrypts m_b under pk and gives the resulting ciphertext $\hat{\boldsymbol{c}}$ back to the adversary. The adversary ends the second phase by outputting a guess $\tilde{b}$ for the hidden bit b. Figure 2.5 depicts this experiment as a game. We say that the scheme is IND-CCA2 secure if no probabilistic polynomial-time adversary $\mathcal{A}$ guesses b correctly with a probability non-negligibly greater than 1/2.

```
Game $\mathsf{G}_{\mathsf{INDCCA}}$ :
 $\boldsymbol{L_{\mathcal{D}}} \leftarrow \mathsf{nil}$;
 $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}(\eta)$;
 $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
 $b \xleftarrow{\$} \{0, 1\}$;
 $\hat{\boldsymbol{c}} \leftarrow \mathcal{E}(pk, m_b)$;
 $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow \mathsf{true}$;
 $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $\mathcal{D}_{\mathcal{A}}(c)$ :
 $\boldsymbol{L_{\mathcal{D}}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L_{\mathcal{D}}}$;
 $m \leftarrow \mathcal{D}(\boldsymbol{sk}, c)$;
 return $m$
```

Fig. 2.5. The IND-CCA2 experiment; the decryption oracle is instrumented to record the queries made in each phase.

Observe that the adversary could trivially win by asking the decryption oracle to decrypt $\hat{\boldsymbol{c}}$. Consequently, the definition forbids the adversary from querying $\hat{\boldsymbol{c}}$ to the decryption oracle. As in the definition of EF-CMA security in the previous section, there are two ways of enforcing this restriction, in a penalty style or in an exclusion style. In addition, we now face another dilemma: should we allow the adversary to query $\hat{\boldsymbol{c}}$ to its oracle in the first phase of the experiment or should we forbid such type of queries altogether? These two dimensions give rise to four different ways of formally defining IND-CCA2 security. Namely, in a penalty style, restricting the oracle queries only in the second phase of the experiment (IND-CCA2-SP) or in both (IND-CCA2-BP), and in an exclusion style, restricting the queries only in the second phase (IND-CCA2-SE) or in both phases (IND-CCA2-BE). There are some obvious relations between these definitions. As in the case of EF-CMA security, security in a penalty style definition implies security in the corresponding exclusion style definition. In a similar manner, security in the version of the definitions where the adversary is forbidden to query the challenge ciphertext $\hat{\boldsymbol{c}}$ just in the second phase implies security when this prohibition is extended to the first phase of the experiment. Figure 2.6 summarizes these and the remaining non-trivial relations between the different formulations of IND-CCA2 security.

Surprisingly, neither of the “B” style definitions implies security in the corresponding “S” variant. What is more, the penalty and the exclusion style definitions

$$\begin{tikzcd}
\mathsf{IND\text{-}CCA2\text{-}SP} \arrow[r, shift left] \arrow[d, shift left] & \mathsf{IND\text{-}CCA2\text{-}BP} \arrow[l, shift left, "/" marking] \arrow[d, shift left] \\
\mathsf{IND\text{-}CCA2\text{-}SE} \arrow[u, shift left] \arrow[r, shift left] & \mathsf{IND\text{-}CCA2\text{-}BE} \arrow[l, shift left, "/" marking] \arrow[u, shift left, "\backslash" marking]
\end{tikzcd}$$

Fig. 2.6. Relations between the different formulations of IND-CCA2 security. An implication $\mathsf{X} \to \mathsf{Y}$ means that security according to definition X implies security according to Y. A negated implication is a separation result.

are not equivalent if the adversary is forbidden from querying $\hat{\boldsymbol{c}}$ to its oracle in both phases of the IND-CCA2 experiment. We will give a proof of the implication IND-CCA2-SE $\to$ IND-CCA2-SP and a rough idea of how to construct pathological schemes that justify the separation results IND-CCA2-BP $\nrightarrow$ IND-CCA2-SP and IND-CCA2-BE $\nrightarrow$ IND-CCA2-BP; the separation of IND-CCA2-SE and IND-CCA2-BE follows from the diagram. For further details the reader may refer to [Bellare et al. 2009], where these results were first reported.

Theorem 2.11. *If an encryption scheme $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ is IND-CCA2-SE secure, then it is IND-CCA2-SP secure as well.*

Proof. We show that for any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the IND-CCA2-SP security of the scheme, there exists an IND-CCA2-SE adversary $\mathcal{B}$ that guesses the hidden bit b with at least the same probability and does not query the challenge ciphertext in the second phase of the experiment.

In the first phase, $\mathcal{B}$ behaves exactly as $\mathcal{A}$. When $\mathcal{B}$ gets the challenge ciphertext $\hat{\boldsymbol{c}}$, it calls $\mathcal{A}_2(\hat{\boldsymbol{c}})$ in a simulated environment where it replaces the decryption oracle with an oracle of its own; $\mathcal{B}_2$ returns whatever $\mathcal{A}_2$ returns. When $\mathcal{A}_2$ makes a decryption query c, if $c \neq \hat{\boldsymbol{c}}$, the simulated oracle responds by forwarding the query to the original oracle, otherwise it returns some fixed message $\perp$. It is easy to see that $\mathcal{B}_2$ never queries the challenge $\hat{\boldsymbol{c}}$ to the decryption oracle. Moreover, we have

$$\begin{aligned}
\Pr\left[\mathsf{G}^{\mathcal{A}}_{\mathsf{IND-CCA}} : \tilde{b} = b \wedge (\mathsf{true}, \hat{\boldsymbol{c}}) \notin \boldsymbol{L}_{\mathcal{D}}\right] &= \Pr\left[\mathsf{G}^{\mathcal{B}}_{\mathsf{IND-CCA}} : \tilde{b} = b \wedge (\mathsf{true}, \hat{\boldsymbol{c}}) \notin \boldsymbol{L}_{\mathcal{D}}\right] \\
&\leq \Pr\left[\mathsf{G}^{\mathcal{B}}_{\mathsf{IND-CCA}} : \tilde{b} = b\right]
\end{aligned}$$

which concludes the proof. □

Theorem 2.12. *The three separation results from Figure 2.6 hold.*

Proof.

IND-CCA2-BP $\nrightarrow$ IND-CCA2-SP :

Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor one-way permutations and $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ an IND-CCA2-BP secure encryption scheme. We show how to construct an encryption scheme $(\overline{\mathcal{KG}}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ which is IND-CCA2-BP secure but not IND-CCA2-SP secure.

```
$\overline{\mathcal{KG}}(\eta)$ :
  $(pk, sk) \leftarrow \mathcal{KG}(\eta)$;
  $(pk_f, sk_f) \leftarrow \mathcal{KG}_f(\eta)$;
  $\hat{x} \xleftarrow{\$} \{0,1\}^k$;
  $\hat{y} \leftarrow f(pk_f, \hat{x})$;
  $\overline{pk} \leftarrow (pk, pk_f, \hat{y})$;
  $\overline{sk} \leftarrow (sk, \hat{x})$;
  return $(\overline{pk}, \overline{sk})$

$\overline{\mathcal{E}}((pk, pk_f, \hat{y}), m)$ :
  if $f(pk_f, m) = \hat{y}$ then
    return $1 \parallel 1^k$
  else
    $c \leftarrow \mathcal{E}(pk, m)$;
    return $0 \parallel c$

$\overline{\mathcal{D}}((sk, \hat{x}), s \parallel c)$ :
  if $s = 0$ then
    return $\mathcal{D}(sk, c)$
  else
    if $c = 1^k$ then
      return $\hat{x}$
    else return $\perp$
```

The above scheme is devised in such a way that the ability of an adversary to query the challenge ciphertext in the first phase leads to an attack, but this attack is no longer possible if such a query is disallowed. The intuition is to introduce a weak message $\hat{x}$ with a single ciphertext $(1 \parallel 1^k)$. This message should be hard to compute without the secret key of the scheme, but it can be trivially obtained by asking for the decryption of $(1 \parallel 1^k)$. In the other hand, the encryption algorithm of the scheme should be able to efficiently test if a given message equals the weak message $\hat{x}$. We include the message $\hat{x}$ in plain as part of the secret key of the scheme, but conceal its value in the public key using a one-way permutation.

To show that the above scheme is not IND-CCA2-SP secure, consider the following adversary $(\mathcal{A}_1, \mathcal{A}_2)$:

```
**Adversary** $\mathcal{A}_1(\overline{pk})$ :
  $m_0 \leftarrow \mathcal{D}(1 \parallel 1^k)$;
  $m_1 \xleftarrow{\$} \{0,1\}^k \setminus \{m_0\}$;
  return $(m_0, m_1)$

**Adversary** $\mathcal{A}_2(c)$ :
  if $c = 1 \parallel 1^k$ then return 0
  else return 1
```

This adversary guesses the hidden bit in the IND-CCA2 experiment with probability 1. However, it queries the challenge ciphertext to the decryption oracle with probability 1/2 during its first phase, and therefore this adversary does not do any better than a random guess according to the winning condition of the IND-CCA2-BP variant.

To see why the scheme is IND-CCA2-BP secure, observe that the only way to guess the hidden bit b with probability significantly greater than 1/2 is to either break the security of the original encryption scheme, or to somehow obtain the value of the weak message $\hat{x}$. Indeed, given an adversary $\mathcal{B}$ that breaks the IND-CCA2-BP security of the scheme with non-negligible probability, one can construct an adversary $\mathcal{A}$ against the IND-CCA2-BP security of the original scheme and an inverter $\mathcal{I}$ for the one-way trapdoor permutation such that at least one of them succeeds with non-negligible probability [Bellare et al. 2009,Theorem 3.1].

IND-CCA2-BE $\not\rightarrow$ *IND-CCA2-BP* :

Let $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ be an encryption scheme IND-CCA2-BE secure. Again, using a trapdoor one-way permutation $(\mathcal{KG}_f, f, f^{-1})$ we construct an encryption scheme

$(\overline{\mathcal{KG}}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ which is IND-CCA2-BE secure but not IND-CCA2-BP secure.

```
$\overline{\mathcal{KG}}(\eta)$ :
  $(pk, sk) \leftarrow \mathcal{KG}(\eta)$;
  $(pk_f, sk_f) \leftarrow \mathcal{KG}_f(\eta)$;
  $\hat{x} \xleftarrow{\$} \{0,1\}^k$;
  $\hat{y} \leftarrow f(pk_f, \hat{x})$;
  $\overline{pk} \leftarrow (pk, pk_f, \hat{y})$;
  $\overline{sk} \leftarrow (sk, \hat{x})$;
  return $(\overline{pk}, \overline{sk})$
```

```
$\overline{\mathcal{E}}((pk, pk_f, \hat{y}), m)$ :
  if $f(pk_f, m) = \hat{y}$ then
    $w \xleftarrow{\$} \{0,1\}^k$;
    return $1 \parallel w$
  else
    $c \leftarrow \mathcal{E}(pk, m)$;
    return $0 \parallel c$
```

```
$\overline{\mathcal{D}}((sk, \hat{x}), s \parallel c)$ :
  if $s = 0$ then
    return $\mathcal{D}(sk, c)$
  else
    if $|c| = k$ then
      return $\hat{x}$
    else return $\perp$
```

To show that the above scheme is not IND-CCA2-BP secure, consider the following adversary $(\mathcal{A}_1, \mathcal{A}_2)$:

```
Adversary $\mathcal{A}_1(\overline{pk})$ :
  $m_0 \leftarrow \mathcal{D}(1 \parallel 1^k)$;
  $m_1 \xleftarrow{\$} \{0,1\}^k \setminus \{m_0\}$;
  return $(m_0, m_1)$
```

```
Adversary $\mathcal{A}_2(s \parallel c)$ :
  if $s = 1$ then return $0$
  else return $1$
```

This adversary guesses the hidden bit in the IND-CCA2 game with probability 1 and queries the challenge ciphertext to the decryption oracle only with probability $2^{-k}/2$. Therefore, its winning probability according to IND-CCA2-BP is $1 - 2^{-k}/2$. Observe, however, that this adversary is not valid according to the IND-CCA2-BE variant because it queries the challenge ciphertext to the decryption oracle with non-zero probability.

To see why the scheme is IND-CCA2-BE secure, observe that the only way to guess the hidden bit b with probability significantly greater than $1/2$ is to either break the IND-CCA2-BP security of the original encryption scheme, or to somehow obtain the value of the weak message $\hat{x}$. But a valid adversary cannot obtain the value $\hat{x}$ from the decryption oracle because any ciphertext of the form $(1 \parallel c)$ might be the challenge ciphertext with probability $2^{-k}/2$. Therefore, $\hat{x}$ is concealed by the one-way permutation and any adversary that succeeds in obtaining it can be used to invert the permutation with non-negligible probability.

IND-CCA2-BE $\not\rightarrow$ IND-CCA2-SE :

Follows from the above separation results and the diagram in the figure. □

In the remainder whenever we talk about IND-CCA2 security we will be referring to the IND-CCA2-SE variant of the definition, which is together with IND-CCA2-SP the strongest variant according to the taxonomy in Figure 2.6.

Definition 2.13 (IND-CCA2 security). *An encryption scheme $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ is IND-CCA2 secure if the advantage*

$$\left|\Pr\left[\mathsf{G}^{\mathcal{A}}_{\mathsf{INDCCA}} : \tilde{b} = b\right] - \frac{1}{2}\right|$$

is negligible for any probabilistic polynomial-time adversary $\mathcal{A}$ that does not query the decryption oracle with the challenge ciphertext during the second phase of the IND-CCA2 experiment.

3 Reasoning about Games

According to Shoup [2004], steps in game-based cryptographic proofs can be classified into three broad categories:

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

A bridging step from a game G_1 to a game G_2 typically replaces a program fragment c_1 by an observationally equivalent fragment c_2. In general, however, c_1 and c_2 are observationally equivalent only in the particular context where the substitution is done. We justify such transitions through a relational Hoare logic that generalizes observational equivalence through pre- and post-conditions that characterize the context where the substitution is valid. This relational Hoare logic may as well be used to establish (in)equalities between the probability of events in two games (as shown by the rules [PrEq] and [PrLe] below) and to establish program invariants that serve to justify other program transformations or more complex probabilistic reasoning.

3.1 Probabilistic Relational Hoare Logic (pRHL)

The relational Hoare logic that we propose elaborates on and extends to probabilistic programs Benton's 2004 relational Hoare logic. Benton's logic uses judgments of the form $\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi$, that relate two programs, c_1 and c_2, w.r.t. a pre-condition Ψ and a post-condition Φ, both defined as relations on deterministic states. Such a judgment states that for every pair of initial memories m_1, m_2 satisfying the pre-condition Ψ, if the evaluations of c_1 in m_1 and c_2 in m_2 terminate with final memories m_1' and m_2' respectively, then $m_1' \mathit{\Phi}\ m_2'$ holds. In a probabilistic setting, the evaluation of a program in an initial memory yields instead a (sub-)probability distribution over program memories. In order to give a meaning to a judgment like the above one, we therefore need to lift relations over memories to relations over distributions.[1] We use the mechanism presented in Section 2.1.

Definition 3.1 (pRHL judgment). *We say that two programs c_1 and c_2 are equivalent with respect to pre-condition $\mathit{\Psi}$ and post-condition $\mathit{\Phi}$ iff*

$$\vdash c_1 \sim c_2 : \mathit{\Psi} \Rightarrow \mathit{\Phi} \ \stackrel{\mathit{def}}{=}\ \forall m_1\ m_2.\ m_1\ \mathit{\Psi}\ m_2 \implies ([\![c_1]\!]\ m_1)\ \mathit{\Phi}^{\#}\ ([\![c_2]\!]\ m_2)$$

Definition 3.2 (Semantic equivalence). *We say that two programs c_1 and c_2 are semantically equivalent, and note it as $\vdash c_1 \equiv c_2$, if they are equivalent w.r.t equality on memories as pre- and post-condition.*

Rather than defining the rules for pRHL and proving them sound in terms of the meaning of judgments, we place ourselves in a semantic setting and derive the rules as lemmas. This allows to easily extend the system by deriving extra rules, or even to resort to the semantic definition if the system turns out to be insufficient. Figure 3.1 gathers some representative derived rules. To improve readability, we define for a Boolean expression e the relations

$$e\langle 1\rangle \ \stackrel{\text{def}}{=}\ \lambda m_1\ m_2.\ [\![e]\!]\ m_1 = \mathsf{true} \qquad e\langle 2\rangle \ \stackrel{\text{def}}{=}\ \lambda m_1\ m_2.\ [\![e]\!]\ m_2 = \mathsf{true}$$

As pRHL allows for arbitrary relations, we freely use higher-order logic; in particular, PER and SYM are predicates over relations that stand for *partial equivalence relation* and *symmetric relation* respectively.

Most rules admit, in addition to their symmetrical version of Figure 3.1, one-sided (left and right) variants, e.g. for assignments

$$\frac{m_1\ \mathit{\Psi}\ m_2 = (m_1\{[\![e_1]\!]m_1/x_1\})\ \mathit{\Phi}\ m_2}{\vdash x_1 \leftarrow e_1 \sim \mathsf{skip} : \mathit{\Psi} \Rightarrow \mathit{\Phi}}\text{[Assn1]}$$

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

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

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

[1] An alternative would be to develop a logic in which $\mathit{\Psi}$ and $\mathit{\Phi}$ are relations over distributions. However, we do not believe such a logic would allow a similar level of proof automation.

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

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

$$\frac{\begin{array}{l} m_1\ \Psi\ m_2 \implies ([\![d_1]\!]\ m_1)\ \Theta^{\#}\ ([\![d_2]\!]\ m_2) \\ \qquad\qquad\qquad \text{where } v_1\ \Theta\ v_2 = (m_1\{v_1/x_1\})\ \Phi\ (m_2\{v_2/x_2\}) \end{array}}{\vdash x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Psi \Rightarrow \Phi}\text{[Rnd]}$$

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

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

$$\frac{\Psi \subseteq \Psi' \quad \vdash c_1 \sim c_2 : \Psi' \Rightarrow \Phi' \quad \Phi' \subseteq \Phi}{\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi}\text{[Sub]}$$

$$\frac{\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi \quad \mathsf{SYM}(\Psi) \quad \mathsf{SYM}(\Phi)}{\vdash c_2 \sim c_1 : \Psi \Rightarrow \Phi}\text{[Sym]}$$

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

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

Fig. 3.1. Selection of derived rules of pRHL.

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

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

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

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

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

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

In order to apply the rule [Trans], we are essentially forced to have

$$\Psi_1 = \Psi_2 \qquad \text{and} \qquad \Phi_1 = \Phi_2$$

and we must also choose the same pre- and post-condition for the intermediate game c_2. This constraints make the rule [Trans] impractical in some cases; we use instead the rule [Comp] to introduce intermediate games in those cases:

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

The soundness of this rule relies on Lemma 2.3 and on the fact that the denotation of a program maps an initial memory to a distribution with countable support. This is true if we only allow values to be sampled from distributions with countable support, a reasonable restriction that does not affect our application to cryptographic proofs.

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

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

We can then prove that the following rule is sound:

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

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

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

Hence, the soundness of the above rule is immediate from the soundness of rule [Rnd]. Section 3.2.2 shows that rule [Perm] is enough to prove several program equivalences appearing in cryptographic proofs. However, observe that rule [Perm] is far from being complete as shown by the following program equivalence that cannot be derived using just this rule:

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

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

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

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

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

and analogously,

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

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

$$(\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi) \wedge m_1\ \Psi\ m_2 \implies \llbracket c_1 \rrbracket\ m_1\ \mathbb{1} = \llbracket c_2 \rrbracket\ m_2\ \mathbb{1}$$

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

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

Perhaps more surprisingly, we also have $(\llbracket c \rrbracket\ m_1)\ \neg\Phi^{\#}\ (\llbracket c \rrbracket\ m_2)$, for which it suffices to take the following distribution as a witness for the existential:

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

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

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

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

As an additional example, observe that we have

$$\begin{array}{l}\vdash x \xleftarrow{\$} \{0,1\};\ y \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\};\ y \leftarrow x : \mathsf{true} \Rightarrow =_{\{x\}}\\ \vdash x \xleftarrow{\$} \{0,1\};\ y \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\};\ y \leftarrow x : \mathsf{true} \Rightarrow =_{\{y\}}\end{array}$$

but clearly the following judgment does not hold

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

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

3.1.1 Observational Equivalence

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

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

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

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

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

Observational equivalence is more amenable to mechanization than full-fledged pRHL. To support automation, CertiCrypt implements a calculus of variable dependencies and provides a function eqobs_in, that given a program c and a set of output variables O, computes a set of input variables I such that $\vdash c \simeq_O^I c$. Analogously, it provides a function eqobs_out, that given a set of input variables

I, computes a set of output variables O such that $\vdash c \simeq_O^I c$. This suggests a simple procedure to establish a self-equivalence of the form $\vdash c \simeq_O^I c$: just compute a set I' such that $\vdash c \simeq_O^{I'} c$ using eqobs_in and check whether $I' \subseteq I$, or equivalently, compute a set O' such that $\vdash c \simeq_{O'}^I c$ using eqobs_out and check whether $O \subseteq O'$.

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

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

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

3.2 Bridging Steps

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

3.2.1 Certified Program Transformations

We automate several transformations that consist in applying compiler optimizations. More precisely, we provide support for a rich set of transformations based on dependency and dataflow analyses, and for inlining procedure calls in programs. Each transformation is implemented as a function in CertiCrypt that performs the transformation itself, together with a rule that proves its correctness and a tactic that applies the rule backwards.

Transformations based on dependencies

The functions eqobs_in and eqobs_out and the relational Hoare logic presented in Section 3.1 provide the foundations to support transformations such as dead code elimination and code reordering.

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

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

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

We provide a tactic (`swap`) that given two programs tries to hoist their common instructions to obtain a maximal common prefix[2], which can then be eliminated using the above tactics. Its correctness is based on the rule

$$\frac{\vdash c_1 \simeq^{I_1}_{O_1} c_1 \quad \vdash c_2 \simeq^{I_2}_{O_2} c_2 \quad \mathsf{modifies}(c_1, O_1) \quad \mathsf{modifies}(c_2, O_2) \qquad O_1 \cap O_2 = \emptyset \quad I_1 \cap O_2 = \emptyset \quad I_2 \cap O_1 = \emptyset}{\vdash c_1;\ c_2 \equiv c_2;\ c_1}$$

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

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

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

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

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

Optimizations based on dataflow analyses

CertiCrypt has built-in, generic, support for such optimizations: given an abstract domain D (a semi-lattice) for the analysis, transfer functions for assignment and branching instructions, and an operator that optimizes expressions in the language, we construct a certified optimization function $\mathsf{optimize} : \mathcal{C} \to D \to \mathcal{C} \times D$. When given a command c and an element $\delta \in D$, this function transforms c into its optimized version c' assuming the validity of δ. In addition, it returns an abstract post-condition $\delta' \in D$, valid after executing c (or c'). We use these abstract post-conditions to state the correctness of the optimization and to apply it recursively. The correctness of optimize is proved using a mixture of the techniques of [Benton 2004] and [Bertot et al. 2006; Leroy 2006]: we express the validity of the information contained in the analysis domain using a predicate $\mathsf{valid}(\delta, m)$ that states the agreement between the compile time abstract values in δ and the run time memory m. Correctness is expressed in terms of a pRHL judgment:

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

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

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

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

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

3.2.2 Algebraic Equivalences

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

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

In the proof of IND-CCA2 security of OAEP described in Section 6.1 we use the equivalences

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

and (for a permutation f):

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

We show the usefulness of rule [Perm] by proving the first of these two equivalences, known as *optimistic sampling*, that we also used in §1.4.2. Define

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

By rule [Assn] we have

$$\vdash y \leftarrow x \oplus z \sim x \leftarrow y \oplus z : \Theta \Rightarrow \Phi \tag{3.1}$$

We apply rule [Perm] to prove

$$\vdash x \stackrel{\$}{\leftarrow} \{0,1\}^k \sim y \stackrel{\$}{\leftarrow} \{0,1\}^k : \Psi \Rightarrow \Theta \tag{3.2}$$

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

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

Take $f(v) \stackrel{\text{def}}{=} v \oplus m_2(z)$ to be such a permutation. Conclude from (3.1) and (3.2) by a final application of rule [Seq].

3.2.3 Inter-procedural Code Motion

Game-based proofs commonly include bridging steps consisting in a semantics-preserving reordering of instructions. When the reordering is intra-procedural, the tactic `swap` presented in the previous section generally suffices to justify the transformation. However, proofs in the random oracle model (see §1.4.2 for an example of a random oracle) often include transformations where random values used inside oracles are sampled beforehand, or conversely, where sampling a random value at some point in a game is deferred to a later point, possibly in a different procedure. The former type of transformation, called eager sampling, is useful for moving random choices upfront: a systematic application of eager sampling transforms a probabilistic game G that samples a fixed number of values into a semantically equivalent game $S; G'$, where S samples the values that might be needed in G, and G' is a completely deterministic program to the exception of adversaries that may still make their own random choices.[3] The dual transformation, called lazy sampling, can be used to postpone sampling random values until they are actually used for the first time—thus, one readily knows the exact distribution of these values by reasoning locally, without the need to maintain and reason about probabilistic invariants. In this section, we present a general method to prove the correctness of inter-procedural code motion. The method is based on a logic for swapping statements that generalizes the earlier lemma reported in [Barthe et al. 2009c].

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, we do not constrain S and merely require that it satisfies three basic properties for some sets of variables X and I:

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

Some rules of the logic are given in Figure 3.2; 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,

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

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

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

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

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

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

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

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

Fig. 3.2. Selected rules of a logic for swapping statements.

An application

Consider the games $\mathsf{G}_{\mathsf{lazy}}$ and $\mathsf{G}_{\mathsf{eager}}$ in Figure 3.3. Both games define an oracle

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

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

```
Game G_eager :
 $\boldsymbol{L} \leftarrow \mathsf{nil};\ \hat{\boldsymbol{y}} \xleftarrow{\$} \{0,1\}^\ell;\ b \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}_{\mathrm{eager}}(x)$ :
 if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
   if $x = 0^k$ then $y \leftarrow \hat{\boldsymbol{y}}$ else $y \xleftarrow{\$} \{0,1\}^\ell$;
   $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
 else $y \leftarrow \boldsymbol{L}[x]$
 return $y$
```

Fig. 3.3. An example of eager sampling justified by inter-procedural code motion.

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

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

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

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

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

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

We are thus left to show

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

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

$$\begin{aligned}
e &= e' = x \notin \mathsf{dom}(\boldsymbol{L})\\
c_1 &= y \xleftarrow{\$} \{0,1\}^{\ell};\ \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}\\
c'_1 &= (\mathsf{if}\ x = 0^k\ \mathsf{then}\ y \leftarrow \hat{\boldsymbol{y}}\ \mathsf{else}\ y \xleftarrow{\$} \{0,1\}^{\ell});\ \boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}\\
c_2 &= c'_2 = y \leftarrow \boldsymbol{L}[x]
\end{aligned}$$

There are two non-trivial proof obligations:

1. $\vdash c_2; S \sim S; c'_2 : =_{\mathcal{V}} \wedge\ (x \in \mathsf{dom}(\boldsymbol{L}))\langle 1\rangle \Rightarrow =_{\mathcal{V}}$
   This corresponds to showing that the code in the *else* branch in the conditional of each implementation of $\mathcal{O}$ commutes with S, and follows from [S-Assn];
2. $\vdash c_1; S \sim S; c'_1 : =_{\mathcal{V}} \wedge\ (x \notin \mathsf{dom}(\boldsymbol{L}))\langle 1\rangle \Rightarrow =_{\mathcal{V}}$
   By case analysis on $x = 0^k$:
   a) If $x = 0^k$, we can invoke certified program transformations—using the precondition that $x \notin \mathsf{dom}(\boldsymbol{L})$—to simplify the goal to the following easily provable form:

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

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

3.3 Reasoning about Failure Events

One common technique to justify a *lossy* transformation $G, A \to G', A$, where $\Pr[G : A] \neq \Pr[G' : A]$ is based on what cryptographers call *failure events*. This technique relies on a *fundamental lemma* that allows to bound the difference in the probability of an event in two games: one identifies a failure event and argues that both games behave identically until failure occurs. One can then bound the difference in probability of another event by the probability of failure in either game. Consider for example the following two program snippets and their instrumented versions:

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

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

Lemma 3.3 (Fundamental Lemma). *Let G_1, G_2 be two games and let A, B, and F be events. If* $\Pr[G_1 : A \wedge \neg F] = \Pr[G_2 : B \wedge \neg F]$*, then*

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

Proof.

$$\begin{array}{ll} \multicolumn{2}{l}{|\Pr[G_1 : A] - \Pr[G_2 : B]\,|} \\ \quad = & |\Pr[G_1 : A \wedge F] + \Pr[G_1 : A \wedge \neg F] - \Pr[G_2 : B \wedge F] - \Pr[G_2 : B \wedge \neg F]\,| \\ \quad = & |\Pr[G_1 : A \wedge F] - \Pr[G_2 : B \wedge F]\,| \\ \quad \leq & \max(\Pr[G_1 : A \wedge F], \Pr[G_2 : B \wedge F]) \\ \quad \leq & \max(\Pr[G_1 : F], \Pr[G_2 : F]) \end{array}$$

□

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

3.3.1 A Logic for Bounding the Probability of Events

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

$$\vdash [\![c]\!] g \preceq f \quad \stackrel{\mathrm{def}}{=} \quad \forall m.\ [\![c]\!]\ m\ g \le f\ m$$

Figure 3.4 gathers some rules for proving the validity of such triples. The rule for adversary calls assumes that f depends only on variables that the adversary cannot modify directly (but may modify indirectly through oracle calls, of course). The correctness of this rule is proved using the induction principle for well-formed adversaries together with the rest of the rules of the logic.

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

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

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

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

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

Fig. 3.4. Selected rules of a logic for bounding the probability of events.

The rules bear some similarity with the rules of (standard) Hoare logic. However, there are some subtle differences. For example, the premises of the rules for branching statements do not consider guards. The rule

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

where $f_{|e}$ is defined as $(\lambda m.$ if $[\![e]\!] m$ then $f(m)$ else $0)$ can be derived from the rule for conditionals in the figure by two simple applications of the “rule of consequence”. Moreover, the rule for conditional statements (and its variant above) is incomplete: consider a statement of the form $[\![\mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ such that $[\![c_1]\!] g \preceq f$ is valid, but not $[\![c_2]\!] g \preceq f$; the triple $[\![\mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ 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.[4]

Digression

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

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

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

3.3.2 Automation

In most applications of Lemma 3.3, failure can only be triggered by oracle calls. Typically, the flag **bad** that signals failure is set in the code of an oracle for which an upper bound for the number of queries made by the adversary is known. The following lemma provides a general method for bounding the probability of failure under such circumstances.

Lemma 3.4 (Failure Event Lemma). *Consider an event F and a game G that gives adversaries access to an oracle $\mathcal{O}$. Let $\mathsf{cntr} : \mathcal{E}_{\mathsf{Nat}}$, $h : \mathbb{N} \to [0,1]$ be such that cntr and F do not depend on variables that can be written outside $\mathcal{O}$, and for any initial memory m,*

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

and

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

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

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

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

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

$$f(m) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } [\![\mathsf{cntr}]\!]\ m > q \\ \mathbb{1}_F(m) + \mathbb{1}_{\neg F}(m) \displaystyle\sum_{i=[\![\mathsf{cntr}]\!] m}^{q-1} h(i) & \text{if } [\![\mathsf{cntr}]\!]\ m \le q \end{cases}$$

We show $[\![G]\!] f \preceq f$ 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 $[\![\mathcal{O}.\mathsf{body}]\!] f \preceq f$. We must show that for every m, $[\![\mathcal{O}.\mathsf{body}]\!]\ m\ f \le f(m)$. This is trivial when cntr is not incremented, because we have

$$[\![\mathcal{O}.\mathsf{body}]\!]\ m\ f = f(m)\ ([\![\mathcal{O}.\mathsf{body}]\!]\ m\ \mathbb{1}) \le f(m)$$

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

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

Using the rules in Figure 3.4, 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

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

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 $F = \textbf{bad}$ and defining h suitably. In most practical applications the probability of an oracle call raising failure is history-independent and hence h is a constant function. The proof of Lemma 4.3 given in Section 4.3.2 is an exception for which the full generality of the lemma is needed.

4 The PRP/PRF Switching Lemma

Cryptographic systems are generally built incrementally by combining basic primitives with the goal of achieving a higher level security goal. Rather than designing a system for a particular choice of a primitive, one designs the system assuming a generic and simplified model of the primitive. The security of the whole system is then analyzed under the assumption that this model behaves in an ideal way. Since in practice the construction that implements the primitive will definitely deviate from this ideal behavior, the actual security of the system depends on how wide the gap between the idealized and the actual behavior is. Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that are used to model blockciphers and thus 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 is used to fill 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.

In this Chapter we will formally define the notions of pseudorandom function and pseudorandom permutation and their security, and we will overview two different game-based proofs of the PRP/PRF switching lemma. Both use the Fundamental Lemma of game-playing (Lemma 3.3) to bound the advantage of an adversary by the probability of a failure event, but each proof bounds the probability of failure using a different technique. We first present a proof that uses the principle of eager sampling so that all random choices are done up front and the probability is directly computable. We then present a significantly more compact proof that uses Lemma 3.4 (see §3.3.2) to bound the probability of failure.

4.1 Pseudorandom Functions

A pseudorandom function is a key-indexed family of functions $\{f_k \mid k \in K\}$ with the property that an instance selected at random according to some distribution on K is computationally indistinguishable from a perfectly random function. Unless otherwise said we will consider that the distribution on K that makes this property hold is the uniform distribution.

Consider an adversary who has only blackbox access to an oracle and is put in either of two scenarios: one where the oracle is a random instance of a function drawn from a family of pseudorandom functions, and other where the oracle is a perfectly random function. This adversary should only be able to tell apart both scenarios with a small probability. We can define this formally using games.

Definition 4.1 (PRF-advantage). *Let $\{f_k : A \to B \mid k \in K\}$ be a pseudorandom function family, and $\mathcal{A}$ an adversary with blackbox access to an oracle $\mathcal{O}$ as in the following two games:*

```
Game G_PRF :
 $\boldsymbol{k} \xleftarrow{\$} K;\ b \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
 return $f_{\boldsymbol{k}}(x)$
```

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

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

The PRF-advantage of $\mathcal{A}$ against f is defined as

$$\mathbf{Adv}^{\mathcal{A}}_{\mathsf{PRF}_f} \overset{def}{=} |\Pr[\mathsf{G}_{\mathsf{PRF}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]|$$

The concept of pseudorandom function was first introduced by Goldreich, Goldwasser, and Micali [1986]. Rather than considering a single family of key-indexed functions, they consider a collection of families parametrized by a security parameter η. In this asymptotic setting, a pseudorandom function is secure if all adversaries that execute in polynomial-time on η have a negligible PRF-advantage (as a function of η). In contrast, in the setting of exact security there is no absolute notion of security for pseudorandom functions. The above definition only associates to each adversary $\mathcal{A}$ a real number, its PRF-advantage. In practice one considers all adversaries consuming no more than a certain amount of computational resources, and gives an upper bound for their PRF-advantage.

4.2 Pseudorandom Permutations

A pseudorandom permutation is key-indexed family of permutations $\{f_k \mid k \in K\}$ on A such that a permutation randomly drawn from the family is computationally indistinguishable from a permutation drawn uniformly from the set of all permutations on A. Again, we define this notion formally using games.

Definition 4.2 (PRP-advantage). *Let $\{f_k : A \to A \mid k \in K\}$ be a pseudorandom permutation family, and $\mathcal{A}$ an adversary with blackbox access to an oracle $\mathcal{O}$ as in the following two games:*

```
Game G_PRP :
 $\boldsymbol{k} \xleftarrow{\$} K$; $b \leftarrow \mathcal{A}()$

Oracle $\mathcal{O}(x)$ :
 return $f_{\boldsymbol{k}}(x)$
```

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

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

where the instruction $y \xleftarrow{\$} A \setminus \mathsf{ran}(\boldsymbol{L})$ samples uniformly an element of A that is not in the range of the association list $\boldsymbol{L}$, thus ensuring that oracle $\mathcal{O}$ in G_{RP} implements an injective—and therefore bijective—function. The PRP-advantage of $\mathcal{A}$ against f is defined as

$$\mathbf{Adv}^{\mathcal{A}}_{\mathsf{PRP}_f} \overset{def}{=} |\Pr[\mathsf{G}_{\mathsf{PRP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RP}} : b = 1]|$$

The notion of pseudorandom permutation is due to Luby and Rackoff [1988], who also observe that the notions of pseudorandom function and permutation are no different in an asymptotic setting, and show how to construct a pseudorandom permutation from a pseudorandom function.

4.3 The PRP/PRF Switching Lemma

We already observed that every pseudorandom permutation family is also a pseudorandom function family. But how well does a pseudorandom permutation perform as a pseudorandom function? Let us first consider the simpler problem of comparing a perfectly random function to a random permutation. Suppose you give to an adversary blackbox access to an oracle implementing either a random function or a random permutation, and you ask it 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 $\{0,1\}^\ell$. Due to the birthday problem, no matter the strategy the adversary follows, after roughly $2^{\ell/2}$ queries to the oracle it will be able to tell in which scenario it 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 us a lower bound for the advantage of an adversary in distinguishing a random function from a random permutation. The following lemma gives an upper bound.

Lemma 4.3 (PRP/PRF switching lemma). *Let $\mathcal{A}$ be an adversary with blackbox 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 oracle $\mathcal{O}$. Then,*

$$|\Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \le \frac{q(q-1)}{2^{\ell+1}} \tag{4.1}$$

We overview two different machine-checked proofs of the PRP/PRF switching lemma that exploit the code-based techniques presented in earlier sections. Both proofs use the Fundamental Lemma to bound the advantage of the adversary by the probability of a failure event. The first proof uses the eager sampling technique of Section 3.2.3 to bound the probability of failure, whereas the second one relies on Lemma 3.4 of Section 3.3.2. We begin by introducing in Figure 4.1 annotated versions $\mathsf{G}^{\mathbf{bad}}_{\mathsf{RP}}$ and $\mathsf{G}^{\mathbf{bad}}_{\mathsf{RF}}$ of the games G_{RP} and G_{RF}. These annotated games set a flag **bad** whenever the oracle corresponding to a random function would return a value colliding with a response to a previous query, but are otherwise semantically equivalent to the original games. The annotated games are syntactically identical until the point where **bad** is set, so we can appeal to Lemma 3.3 to bound the difference in the probability of b being equal to 1 in the original games:

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

```
Game $\mathsf{G}^{\mathbf{bad}}_{\mathsf{RP}}$ :
 $\boldsymbol{L} \leftarrow$ nil; $b \leftarrow \mathcal{A}()$
Oracle $\mathcal{O}(x)$ :
 if $x \notin$ dom($\boldsymbol{L}$) then
   $y \xleftarrow{\$} \{0,1\}^\ell$;
   if $y \in$ ran($\boldsymbol{L}$) then
     bad $\leftarrow$ true;
     $y \xleftarrow{\$} \{0,1\}^\ell \setminus$ ran($\boldsymbol{L}$)
   $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
 return $\boldsymbol{L}[x]$
```

```
Game $\mathsf{G}^{\mathbf{bad}}_{\mathsf{RF}}$ :
 $\boldsymbol{L} \leftarrow$ nil; $b \leftarrow \mathcal{A}()$
Oracle $\mathcal{O}(x)$ :
 if $x \notin$ dom($\boldsymbol{L}$) then
   $y \xleftarrow{\$} \{0,1\}^\ell$;
   if $y \in$ ran($\boldsymbol{L}$) then
     bad $\leftarrow$ true
   $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
 return $\boldsymbol{L}[x]$
```

```
Game $\mathsf{G}^{\mathsf{eager}}_{\mathsf{RF}}$ :
 $\boldsymbol{L} \leftarrow$ nil; $S$; $b \leftarrow \mathcal{A}()$
Oracle $\mathcal{O}(x)$ :
 if $x \notin$ dom($\boldsymbol{L}$) then
   if $0 < |\boldsymbol{Y}|$ then
     $y \leftarrow$ hd($\boldsymbol{Y}$);
     $\boldsymbol{Y} \leftarrow$ tl($\boldsymbol{Y}$)
   else $y \xleftarrow{\$} \{0,1\}^\ell$
   $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
 return $\boldsymbol{L}[x]$
```

$S \overset{\text{def}}{=} \boldsymbol{Y} \leftarrow \mathsf{nil};\ \mathsf{while}\ |\boldsymbol{Y}| < q\ \mathsf{do}\ \left(y \xleftarrow{\$} \{0,1\}^\ell;\ \boldsymbol{Y} \leftarrow y :: \boldsymbol{Y}\right)$

Fig. 4.1. Games used in the proofs of the PRP/PRF Switching Lemma.

4.3.1 A Proof Based on Eager Sampling

We make a first remark: the probability of **bad** being set in game $\mathsf{G}^{\mathbf{bad}}_{\mathsf{RF}}$ 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 $\mathsf{col}(\boldsymbol{L})$,

$$\mathsf{col}(\boldsymbol{L}) \overset{\text{def}}{=} \exists x_1, x_2 \in \mathsf{dom}(\boldsymbol{L}).\ x_1 \neq x_2 \ \wedge\ \boldsymbol{L}[x_1] = \boldsymbol{L}[x_2]$$

We prove that

$$\vdash \mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}} \sim \mathsf{G}_{\mathsf{RF}} : \mathsf{true} \Rightarrow \mathbf{bad}\langle 1\rangle \implies \mathsf{col}(\boldsymbol{L})\langle 2\rangle$$

Thus,

$$\Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathbf{bad}} : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_{\mathsf{RF}} : \mathsf{col}(\boldsymbol{L})\right] \tag{4.2}$$

Using the logic for swapping statements, we then modify the oracle in G_{RF} 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 Figure 4.1; each time a query is made, the oracle pops a value from list $\boldsymbol{Y}$ and gives it back to the adversary as the response. Since the initialization code S terminates and does not modify $\boldsymbol{L}$, we can conclude that

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

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 bounded by the probability of having a collision in $\boldsymbol{Y}$ immediately after executing S. We conclude that the bound in (4.1) holds by analyzing the loop in S.

Observe that if there are no collisions in $\boldsymbol{Y}$ in a memory m, we can prove by induction on $(q - |\boldsymbol{Y}|)$ that the probability of sampling a colliding value in the remaining loop iterations is

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

We thus have,

$$\begin{aligned}\Pr\left[\mathsf{G}_{\mathsf{RF}}^{\text{eager}} : \mathsf{col}(L)\right] &\leq \Pr\left[S, m\{\mathsf{nil}/\boldsymbol{Y}\} : \exists i, j \in \mathbb{N}.\ i < j < q\ \wedge\ \boldsymbol{Y}[i] = \boldsymbol{Y}[j]\right] \\ &= \frac{q(q-1)}{2^{\ell+1}}\end{aligned}$$

□

4.3.2 A Proof Based on the Failure Event Lemma

The bound in (4.1) follows from a direct application of Lemma 3.4. It suffices to take $F = \mathbf{bad}$, $h(i) = i\,2^{-\ell}$, and $\mathsf{cntr} = |\boldsymbol{L}|$. If $\mathbf{bad}$ is initially set to false in memory m, we have

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

The first equation holds because $\mathcal{A}$ does not make more than q queries to $\mathcal{O}$. The inequality is obtained from Lemma 3.4; we use the logic of Section 3.3.1 to bound the probability of $\mathbf{bad}$ being set in one call to the oracle by $h(\mathsf{cntr})$.

4.3.3 Comparison of Both Proofs

The proof of the PRP/PRF switching lemma that bounds the probability of failure using Lemma 3.4 presented in Section 4.3.2 is considerably shorter compared to the one presented in Section 4.3.1, that uses the principle of eager sampling to reduce the problem of bounding the probability of failure to local reasoning about a loop. The former proof takes just about 100 lines of Coq, compared to the 400 lines that takes the latter. Both proofs are significantly more compact than the 900-lines proof reported in Barthe et al. [2009c]. That proof used an earlier mechanization of the eager sampling technique that only allowed to fix the value of one response of the oracle at a time. Thus, in order to fix in advance the response to all the q queries that could be made by the adversary, an induction argument was necessary.

4.4 Pseudorandom Permutations as Pseudorandom Functions

In view of Lemma 4.3, we can now answer our original question: how well does a pseudorandom permutation perform as a pseudorandom function?

Let $\{f_k \mid k \in K\}$ be a pseudorandom permutation family on $\{0,1\}^\ell$ and let $\mathcal{A}$ be an adversary that makes at most $q > 0$ queries to its oracle. The PRF-advantage of $\mathcal{A}$ is

$$\begin{aligned}
\mathbf{Adv}^{\mathcal{A}}_{\mathsf{PRF}_f} &= |\Pr[\mathsf{G}_{\mathsf{PRF}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \\
&= |\Pr[\mathsf{G}_{\mathsf{PRP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] + \Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \\
&\le |\Pr[\mathsf{G}_{\mathsf{PRP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RP}} : b = 1]| + |\Pr[\mathsf{G}_{\mathsf{RP}} : b = 1] - \Pr[\mathsf{G}_{\mathsf{RF}} : b = 1]| \\
&\le \mathbf{Adv}^{\mathcal{A}}_{\mathsf{PRP}_f} + \frac{q(q-1)}{2^{\ell+1}}
\end{aligned}$$

Consider this bound in an asymptotic setting, and assume ℓ is linearly proportional to the security parameter η. Every polynomial-time adversary can make only a polynomial number of queries to its oracle, so q is polynomial on the security parameter, and the term on the right hand side of the last inequality is negligible on η if f is secure as a pseudorandom permutation. Hence, f is secure as a pseudorandom function whenever f is secure as a pseudorandom permutation.

We already observed that the construction of Luby and Rackoff [1988] provides a means to build a pseudorandom permutation from a pseudorandom function; many other authors have studied variations of this construction. In contrast, the reverse direction has been historically much less studied. Although it follows from the PRP/PRF switching lemma that in a complexity-theoretical setting, a pseudorandom permutation *is* a pseudorandom function, there are constructions that achieve better security at a low efficiency cost [Hall et al. 1998].

4.5 Discussion

Despite the apparent simplicity of the PRP/PRF switching lemma, some purported proofs in the literature contain a subtle error in reasoning about conditional probabilities (cf. Impagliazzo and Rudich [1989]). Let us briefly report the argument in those proofs.

Intuitive proof.

Let $\mathsf{collision}$ be the event that adversary $\mathcal{A}$ gets the same answer to two different queries when interacting with a random function. Since a random permutation behaves the same as a random function when no collisions are observed, we have that

$$\Pr\left[\mathsf{G_{RP}} : b = 1\right] = \Pr\left[\mathsf{G_{RF}} : b = 1 \mid \neg\mathsf{collision}\right] \tag{4.3}$$

Let $x = \Pr\left[\mathsf{G_{RF}} : b = 1 \mid \neg\mathsf{collision}\right]$, $y = \Pr\left[\mathsf{G_{RF}} : b = 1 \mid \mathsf{collision}\right]$. Then,

$$\begin{aligned}
&\left|\Pr\left[\mathsf{G_{RP}} : b = 1\right] - \Pr\left[\mathsf{G_{RF}} : b = 1\right]\right| \\
&\qquad = \left|x - \left(x \Pr\left[\mathsf{G_{RF}} : \neg\mathsf{collision}\right] + y \Pr\left[\mathsf{G_{RF}} : \mathsf{collision}\right]\right)\right| \\
&\qquad = \left|x \left(1 - \Pr\left[\mathsf{G_{RF}} : \neg\mathsf{collision}\right]\right) - y \Pr\left[\mathsf{G_{RF}} : \mathsf{collision}\right]\right| \\
&\qquad = |x - y| \Pr\left[\mathsf{G_{RF}} : \mathsf{collision}\right]
\end{aligned}$$

Since $0 \le x, y \le 1$,

$$\left|\Pr\left[\mathsf{G_{RP}} : b = 1\right] - \Pr\left[\mathsf{G_{RF}} : b = 1\right]\right| \le \Pr\left[\mathsf{G_{RF}} : \mathsf{collision}\right]$$

Since the adversary makes at most q queries to the oracle,

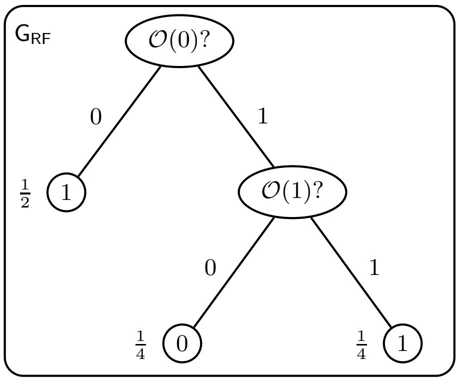
$$\Pr\left[\mathsf{G_{RF}} : \mathsf{collision}\right] \le \frac{q(q-1)}{2^{\ell+1}}$$

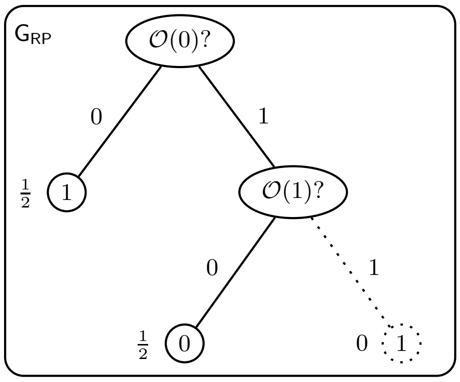
and the bound (4.1) follows. □

The reader may be wondering where is the error in the above argument. The problem is that equation (4.3) might not hold, no matter how appealing the intuitive justification we gave can look. We can actually illustrate this with a counterexample for $\ell = 1$, by showing a particular adversary for which the equation does not hold:

```
Adversary $\mathcal{A}_1()$ :
 $y \leftarrow \mathcal{O}(0)$;
 if $y = 0$ then return 1
 else
   $y \leftarrow \mathcal{O}(1)$;
   if $y = 1$ then return 1 else return 0
```

Let us analyze how this adversary fares in each game with respect to equation (4.3). We can better depict the behavior of the adversary using a tree. The values in leaves represent the bit b the adversary returns in each case whereas the label on the left of a node indicates the probability of reaching it.





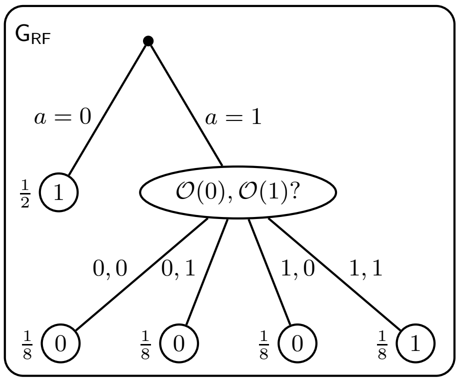
It follows that

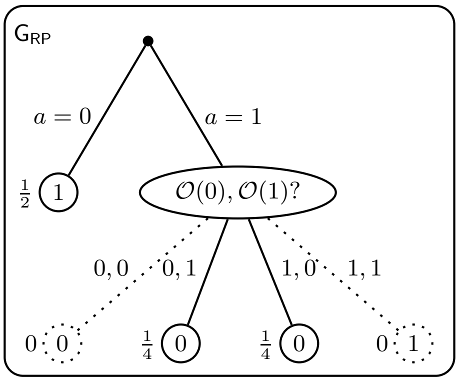
$$\frac{1}{2} = \Pr\left[\mathsf{G_{RP}} : b = 1\right] \quad \neq \quad \Pr\left[\mathsf{G_{RF}} : b = 1 \mid \neg\mathsf{collision}\right] = \frac{2}{3} \tag{4.4}$$

The reason of the discrepancy is that the number of queries made by the adversary varies depending on the answer it receives from its first query.

The same would happen if the number of queries depended on the internal random choices of $\mathcal{A}$. For instance, the following adversary makes either zero or two queries depending on the result of sampling a fair coin and achieves the same probabilities in equation (4.4) as the adversary we showed previously:

```
**Adversary** $\mathcal{A}_2()$ :
  $a \xleftarrow{\$} \{0,1\}$;
  if $a = 0$ then return $1$
  else
    $y_0 \leftarrow \mathcal{O}(0)$; $y_1 \leftarrow \mathcal{O}(1)$;
    if $y_0 \wedge y_1$ then return $1$ else return $0$
```





Obviously, the result we proved in the previous section holds for both of the above adversaries. We can compute the actual advantage they achieve and check that the bound in Equation (4.1) holds:

$$\left|\Pr\left[\mathsf{G}_{\mathsf{RP}}^{\mathcal{A}_1} : b = 1\right] - \Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathcal{A}_1} : b = 1\right]\right| = \left|\frac{1}{2} - \frac{3}{4}\right| = \frac{1}{4} \leq \frac{q(q-1)}{2^{\ell+1}} = \frac{1}{2}$$

$$\left|\Pr\left[\mathsf{G}_{\mathsf{RP}}^{\mathcal{A}_2} : b = 1\right] - \Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathcal{A}_2} : b = 1\right]\right| = \left|\frac{1}{2} - \frac{5}{8}\right| = \frac{1}{8} \leq \frac{q(q-1)}{2^{\ell+1}} = \frac{1}{2}$$

Indeed, both adversaries make at most $q = 2$ queries and the bound holds. The following adversary $\mathcal{A}_3$ mounting a birthday attack with $q = 2$ achieves the maximal value of 1/2:

```
**Adversary** $\mathcal{A}_3()$ :
 $y_1 \leftarrow \mathcal{O}(0)$;
 $y_2 \leftarrow \mathcal{O}(1)$;
 if $y_1 \neq y_2$ then return 1 else return 0
```

$$\left|\Pr\left[\mathsf{G}_{\mathsf{RP}}^{\mathcal{A}_3} : b = 1\right] - \Pr\left[\mathsf{G}_{\mathsf{RF}}^{\mathcal{A}_3} : b = 1\right]\right| = \left|1 - \frac{1}{2}\right| = \frac{1}{2}$$

4.6 Related Work

The standard *proof* of the PRP/PRF switching lemma is due to Impagliazzo and Rudich [1989,Theorem 5.1]. The above observation about the error in the reasoning in the standard proof of the lemma and the first counterexample we showed are due to Bellare and Rogaway [2006].

Bellare and Rogaway [2006,Lemma 1] give a game-based proof of the PRP/PRF switching lemma. Their proof is similar to ours, but they make the additional assumption that the adversary never asks an oracle query twice. Just as in the proof we presented, they use the Fundamental Lemma of game-playing to bound the difference in the probability of the adversary outputting 1 when interacting with either a random permutation or a random function. However, the justification of the bound on the probability of the **bad** flag being set when the adversary interacts with a random function remains informal. The same authors give also a proof of the PRP/PRF switching lemma that does not use games, under the assumption that the adversary is deterministic and makes exactly q different queries to its oracle. We note that the assumption of the adversary being deterministic is without loss of generality only if it is computationally unbounded, and therefore the argument does not hold in an asymptotic setting where the adversary must execute in polynomial-time.

Shoup [2004,Section 5.1] gives a game-based proof of the PRP/PRF switching lemma under the assumption that the adversary makes exactly q distinct queries to its oracle. In the games he considers, the challenger acts as intermediary between the oracle and the adversary. Rather than the adversary calling the oracle at its discretion, it is the challenger that calls the adversary to get the next query; it then asks the query itself to the oracle and gives the response back the adversary in the next call. There is probably nothing wrong with this formulation, but we feel that it imposes unnecessary restrictions on the form of the adversary that do not help to make the proof more clear.

Affeldt, Tanaka, and Marti [2007] present a formalization of a game-based proof of the PRP/PRF switching lemma in the Coq proof assistant. 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. For instance, the two adversaries we gave in the previous section as counterexamples to the probabilistic reasoning in the original proof of the PRP/PRF switching lemma are ruled out by this formulation.

5 Unforgeability of Full-Domain Hash Signatures

In this chapter we will go through the formalization in CertiCrypt of two different proofs of security of the Full-Domain Hash (FDH) signature scheme. The FDH scheme was first proposed by Bellare and Rogaway [1996] as an efficient RSA-based signature scheme, but is in fact an instance of an earlier construction described by the same authors in 1993. Here, we will consider this latter, more general construction, which is based on a family of one-way trapdoor permutations f on a cyclic group G, and a hash function $H : \{0,1\}^* \to G$ whose range is the full domain of f. The RSA-based scheme is obtained by instantiating f with the RSA function, and the hash function with some cryptographic hash function, such as SHA-1 with the length of its output extended to match that of the RSA modulus.

Definition 5.1 (Trapdoor permutation). *A family of trapdoor permutations is a triple of algorithms $(\mathcal{KG}, f, f^{-1})$. For a given value of the security parameter η, the key generator $\mathcal{KG}(\eta)$ randomly selects a pair of keys (pk, sk) such that $f(pk,\cdot)$ is a permutation on its domain and $f^{-1}(sk,\cdot)$ is its inverse. We say that a family of trapdoor permutations is* one-way *if it cannot be inverted in probabilistic polyonomial-time on a uniformly distributed element in its domain.*

Definition 5.2 (Full-Domain Hash signature scheme). *Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor permutations on cyclic groups G_η and let H be family of hash functions from bitstrings of arbitrary length onto the domain of the permutations. The Full-Domain Hash digital signature scheme is composed of the following triple of algorithms:*

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

The key generation algorithm just runs the key generation algorithm of the underlying trapdoor permutation obtaining a public key pk, that is used as the verification *key of the scheme, and a secret key (trapdoor) sk used to compute signatures. The signature of a message $m \in \{0,1\}^*$ is simply $f^{-1}(sk, H(m))$, the preimage under f of its digest. To verify a purported signature σ on a message m, it suffices to check whether $H(m)$ and $f(pk, \sigma)$ coincide.*

The FDH scheme can be proved secure in the random oracle model against existential forgery under adaptive chosen-message attacks (see Definition 2.10 in §2.5). This means that if we regard the hash function H as a truly random function, then any computationally feasible adversary with access to the public key and that can ask for the signature of messages of its choice, succeeds in forging a signature for a fresh message only with a negligible probability. This asymptotic security statement is desirable, but of limited practical utility because it does not give any hint as to how to choose the scheme parameters to attain a certain degree of security. A much more useful result would be an exact security statement, a bound that quantifies the gap between the security of the scheme and the intractability of inverting the trapdoor permutation.

Consider an adversary against the existential unforgeability of FDH that makes at most $q_{\mathsf{H}}(\eta)$ and $q_{\mathsf{S}}(\eta)$ queries to the hash and signing oracles, respectively. In a code-based setting, such an adversary is regarded as a black-box procedure $\mathcal{A}$ run in the context of the following attack game:

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

```
Oracle $H(m)$ :
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} G$; $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$
  return $\boldsymbol{L}[m]$

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

This adversary succeeds in forging a FDH signature for a fresh message with probability

$$\Pr\left[\mathsf{G_{EF}} : h = f(pk, \sigma) \land m \notin \boldsymbol{S}\right]$$

Note that in the above game the signing oracle makes a hash query each time the adversary asks for the signature of a message, and an additional hash query is made at the end as part of the verification of the signature returned by the adversary. Thus the number of effective hash queries made during the whole game is at most $q_{\mathsf{H}} + q_{\mathsf{S}} + 1$. All this is captured by the following post-condition of $\mathsf{G_{EF}}$,

$$\Psi \overset{\text{def}}{=} |\boldsymbol{L}| \leq q_{\mathsf{H}} + q_{\mathsf{S}} + 1 \ \land \ |\boldsymbol{S}| \leq q_{\mathsf{S}}$$

This implies in particular that $\Pr\left[\mathsf{G_{EF}} : A\right] = \Pr\left[\mathsf{G_{EF}} : A \land \Psi\right]$ for any event A.

In the remainder of this chapter we will show two different ways of constructing an inverter $\mathcal{I}$ that uses the forger $\mathcal{A}$ to invert f. These constructions effectively reduce the security of the signature scheme to the intractability of inverting the underlying trapdoor permutation.

5.1 The Original Proof

We first present a game-based proof of the original result by Bellare and Rogaway [1993], which provides a security bound that depends on the number of queries the adversary makes to both, the hash and the signing oracle.

Theorem 5.3 (Original bound). *Let $\mathcal{A}$ be an adversary mounting a chosen-message existential forgery attack against FDH that makes at most q_{H} queries to the hash oracle H and at most q_{S} queries to the signing oracle* Sign. *Suppose $\mathcal{A}$ succeeds in forging a signature for a fresh message within time t with probability ϵ. Then, there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f with probability ϵ' within time t', where*

$$\epsilon' \geq (q_{\mathsf{H}} + q_{\mathsf{S}} + 1)^{-1}\ \epsilon \tag{5.1}$$

$$t' \leq t + (q_{\mathsf{H}} + q_{\mathsf{S}})\ O(t_f) \tag{5.2}$$

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

Proof. The inverter $\mathcal{I}$ shown in the context of game $\mathsf{G_{OW}}$ in Figure 5.1 achieves the probability and time bounds in the statement. It simulates an environment for $\mathcal{A}$ where it replaces the hash and signing oracles with versions of its own.

```
Game $\mathsf{G_{OW}}$ :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $y \xleftarrow{\$} G$;
  $x \leftarrow \mathcal{I}(pk, y)$

Adversary $\mathcal{I}(pk, y)$ :
  $\hat{\boldsymbol{pk}} \leftarrow pk$;
  $\hat{\boldsymbol{y}} \leftarrow y$;
  $\boldsymbol{j} \xleftarrow{\$} \{0, \ldots, q\}$;
  $\boldsymbol{P}, \boldsymbol{L} \leftarrow$ nil;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  return $\sigma$

Oracle $H(m)$ :
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    if $|\boldsymbol{L}| = \boldsymbol{j}$ then $h \leftarrow \hat{\boldsymbol{y}}$
    else $r \xleftarrow{\$} G$; $h \leftarrow f(\hat{\boldsymbol{pk}}, r)$
    $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
  return $\boldsymbol{L}[m]$

Oracle $\mathsf{Sign}(m)$ :
  $h \leftarrow H(m)$;
  return $\boldsymbol{P}[m]$
```

Fig. 5.1. The inverter $\mathcal{I}$ in the context of the one-wayness game for the family of trapdoor permutations $(\mathcal{KG}_f, f, f^{-1})$.

In order to stand a chance of forging a signature for a fresh message m, the adversary $\mathcal{A}$ must ask for the hash value of m by querying oracle H. Otherwise, the hash of m would be completely random an independent of the adversary's output, and thus the purported signature σ would only be valid with a negligible probability. The inverter $\mathcal{I}$ tries to guess the query where the hash value of m is asked for the first time. Let $q \overset{\text{def}}{=} q_{\mathsf{S}} + q_{\mathsf{H}}$. The inverter first randomly chooses an index $\boldsymbol{j}$ in $\{0, \ldots, q\}$ and then runs the forger intercepting its oracle queries. The inverter answers to the $\boldsymbol{j}$-th hash query (we index the queries from 0) with its own challenge y, and to the remaining hash queries with a random element in the range of f with a known preimage; it stores this preimage in a list $\boldsymbol{P}$. When the adversary makes a signing query, the inverter first makes the corresponding hash query itself and then obtains the preimage of the hash value under f from the list $\boldsymbol{P}$. The simulation is perfect provided the forger never asks the signature of the message corresponding to the $\boldsymbol{j}$-th hash query, because in this case its preimage will not be in the list $\boldsymbol{P}$. A sufficient condition for the simulation to be correct is that the guess $\boldsymbol{j}$ be correct (i.e. $m = \boldsymbol{M}[\boldsymbol{j}]$), because m cannot appear in a signing query (it must be fresh).

We can readily analyze the extra time the inverter spends in simulating the environment for $\mathcal{A}$ in $\mathsf{G_{OW}}$. The only significant overhead is in the simulation of the hash oracle H. For all but one hash query, the simulated oracle computes the image under f of some element. The time bound (5.2) follows because the adversary makes (either directly or indirectly, through the signing oracle) at most $q_{\mathsf{H}} + q_{\mathsf{S}}$ hash queries.

To prove that the probability bound in equation (5.1) holds, we will exhibit a sequence of games relating the probability

$$\epsilon' = \Pr\left[\mathsf{G_{OW}} : x = f^{-1}(sk, y)\right]$$

of $\mathcal{I}$ successfully inverting f on a random challenge y, to the probability

$$\epsilon = \Pr\left[\mathsf{G_{EF}} : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S}\right]$$

of adversary $\mathcal{A}$ forging a signature for a fresh message. For each game, the main experiment is shown alongside the code of procedures in the environment; code pieces that change with respect to the previous game in the sequence appear on a gray background.

We start from the game $\mathsf{G_{EF}}$ that encodes the existential forgery attack. In this initial game, the hash oracle H is implemented as a random oracle whereas the signing oracle is implemented as specified by the scheme. In order to be able to encode the freshness condition on the message whose signature is forged, the signing oracle is instrumented to record the queries it gets.

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

```
Oracle $H(m)$ :
if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
  $h \xleftarrow{\$} G$; $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$
return $\boldsymbol{L}[m]$

Oracle Sign$(m)$ :
$\boldsymbol{S} \leftarrow m :: \boldsymbol{S}$; $h \leftarrow H(m)$;
return $f^{-1}(\boldsymbol{sk}, h)$
```

In game $\mathsf{G_1}$ we instrument the hash oracle to keep track of the indices of queries. We use for this purpose a list $\boldsymbol{M}$ where we store the messages queried to the hash oracle so far. Note that this is not really necessary because the value of $\boldsymbol{M}$

can be recovered at any given moment from the value of $\boldsymbol{L}$. We only make this instrumentation for convenience and to make the proof cleaner. We also introduce the guess $\boldsymbol{j}$ that will be used later by the inverter. We sample $\boldsymbol{j}$ uniformly at the end of the game so that its independence from the output of the adversary is evident.

```
Game G_1 :
  $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{M} \leftarrow \mathsf{nil}$;
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow \mathsf{nil}$;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  $h \leftarrow H(m)$;
  $\boldsymbol{j} \xleftarrow{\$} \{0, \ldots, q\}$
```

```
Oracle H(m) :
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    $h \xleftarrow{\$} G; \boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
    $\boldsymbol{M} \leftarrow m :: \boldsymbol{M}$
  return $\boldsymbol{L}[m]$
Oracle Sign(m) :
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  return $f^{-1}(\boldsymbol{sk}, h)$
```

Consider the predicate

$$\Phi_1 \stackrel{\text{def}}{=} |\boldsymbol{L}| = |\boldsymbol{M}| \wedge (\forall m \in \mathsf{dom}(\boldsymbol{L}).\ \exists i < |\boldsymbol{M}|.\ m = \boldsymbol{M}[i])$$

We prove that

$$\vdash \mathsf{G_{EF}} \sim \mathsf{G_1} : \mathsf{true} \Rightarrow =_{\{\boldsymbol{L},\boldsymbol{S},pk,m,\sigma,h\}} \wedge \Phi_1\langle 2\rangle \tag{5.3}$$

Using the tactics `wp` and `eqobs_in` we construct the relational procedure information for the oracles in the environment of both games. We then extend it automatically to the adversary $\mathcal{A}$ to obtain the information ι that we need to prove the equivalence. The script we use to prove (5.3) is just

```
deadcode ι; eqobs_ctxt ι; wp; ...
```

The tactic `deadcode` removes the random assignment to $\boldsymbol{j}$ in $\mathsf{G_1}$, while `eqobs_ctxt` removes the common prefix and suffix in both games except for the instruction $\boldsymbol{L} \leftarrow \mathsf{nil}$ because it affects the invariant $\Phi_1\langle 2\rangle$. The intermediate goal after applying these first two tactics is

$$\vdash \boldsymbol{L} \leftarrow \mathsf{nil} \sim \boldsymbol{M}, \boldsymbol{L} \leftarrow \mathsf{nil} : =_{\{pk,\boldsymbol{sk}\}} \Rightarrow =_{\{\boldsymbol{L}\}} \wedge \Phi_1\langle 2\rangle$$

The tactic `wp` is then used to compute the weakest relational pre-condition of $=_{\{\boldsymbol{L}\}} \wedge \Phi_1\langle 2\rangle$ with respect to the two resulting program fragments; the ellipsis stands for a straightforward script to prove that this weakest pre-condition holds. Games $\mathsf{G_{EF}}$ and $\mathsf{G_1}$ are thus equivalent on h, pk, σ, m, and $\boldsymbol{S}$, which implies

$$\Pr\left[\mathsf{G_{EF}} : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S}\right] = \Pr\left[\mathsf{G_1} : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S}\right] \tag{5.4}$$

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

$$\Psi \land \Phi_1 \land m \in \mathsf{dom}(\boldsymbol{L}) \implies \exists i \leq q.\ m = \boldsymbol{M}[i]$$

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

$$\frac{\Pr[\mathsf{G}_1 : h = f(pk, \sigma) \land m \notin \boldsymbol{S}]}{q+1} \leq \Pr[\mathsf{G}_1 : h = f(pk, \sigma) \land m \notin \boldsymbol{S} \land m{=}\boldsymbol{M}[\boldsymbol{j}]] \quad (5.5)$$

We apply now a semantics preserving transformation. Game G_2 eagerly samples the value $\hat{\boldsymbol{y}}$ that is given as answer to the $\boldsymbol{j}$-th hash query, and that will later become the challenge to the inverter.

```
Game $\mathsf{G}_2$ :
  $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $\hat{\boldsymbol{y}} \stackrel{\$}{\leftarrow} G$;
  $\boldsymbol{j} \stackrel{\$}{\leftarrow} \{0, \ldots, q\}$;
  $\boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S} \leftarrow \mathsf{nil}$;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  $h \leftarrow H(m)$
```

```
Oracle $H(m)$ :
  if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
    if $|\boldsymbol{L}| = \boldsymbol{j}$ then $h \leftarrow \hat{\boldsymbol{y}}$
    else $h \stackrel{\$}{\leftarrow} G$;
    $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
    $\boldsymbol{M} \leftarrow m :: \boldsymbol{M}$
  return $\boldsymbol{L}[m]$
Oracle $\mathsf{Sign}(m)$ :
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
  return $f^{-1}(\boldsymbol{sk}, h)$
```

We obtain

$$\vdash \mathsf{G}_1 \simeq^{\emptyset}_{\{\boldsymbol{j}, \boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S}, pk, m, \sigma, h\}} \mathsf{G}_2$$

Therefore,

$$\begin{aligned} &\Pr[\mathsf{G}_1 : h = f(pk, \sigma) \land m \notin \boldsymbol{S} \land m = \boldsymbol{M}[\boldsymbol{j}]] \\ &\qquad = \Pr[\mathsf{G}_2 : h = f(pk, \sigma) \land m \notin \boldsymbol{S} \land m = \boldsymbol{M}[\boldsymbol{j}]] \end{aligned} \quad (5.6)$$

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

$$\vdash h \stackrel{\$}{\leftarrow} G \simeq^{\emptyset}_{\{h\}} r \stackrel{\$}{\leftarrow} G; h \leftarrow f(\boldsymbol{pk}, r)$$

In preparation for the next transformation, we also introduce a flag **bad** to signal whether the simulation failed, i.e. whether the adversary asked for the signature of $\boldsymbol{M}[\boldsymbol{j}]$.

```
Game $\mathsf{G}_3$ :
 $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
 $\hat{\boldsymbol{y}} \xleftarrow{\$} G$;
 $\boldsymbol{j} \xleftarrow{\$} \{0, \ldots, q\}$;
 $\boldsymbol{P} \leftarrow$ nil;
 $\boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
 $(m, \sigma) \leftarrow \mathcal{A}(\boldsymbol{pk})$;
 $h \leftarrow H(m)$

Oracle $H(m)$ :
 if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
   if $|\boldsymbol{L}| = \boldsymbol{j}$ then $h \leftarrow \hat{\boldsymbol{y}}$
   else $r \xleftarrow{\$} G$; $h \leftarrow f(\boldsymbol{pk}, r)$
   $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
   $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
   $\boldsymbol{M} \leftarrow m :: \boldsymbol{M}$
 return $\boldsymbol{L}[m]$

Oracle Sign$(m)$ :
 $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
 if $m = \boldsymbol{M}[\boldsymbol{j}]$ then
   bad $\leftarrow$ true;
   return $f^{-1}(\boldsymbol{sk}, h)$
 else return $f^{-1}(\boldsymbol{sk}, h)$
```

We prove the equivalence

$$\vdash \mathsf{G}_2 \sim \mathsf{G}_3 : \mathsf{true} \Rightarrow =_{\{\boldsymbol{j},\boldsymbol{M},\boldsymbol{L},\boldsymbol{S},pk,m,\sigma,h\}} \wedge (\boldsymbol{M}[\boldsymbol{j}] \notin \boldsymbol{S} \implies \neg\mathbf{bad})\langle 2\rangle$$

Hence

$$\begin{aligned}\Pr[\mathsf{G}_2 : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}]] \\ = \Pr[\mathsf{G}_3 : h = f(\boldsymbol{pk}, \sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}]]\end{aligned} \tag{5.7}$$

In game G_4 we modify the signing oracle so that if the signature of the message $\boldsymbol{M}[\boldsymbol{j}]$ is ever asked, instead of actually computing the preimage of its digest using $\boldsymbol{sk}$, the signing oracle simply returns the corresponding $\boldsymbol{P}$-entry. This entry will be undefined, but this poses no problem because as long as the guess $\boldsymbol{j}$ is correct this piece of code is unreachable.

```
Game $\mathsf{G}_4$ :
 $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
 $\hat{\boldsymbol{y}} \xleftarrow{\$} G$;
 $\boldsymbol{j} \xleftarrow{\$} \{0, \ldots, q\}$;
 $\boldsymbol{P} \leftarrow$ nil;
 $\boldsymbol{M}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
 $(m, \sigma) \leftarrow \mathcal{A}(\boldsymbol{pk})$;
 $h \leftarrow H(m)$

Oracle $H(m)$ :
 if $m \notin \mathsf{dom}(\boldsymbol{L})$ then
   if $|\boldsymbol{L}| = \boldsymbol{j}$ then $h \leftarrow \hat{\boldsymbol{y}}$
   else $r \xleftarrow{\$} G$; $h \leftarrow f(\boldsymbol{pk}, r)$
   $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
   $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
   $\boldsymbol{M} \leftarrow m :: \boldsymbol{M}$
 return $\boldsymbol{L}[m]$

Oracle Sign$(m)$ :
 $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}; h \leftarrow H(m)$;
 if $m = \boldsymbol{M}[\boldsymbol{j}]$ then
   bad $\leftarrow$ true;
   return $\boldsymbol{P}[m]$
 else return $f^{-1}(\boldsymbol{sk}, h)$
```

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

$$\begin{aligned}&\Pr[\mathsf{G}_3 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}] \wedge \neg\mathbf{bad}]\\ &\qquad = \Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}] \wedge \neg\mathbf{bad}]\end{aligned}$$

Since $\boldsymbol{M}[\boldsymbol{j}] \notin \boldsymbol{S} \implies \neg\mathbf{bad}$ is a post-condition of G_3, it does not change anything if we remove the last term $\neg\mathbf{bad}$ from the event under consideration,

$$\begin{aligned}&\Pr[\mathsf{G}_3 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}]]\\ &\qquad = \Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}] \wedge \neg\mathbf{bad}]\\ &\qquad \le \Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}]]\end{aligned} \tag{5.8}$$

When answering a signing query for a message $m \neq \boldsymbol{M}[\boldsymbol{j}]$, we may obtain the preimage of its hash value from $\boldsymbol{P}$ rather than using f^{-1}, as in game G_5.

```
Game G5 :                              Oracle H(m) :
 (pk, sk) ← KG_f();                     if m ∉ dom(L) then
 ŷ ←$ G;                                  if |L| = j then h ← ŷ
 j ←$ {0, ..., q};                        else r ←$ G; h ← f(pk, r)
 P, L ← nil;                              P ← (m, r) :: P;
 (m, σ) ← A(pk);                          L ← (m, h) :: L
 y ← ŷ;                                 return L[m]
 x ← σ                                 Oracle Sign(m) :
                                        h ← H(m);
                                        return P[m]
```

Define Φ_4 as

$$\begin{aligned}&\forall m \in \mathsf{dom}(\boldsymbol{L}).\ m \neq \boldsymbol{M}[\boldsymbol{j}] \implies \boldsymbol{P}[m] = f^{-1}(\boldsymbol{sk}, \boldsymbol{L}[m]) \wedge\\ &(\boldsymbol{j} \le |\boldsymbol{M}| \implies \boldsymbol{L}[\boldsymbol{M}[\boldsymbol{j}]] = \hat{\boldsymbol{y}})\end{aligned}$$

We use Φ_4 to prove that the signing oracles in games G_4 and G_5 are equivalent. To this end we show that

$$\vdash \mathsf{G}_4 \sim \mathsf{G}_5 : \mathsf{true} \Rightarrow =_{\{\hat{\boldsymbol{y}},\boldsymbol{j},\boldsymbol{L},m,\sigma,h\}} \wedge \Phi_4\langle 1\rangle$$

If $m = \boldsymbol{M}[\boldsymbol{j}]$, then certainly $\boldsymbol{j} \le |\boldsymbol{M}|$ when the game finishes. In this case, post-condition Φ_4 implies that $h = \boldsymbol{L}[\boldsymbol{M}[\boldsymbol{j}]] = \hat{\boldsymbol{y}}$, which in turn gives

$$\Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk},\sigma) \wedge m \notin \boldsymbol{S} \wedge m = \boldsymbol{M}[\boldsymbol{j}]] \le \Pr\left[\mathsf{G}_5 : f^{-1}(sk, y) = x\right] \tag{5.9}$$

We finally prove that $\vdash \mathsf{G}_5 \simeq^{\emptyset}_{\{sk,x,y\}} \mathsf{G}_{\mathsf{OW}}$. We need to inline the call to the inverter $\mathcal{I}$ in G_{OW} and use y instead of $\hat{\boldsymbol{y}}$ in G_5. The proof script is straightforward,

```
alloc_l $\hat{\boldsymbol{y}}$ $y$; sinline_r $\iota$ $\mathcal{I}$.
eqobs_tl $\iota$; deadcode; eqobs_in.
```

Using the above equivalence we derive

$$\Pr\left[\mathsf{G}_5 : f^{-1}(sk, y) = x\right] = \Pr\left[\mathsf{G}_{\mathsf{OW}} : f^{-1}(sk, y) = x\right] \tag{5.10}$$

Putting all the above results together, we conclude

$$\frac{\Pr\left[\mathsf{G}_{\mathsf{EF}} : h = f(pk, \sigma) \land m \notin \boldsymbol{S}\right]}{q+1} \leq \Pr\left[\mathsf{G}_{\mathsf{OW}} : f^{-1}(sk, y) = x\right] \tag{5.11}$$

□

A complete proof of asymptotic security follows trivially. Since f is a one-way permutation family, $\Pr\left[\mathsf{G}_{\mathsf{OW}} : f^{-1}(sk, y) = x\right]$ is negligible in the security parameter provided $\mathcal{I}$ runs in PPT. This is indeed the case, and is proved automatically in CertiCrypt. The number of queries q made by the forger to the hash and signing oracles must necessarily be polynomial on the security parameter η. Since the product of a negligible function and a polynomial is still a negligible function, it follows from (5.11) that the probability of a successful existential forgery is negligible.

5.2 Improved Bound

A tighter security bound for FDH appears in [Coron 2000]; this bound is independent of the number of hash queries. This is of much practical significance since the number of hash values a real-world forger can compute is only limited by the time and computational resources it invests, whereas the number of signatures it gets could be limited by the owner of the private key. Once the owner of a key has used it to sign a certain quantity of messages, he could simply discard that key and generate a new one.

Theorem 5.4 (Improved bound). *Assume the underlying trapdoor permutation $(\mathcal{KG}_f, f, f^{-1})$ is homomorphic with respect to the group operation in its domain, i.e. for every (pk, sk) that might be output by $\mathcal{KG}_f$, and every x, y, $f(pk, x \times y) = f(pk, x) \times f(pk, y)$. Let $\mathcal{A}$ be an adversary against the existential unforgeability of FDH that makes at most q_{H} and q_{S} queries to the hash and signing oracles respectively. Suppose $\mathcal{A}$ succeeds in forging a signature for a fresh message within time t with probability ϵ during experiment G_{EF}. Then, there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f with probability ϵ' within time t' during experiment G_{OW}, where*

$$\epsilon' \geq \frac{1}{q_{\mathsf{S}}+1}\left(1 - \frac{1}{q_{\mathsf{S}}+1}\right)^{q_{\mathsf{S}}} \epsilon \tag{5.12}$$

$$t' \leq t + (q_{\mathsf{H}} + q_{\mathsf{S}})\ O(t_f) \tag{5.13}$$

These bounds hold for the inverter shown in Figure 5.2. The inverter first samples $q+1$ bits at random, choosing true with probability p and false with probability $(1-p)$, and stores them in a list $\boldsymbol{T}$. It answers to the $\boldsymbol{i}$-th hash query as follows: it picks uniformly a value r from the domain of f and stores it in a list $\boldsymbol{P}$, then replies according to the $\boldsymbol{i}$-th entry in $\boldsymbol{T}$: if it is true, answers with $y \times f(\boldsymbol{pk},r)$ where y is its challenge, if it is false answers with simply $f(\boldsymbol{pk},r)$. In both cases the answers are indistinguishable from those of a random function. When the adversary asks for the signature of a message m, the inverter makes the corresponding hash query itself and then answers with the m entry in the list $\boldsymbol{P}$. The simulation is correct provided the entries in $\boldsymbol{T}$ corresponding to messages appearing in signing queries are false, because in this case the corresponding entries in $\boldsymbol{P}$ coincide with the preimage of their hash value. The aim of the inverter is to inject its challenge in as many hash queries as possible, while at the same time maximizing the probability of the simulation being correct. The parameter p is left unspecified through the proof and will be chosen later to find the best compromise between these two competing goals.

```
Game G_OW :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $y \xleftarrow{\$} G$;
  $x \leftarrow \mathcal{I}(pk, y)$

Adversary $\mathcal{I}(pk, y)$ :
  $\boldsymbol{pk} \leftarrow pk$;
  $\hat{\boldsymbol{y}} \leftarrow y$;
  $\boldsymbol{i} \leftarrow 0$;
  $\boldsymbol{T}, \boldsymbol{P}, \boldsymbol{L} \leftarrow$ nil;
  while $|\boldsymbol{T}| \leq q$ do
    $b \xleftarrow{\$}$ true $\oplus_p$ false;
    $\boldsymbol{T} \leftarrow b :: \boldsymbol{T}$
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  $h \leftarrow H(m)$;
  return $\sigma \times \boldsymbol{P}[m]^{-1}$

Oracle $H(m)$ :
  if $m \notin$ dom($\boldsymbol{L}$) then
    $r \xleftarrow{\$} G$;
    if $\boldsymbol{T}[\boldsymbol{i}]$ = true then
      $h \leftarrow \hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, r)$
    else $h \leftarrow f(\boldsymbol{pk}, r)$
    $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
    $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}[m]$

Oracle Sign($m$) :
  $h \leftarrow H(m)$;
  return $\boldsymbol{P}[m]$
```

Fig. 5.2. The inverter $\mathcal{I}$ in the context of the one-wayness game for the family of trapdoor permutations $(\mathcal{KG}_f, f, f^{-1})$. We use (true $\oplus_p$ false) to denote a Bernoulli distribution with success probability p, i.e. the discrete distribution that takes value true with probability p and false with probability $(1-p)$.

The success of the inverter is not guaranteed by the sole success of the forger. It depends on two additional conditions: that the simulation is consistent, so that the forger behaves as expected, and that the forgery can be used to compute the preimage of the challenge y.

Let us analyze first the probability of the simulation being consistent. The forger $\mathcal{A}$ must not be able to distinguish the simulation from the experiment in G_{EF}. The forger's view, and in particular the distribution of the answers it gets from the hash and signing oracles must be the same as in the original experiment. The responses of the simulated hash oracle are uniformly distributed, as in the original oracle. However, the inverter might not be able to answer with consistent signatures to every signing query made by the forger. The reason is that the inverter knows the preimage of a message hash only if the hash was computed as $f(\boldsymbol{pk}, r)$ for a random r, which occurs only if the corresponding entry in the list $\boldsymbol{T}$ has been chosen as false. Note that we did not have this problem in the previous proof, because the mere fact that the guess of the inverter is correct implied that all signing queries could be consistently answered.

The forgery (m, σ) that $\mathcal{A}$ outputs can be used by the inverter to compute the preimage of its challenge only if the hash of message m has been computed as $\hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, r)$ for a random r, which occurs only if the $\boldsymbol{T}$-entry corresponding to m has been chosen beforehand as true. Indeed, if that is the case and the forged signature is correct, we have

$$f^{-1}(sk, y) = f^{-1}(sk, H(m)) \times r^{-1} = \sigma \times r^{-1}$$

We begin our sequence of games by bounding the probability of the above two conditions. To this end, in game G_1 we instrument the hash oracle to keep track of the indices of queries. We add the initialization of list $\boldsymbol{T}$ at the end of the game, so that it becomes part of the probability space and its independence from the rest of the game is obvious.

```
Game G_1 :
  $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;                       ⎤
  $\boldsymbol{i} \leftarrow 0$; $\boldsymbol{I} \leftarrow$ nil;            ⎥
  $\boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;                           ⎥ $\mathsf{G}_1'$
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;                                   ⎥
  $h \leftarrow H(m)$;                                                        ⎦
  $\boldsymbol{T} \leftarrow$ nil;                                            ⎤
  while $|\boldsymbol{T}| \leq q$ do                                          ⎥ Init$_{\boldsymbol{T}}$
    $b \xleftarrow{\$}$ true $\oplus_p$ false;                                ⎥
    $\boldsymbol{T} \leftarrow b :: \boldsymbol{T}$                           ⎦

Oracle $H(m)$ :
  if $m \notin$ dom$(\boldsymbol{L})$ then
    $h \xleftarrow{\$} G$; $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
    $\boldsymbol{I} \leftarrow (m, \boldsymbol{i}) :: \boldsymbol{I}$;
    $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
  return $\boldsymbol{L}[m]$

Oracle Sign$(m)$ :
  $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}$; $h \leftarrow H(m)$;
  return $f^{-1}(\boldsymbol{sk}, h)$
```

We require that the forged signature σ verifies, but in addition that the entry in $\boldsymbol{T}$ corresponding to m be true and that the entries corresponding to signing queries be false,

$$\text{success} \overset{\text{def}}{=} \boldsymbol{T}[\boldsymbol{I}[m]] \ \wedge\ \forall m' \in \boldsymbol{S}.\ \neg \boldsymbol{T}[\boldsymbol{I}[m']] \tag{5.14}$$

Observe that this condition alone implies the freshness of message m, we do not need to state it explicitly. We would like to compute now a lower bound for $\Pr[\mathsf{G}_1 : h = f(pk, \sigma)\ \wedge\ \text{success}]$ in terms of $\Pr[\mathsf{G}_1 : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S}]$. This would be a relatively easy task if the event success were independent of $h = f(pk, \sigma)$, but this is not the case. However, we have for any initial memory μ,

$$
\begin{aligned}
&\Pr[\mathsf{G}_1, \mu : h = f(pk, \sigma) \wedge \mathsf{success}] \\
&\quad = [\![\mathsf{G}_1]\!]\ \mu\ \mathbb{1}_{(h=f(pk,\sigma)\wedge \mathsf{success})} \\
&\quad = [\![\mathsf{G}_1']\!]\ \mu\ (\lambda\mu'.\ [\![\mathsf{Init}_{\boldsymbol{T}}]\!]\ \mu'\ (\lambda\mu''.\ \mathbb{1}_{h=f(pk,\sigma)}\ \mu'' \times \mathbb{1}_{\mathsf{success}}\ \mu'')) \\
&\quad = [\![\mathsf{G}_1']\!]\ \mu\ (\lambda\mu'.\ \mathbb{1}_{(h=f(pk,\sigma)\wedge m\notin \boldsymbol{S})}\ \mu' \times \Pr[\mathsf{Init}_{\boldsymbol{T}}, \mu' : \mathsf{success}]))
\end{aligned}
\tag{5.15}
$$

Thus, computing the original probability is equivalent to measuring a function that can be expressed as a product of two factors, one of which is the probability of event success after initializing $\boldsymbol{T}$ in the intermediate memory μ'. The advantage of doing this is that the second factor in the product is upper bounded by a constant under some conditions on μ' that we can prove to hold. Namely, we can show that the following is a post-condition of the piece of code G_1'

$$
|\boldsymbol{L}| \leq q + 1\ \wedge\ |\boldsymbol{S}| \leq q_{\mathsf{S}}\ \wedge\ \mathsf{ran}(\boldsymbol{I}) = [|\boldsymbol{L}| - 1 \,..\, 0] \tag{5.16}
$$

Furthermore, we can assume that m is not in $\boldsymbol{S}$ in memory μ' because otherwise the measured function would be null. Under this conditions, we will show that

$$
p(1-p)^{q_{\mathsf{S}}} \leq \Pr[\mathsf{Init}_{\boldsymbol{T}}, \mu' : \mathsf{success}]
$$

Consider the loop that initializes $\boldsymbol{T}$,

$$
c \overset{\text{def}}{=} \mathsf{while}\ |\boldsymbol{T}| \leq q\ \mathsf{do}\ (b \xleftarrow{\$} \mathsf{true} \oplus_p \mathsf{false};\ \boldsymbol{T} \leftarrow b :: \boldsymbol{T})
$$

and define

$$
\begin{aligned}
F_\mu(i,n) &\overset{\text{def}}{=} \mathsf{if}\ n < i\ \mathsf{then}\ [\![\boldsymbol{T}[i-n]]\!]\ \mu\ \mathsf{else}\ p \\
G_\mu(l,n) &\overset{\text{def}}{=} \prod_{i \in l} \mathsf{if}\ n < i\ \mathsf{then}\ [\![\neg\boldsymbol{T}[i-n]]\!]\ \mu\ \mathsf{else}\ 1-p
\end{aligned}
$$

This may seem abstruse at first sight, but intuitively, $F_\mu(i, q-k)$ equals the probability of $\boldsymbol{T}[i]$ being true after executing $\mathsf{Init}_{\boldsymbol{T}}$ in a memory μ where $|\boldsymbol{T}| = k$, while $G_\mu(i, q-k)$ is a lower bound on the probability of $\forall i \in l.\ \neg\boldsymbol{T}[i]$. We prove the following invariant about the while loop c: for every index i and list of indices l such that $i \notin l$,

$$
F_\mu(i, q - |[\![\boldsymbol{T}]\!]\ \mu|)\ G_\mu(l, q - |[\![\boldsymbol{T}]\!]\ \mu|) \leq \Pr[c, \mu : \boldsymbol{T}[i] \wedge \forall i \in l.\ \neg\boldsymbol{T}[i]] \tag{5.17}
$$

For any memory μ satisfying (5.16) and where $m \notin \boldsymbol{S}$, if we take $i = \boldsymbol{I}[m]$ and $l = \boldsymbol{I}[\boldsymbol{S}]$, we obtain

$$
\begin{aligned}
p\ (1-p)^{q_{\mathsf{S}}} &\leq F_\mu(i,q)\ G_\mu(l,q) \\
&\leq \Pr[\mathsf{Init}_{\boldsymbol{T}}, \mu : \boldsymbol{T}[i] \wedge \forall i \in l.\ \neg\boldsymbol{T}[i]] \\
&= \Pr[\mathsf{Init}_{\boldsymbol{T}}, \mu : \mathsf{success}]
\end{aligned}
$$

Since any intermediate memory μ' in (5.15) satisfies (5.16), it follows from the above inequality that

$$\begin{aligned} p\ (1-p)^{q_S}\ \Pr\left[\mathsf{G}_{\mathsf{EF}} : h = f(pk,\sigma) \wedge m \notin \boldsymbol{S}\right] & \\ &= p\ (1-p)^{q_S}\ \Pr\left[\mathsf{G}_1\ : h = f(pk,\sigma) \wedge m \notin \boldsymbol{S}\right] \\ &\leq \Pr\left[\mathsf{G}_1 : h = f(pk,\sigma) \wedge \mathsf{success}\right] \end{aligned} \tag{5.18}$$

In the next game, G_2, we modify the hash oracle to answer to the $\boldsymbol{i}$-th query according to the corresponding entry in list $\boldsymbol{T}$: the oracle samples a random element r in the group, stores it in list $\boldsymbol{P}$, and responds $\hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, r)$ if $\boldsymbol{T}[i]$ is true, and simply $f(\boldsymbol{pk}, r)$ otherwise. In this latter case, the inverse image of the hash value can be recovered from $\boldsymbol{P}$.

```
Game G_2 :                              Oracle H(m) :
  $\hat{y} \xleftarrow{\$} G$;                           if $m \notin \mathsf{dom}(L)$ then
  $T \leftarrow$ nil;                              $r \xleftarrow{\$} G$;
  while $|T| \leq q$ do                        if $T[i]$ then $h \leftarrow \hat{y} \times f(pk, r)$
    $b \xleftarrow{\$}$ true $\oplus_p$ false;                else $h \leftarrow f(pk, r)$
    $T \leftarrow b :: T$                              $P \leftarrow (m, r) :: P$;
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;                $L \leftarrow (m, h) :: L$;
  $i \leftarrow 0$; $I \leftarrow$ nil;                      $I \leftarrow (m, i) :: I$;
  $P, L, S \leftarrow$ nil;                        $i \leftarrow i + 1$
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;                  return $L[m]$
  $h \leftarrow H(m)$                            Oracle Sign(m) : ...
```

Observe that the transformation of G_1 into G_2 can be justified by locally reasoning on the code of the hash oracle, without needing to apply the lazy sampling technique (as we had to do in the previous proof), thanks to the fact that f is a permutation and $f(\boldsymbol{pk}, r)$ acts as a one-time pad,

$$\vdash h \xleftarrow{\$} G \simeq^{\emptyset}_{\{h\}} r \xleftarrow{\$} G;\ h \leftarrow \hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, r)$$

We now modify the way the signing oracle answers to a query m when the corresponding entry $\boldsymbol{T}[\boldsymbol{I}[m]]$ is true: instead of answering with a proper signature, it answers with just $\boldsymbol{P}[m]$. By using the Fundamental Lemma we will see that this change does not modify the probability of the event that interests us.

```
Game [G_3] [G_4] :                      Oracle H(m) : ...
  $\hat{y} \xleftarrow{\$} G$;                           Oracle Sign(m) :
  $T \leftarrow$ nil;                              $S \leftarrow m :: S; h \leftarrow H(m)$;
  while $|T| \leq q$ do                        if $T[I[m]]$ then
    $b \xleftarrow{\$}$ true $\oplus_p$ false;                bad $\leftarrow$ true;
    $T \leftarrow b :: T$                              return [$f^{-1}(sk, h)$] [$P[m]$]
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;                else return $f^{-1}(sk, h)$
  $i \leftarrow 0$; $I \leftarrow$ nil;
  $P, L, S \leftarrow$ nil;
  $(m, \sigma) \leftarrow \mathcal{A}(pk)$;
  $h \leftarrow H(m)$
```

Game G_3 is semantically identical to G_2 (ignoring **bad**) and differs from G_4 only in a portion of code appearing after flag **bad** is set. Thus,

$$\Pr[\mathsf{G}_3 : h = f(pk, \sigma) \wedge \mathsf{success} \wedge \neg\mathbf{bad}] = \Pr[\mathsf{G}_4 : h = f(pk, \sigma) \wedge \mathsf{success} \wedge \neg\mathbf{bad}]$$

But since $\mathsf{success} \implies \neg\mathbf{bad}$ is a post-condition of G_3, it does not change anything if we remove the last term $\neg\mathbf{bad}$ from the event under consideration,

$$\begin{aligned}\Pr[\mathsf{G}_3 : h = f(\boldsymbol{pk}, \sigma) \wedge \mathsf{success}] &= \Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk}, \sigma) \wedge \mathsf{success} \wedge \neg\mathbf{bad}] \\ &\le \Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk}, \sigma) \wedge \mathsf{success}] \qquad (5.19)\end{aligned}$$

The signing oracle might as well respond with $\boldsymbol{P}[m]$ to every query, regardless of the value of $\boldsymbol{T}[\boldsymbol{I}[m]]$,

```
Game G_5 :
 $\hat{\boldsymbol{y}} \xleftarrow{\$} G$;
 $\boldsymbol{T} \leftarrow$ nil;
 while $|\boldsymbol{T}| \le q$ do
   $b \xleftarrow{\$}$ true $\oplus_p$ false;
   $\boldsymbol{T} \leftarrow b :: \boldsymbol{T}$
 $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
 $\boldsymbol{i} \leftarrow 0$; $\boldsymbol{I} \leftarrow$ nil;
 $\boldsymbol{P}, \boldsymbol{L}, \boldsymbol{S} \leftarrow$ nil;
 $(m, \sigma) \leftarrow \mathcal{A}(\boldsymbol{pk})$;
 $h \leftarrow H(m)$;
 $y \leftarrow \hat{\boldsymbol{y}}$;
 $x \leftarrow \sigma \times \boldsymbol{P}[m]^{-1}$
```

```
Oracle $H(m)$ : ...
 if $m \notin$ dom($\boldsymbol{L}$) then
   $r \xleftarrow{\$} G$;
   if $\boldsymbol{T}[\boldsymbol{i}]$ then $h \leftarrow \hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, r)$
     else $h \leftarrow f(\boldsymbol{pk}, r)$
   $\boldsymbol{P} \leftarrow (m, r) :: \boldsymbol{P}$;
   $\boldsymbol{L} \leftarrow (m, h) :: \boldsymbol{L}$;
   $\boldsymbol{I} \leftarrow (m, \boldsymbol{i}) :: \boldsymbol{I}$;
   $\boldsymbol{i} \leftarrow \boldsymbol{i} + 1$
 return $\boldsymbol{L}[m]$
Oracle Sign($m$) :
 $\boldsymbol{S} \leftarrow m :: \boldsymbol{S}$; $h \leftarrow H(m)$;
 return $\boldsymbol{P}[m]$
```

We can guarantee that the response given is a proper signature when $\boldsymbol{T}[\boldsymbol{I}[m]]$ is false by proving the following post-condition of game G_5:

$$\begin{aligned}\forall (m, h) \in \boldsymbol{L}.\quad \boldsymbol{T}[\boldsymbol{I}[m]] &\implies h = \hat{\boldsymbol{y}} \times f(\boldsymbol{pk}, \boldsymbol{P}[m]) \wedge \\ \neg\boldsymbol{T}[\boldsymbol{I}[m]] &\implies h = f(\boldsymbol{pk}, \boldsymbol{P}[m])\end{aligned}$$

This allows us to show that the signing oracles in G_4 and G_5 are equivalent and, using the homomorphic property of f, to show

$$\begin{aligned}\Pr[\mathsf{G}_4 : h = f(\boldsymbol{pk}, \sigma) \wedge \mathsf{success}] &= \Pr[\mathsf{G}_5 : h = f(\boldsymbol{pk}, \sigma) \wedge \mathsf{success}] \\ &\le \Pr[\mathsf{G}_5 : y \times f(\boldsymbol{pk}, \boldsymbol{P}[m]) = f(pk, \sigma)] \\ &= \Pr\left[\mathsf{G}_{\mathsf{OW}} : f^{-1}(sk, y) = x\right]\end{aligned}$$

Therefore, we can conclude

$$p\ (1-p)^{q_{\mathsf{S}}}\ \Pr[\mathsf{G}_{\mathsf{EF}} : h = f(pk, \sigma) \wedge m \notin \boldsymbol{S}] \le \Pr\left[\mathsf{G}_{\mathsf{OW}} : f^{-1}(sk, y) = x\right]$$

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

5.3 Practical Interpretation

If one accepts that it is reasonable to draw practical conclusions from a security proof in the random oracle model, then the results above may be used to choose the scheme parameters based on an estimate of the time needed to invert the underlying trapdoor permutation.

The best known method to invert the RSA function is to factor its modulus. The General Number Field Sieve (GNFS) [Lenstra and Jr. 1993] is currently the most efficient way of factoring large composite integers like RSA moduli; it has been used to factor several RSA moduli from the RSA Factoring Challenge, including the 512-bit RSA-155 number and the largest (non-special) integer factored with a general-purpose algorithm, the 768-bit RSA-768 number [Kleinjung et al. 2010]. On heuristic grounds, an odd composite number N can be factored using GNFS in time sub-exponential with respect to its size in bits. Concretely, let

$$L_N[\alpha,\beta] \overset{\text{def}}{=} \exp((\beta + o(1)) \log(N)^\alpha \log(\log(N))^{1-\alpha}) \tag{5.20}$$

where the constant term that accompanies β tends towards zero as N increases. A good implementation of GNFS takes about $L_N[1/3, 1.923]$ time to factor a number N. This heuristic cannot be used directly to estimate the number of operations required to factor a certain N [Lenstra and Verheul 2001]. However, experimental data suggests that it can be used for limited range extrapolation. If one knows, empirically, that factoring an RSA modulus N using GNFS takes time t, then factoring an RSA modulus $M > N$ will take approximately time

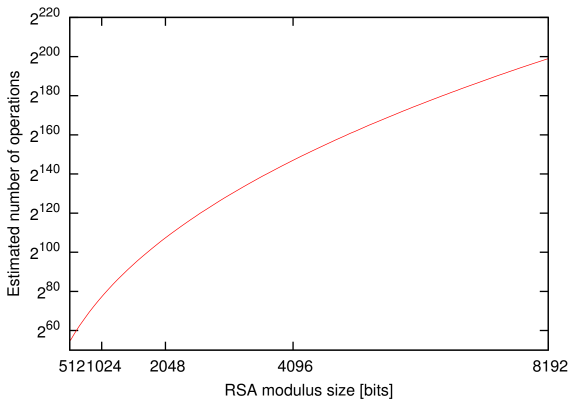
$$t \times \frac{L_M[1/3, 1.923]}{L_N[1/3, 1.923]} \tag{5.21}$$

(omitting the constant term $o(1)$). If $M \gg N$, however, the effect of the constant term can no longer be ignored and the extrapolation will overestimate the time needed to factor M.

The computational effort involved in the factorization of the 512-bit number RSA-155 has been estimated at around 8400 MIPS-years[1], or slightly less than 2^{58} operations [Cavallar et al. 2000]. In comparison, the computational effort involved in the factorization of the 768-bit number RSA-768 has been estimated at around 2^{67} operations [Kleinjung et al. 2010]. Extrapolating from this estimate using (5.21), we can make a rough prediction of the computational effort that would take to break larger RSA moduli (Figure 5.3).

GNFS could factor a 1024-bit number in around 2^{77} operations, and a 2048-bit number in around 2^{107} operations. Assume some safe bounds for q_H and q_S, $q_\mathsf{H} \leq 2^{60}$, $q_\mathsf{S} \leq 2^{20}$. To ensure that no forger within these bounds could forge a RSA-FDH signature within $t = 2^{80}$ operations, one should pick an RSA modulus such that factoring it takes at least $q_\mathsf{S}(t + (q_\mathsf{H} + q_\mathsf{S})O(t_\mathsf{RSA}))$ operations, otherwise

[1] One MIPS-year is the equivalent of a computation running during a full year at a sustained rate of one million instructions per second. Consumer desktop PCs at the time of this writing attain speeds of up to 80,000 MIPS.



Modulus size	Operations
512	2^{58}
768	2^{67}
1024	2^{77}
2048	2^{107}
4096	2^{147}
8192	2^{199}

Fig. 5.3. Estimates for the computational effort required to factor large RSA-moduli.

one can iterate the construction in Theorem 5.4 about q_{S} times to invert the RSA function in less time than using the GNFS algorithm. For a modulus of size k, $t_{\mathsf{RSA}} = O(k^2)$ when the public exponent is small. A 1024-bit modulus would not be enough, but a 2048-bit modulus would do. In contrast, if one were to choose the modulus according to the original security bound, even a 2048-bit modulus would not suffice.

The above guidelines for selecting key sizes by extrapolation should be taken with care. One should not forget that besides the fact that the analysis is based on heuristic ground, we are ignoring the $o(1)$ factor from (5.20). Shamir and Tromer [2003] said to this respect:

> "To determine what key sizes are appropriate for a given application, one needs concrete estimates for the cost of factoring integers of various sizes. Predicting these costs has proved notoriously difficult, for two reasons. First, the performance of modern factoring algorithms is not understood very well: their complexity analysis is often asymptotic and heuristic, and leaves large uncertainty factors. Second, even when the exact algorithmic complexity is known, it is hard to estimate the concrete cost of a suitable hypothetical large-scale computational effort using current technology; it's even harder to predict what this cost would be at the end of the key's planned lifetime, perhaps a decade or two into the future."

Even leaving aside the possibility of a breakthrough in number theory or the discovery of a new factorization method that drastically improves on GNFS, there are a myriad of tweaks and hardware optimizations that can be readily—or at least hypothetically—applied to cut down the cost of factoring an RSA modulus using GNFS. It is conjectured that it could be possible to factor 1024-bit integers, and hence to break 1024-bit RSA keys, in 1 year using a special hardware device that could be built at a cost of US$10M [Lenstra et al. 2003; Shamir and Tromer 2003].

5.4 Discussion and Related Work

An earlier example of a signature scheme that follows the same hash-then-decrypt pattern as FDH is the PKCS #1 scheme proposed by RSA Labs [RSA Data Security, Inc. 2002]. The latest version of PKCS #1 uses the following message encoding as a hash function

$$\mathsf{ENCODE}(m, len) \stackrel{\text{def}}{=} 00 \parallel 01 \parallel \mathrm{FF} \cdots \mathrm{FF} \parallel 00 \parallel h(m)$$

Enough FF bytes are inserted to reach the intended length len of the encoded message. The value $h(m)$ is (an encoding of) the digest of the message computed using one of MD2, MD5, SHA-1, SHA-256, SHA-384 or SHA-512. The problem with the above padding is that it does not map messages to the entire domain of the RSA function, but instead to a much smaller set of encoded messages. As a consequence, the scheme is not known to admit a security reduction to the standard RSA problem.

The generic Full-Domain Hash signature scheme based on a one-way trapdoor permutation was first described in the seminal work of Bellare and Rogaway [1993] to illustrate the applicability of the Random Oracle Model. In that work, the authors give in an appendix a sketch of a proof of its security against adaptive chosen-message attacks. A more detailed proof of the exact security of the RSA-based variant of FDH is given in [Bellare and Rogaway 1996,Theorem 3.1]; the bound corresponds to the one we showed in Section 5.1. The bound we proved in Section 5.2 is due to Coron [2000]. Unfortunately, it is not possible to further improve the security bound of FDH and prove that computing a forgery is as hard as inverting RSA. There is no tighter reduction than the one in Theorem 5.4 as showed by Coron [2002].

Bellare and Rogaway [1996] proposed the Probabilistic Signature Scheme (PSS) as a replacement for FDH; it has since then been incorporated into the PKCS #1 standard as an alternative signing method. PSS is roughly as efficient as FDH but admits a tight security reduction. The Probabilistic FDH scheme (PFDH) [Coron 2002] is a simple probabilistic variant of FDH that follows the same design principles as PSS and also admits a tight reduction, but at the cost of slightly longer signatures. As FDH, PFDH uses a hash function $H : \{0,1\}^* \to G$ and a trapdoor permutation f. Formally, PFDH is parametrized by the length k_0 of the random "salt" it uses, and is composed of the following triple of algorithms:

$$\begin{array}{lll}
\mathcal{KG} & \stackrel{\text{def}}{=} & (pk, sk) \leftarrow \mathcal{KG}_f;\ \mathsf{return}\ (pk, sk) \\
\mathsf{Sign}(sk, m) & \stackrel{\text{def}}{=} & r \stackrel{\$}{\leftarrow} \{0,1\}^{k_0};\ \mathsf{return}\ (f^{-1}(sk, H(m \parallel r)), r) \\
\mathsf{Verify}(pk, m, (\sigma, r)) & \stackrel{\text{def}}{=} & \mathsf{if}\ f(pk, \sigma) = H(m \parallel r)\ \mathsf{then\ return\ true\ else\ return\ false}
\end{array}$$

Observe that FDH is obtained as a special case of this scheme by setting $k_0 = 0$. We have the following result about the security of PFDH.

Theorem 5.5. *Let $\mathcal{A}$ be an adversary mounting a chosen-message existential forgery attack against PFDH that makes at most q_{H} queries to the hash oracle* *and at most $q_{\mathsf{S}} \le 2^{k_0}$ queries to the signing oracle. Suppose the adversary succeeds in forging a signature for a fresh message with probability ϵ within time t. Then, there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f with probability ϵ' within time t', where*

$$\begin{aligned} \epsilon' &\ge \left(1 - \frac{1}{q_{\mathsf{S}}}\right)^{q_{\mathsf{S}}} \epsilon \approx \exp(-1)\ \epsilon \\ t' &\le t + (q_{\mathsf{H}} + q_{\mathsf{S}})\ O(t_f) + q_{\mathsf{H}}\ q_{\mathsf{S}}\ O(k_0) \end{aligned}$$

This means that when $k_0 \ge \log_2(q_{\mathsf{S}})$, the security of PFDH is tightly related to the problem of inverting the underlying trapdoor permutation. If a shorter salt is used, PFDH remains provably secure, but a tight reduction is not possible.

Katz and Wang [2003] describe yet another signature scheme that achieves a tight reduction using a single bit as random salt. The random salt can be remove altogether by computing this bit in a deterministic (but secret) way. This scheme uses the same key generation procedure as FDH and PSS; the signature and verification algorithms are as follows,

```
Sign($sk, m$)             $\stackrel{\text{def}}{=}$  if $m \in \boldsymbol{S}$ then return $\boldsymbol{S}[m]$
                              else $b \stackrel{\$}{\leftarrow} \{0,1\}$; $\sigma \leftarrow f^{-1}(sk, H(b \parallel m))$; $\boldsymbol{S}[m] \leftarrow \sigma$;
                                  return $\sigma$
Verify($pk, m, \sigma$)   $\stackrel{\text{def}}{=}$  if $f(pk,\sigma) = H(0 \parallel m) \vee f(pk,\sigma) = H(1 \parallel m)$
                              then return true else return false
```

The signature algorithm is stateful: it will compute a signature for a message only once and return the same signature subsequently. To avoid maintaining state and remove the randomness, the bit b could be computed deterministically but in a secret way from message m. To avoid computing two hash values during verification, the bit b could be appended to the signature. We have the following result about the security of the above scheme,

Theorem 5.6. *Let $\mathcal{A}$ be an adversary mounting a chosen-message existential forgery attack against the Katz-Wang scheme that makes at most q_{H} queries to the hash oracle and at most q_{S} queries to the signing oracle. Suppose the adversary succeeds in forging a signature for a fresh message with probability ϵ within time t. Then, there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the range of f with probability ϵ' within time t', where*

$$\begin{aligned} \epsilon' &\ge \frac{1}{2}\ \epsilon \\ t' &\le t + (q_{\mathsf{H}} + q_{\mathsf{S}} + 1)\ O(t_f) \end{aligned}$$

Using just one *random* bit we have cut down the gap between the security of the scheme and the problem of inverting f by a factor of q_{S} !

The impossibility of finding a tight reduction from the security of a scheme to a hard problem does not necessarily mean that we should deem the scheme as insecure. It could be the case that the hard problem is not general enough, or maybe a slightly modified version of the scheme would admit a tight reduction. Or perhaps the protocol is indeed secure for practical matters, even though a tight reduction does not exist. The absence of a tight reduction for RSA-FDH does not mean that its security is only loosely related to the RSA problem. The result about the Katz-Wang scheme described above suggests exactly the contrary. In fact, it is easy to see that the security of RSA-FDH is equivalent to the following problem:

> You are given an RSA modulus N and its public exponent e, just as in the standard RSA problem. You are also given a set $\{y_1, \ldots, y_q\}$ of q values uniformly sampled from $\mathbb{Z}_N$. At any time you may chose a value y_i and get its e-th root modulo N, i.e. a solution x_i to $x_i^e \equiv y_i (\mathsf{mod}\ N)$; you may chose up to q_S of such values. Your goal is to compute the e-th root modulo N of one of the remaining y_i values.

It is hard to imagine how this problem could be any easier than solving the standard RSA problem.

Using CryptoVerif [Blanchet 2006], Blanchet and Pointcheval [2006] gave an alternative formal proof of the security of FDH against existential forgery under adaptive chosen-message attacks. This work has stirred considerable interest and shown the benefits of machine-checked verification. It also exposed one major weakness of CryptoVerif: it deviates from the style that is natural to cryptographers since it is difficult to recover a reductionist argument from the proof trace that the prover outputs, and even if one manages to do so, most likely the reduction will not be optimal. Indeed, only the original, suboptimal bound of Bellare and Rogaway has been proved in CryptoVerif.

Our machine-checked proofs follow quite closely the pen-and-paper game-based proofs of FDH (cf. [Pointcheval 2005]). There is however one important difference: in order to justify local transformations, machine-checked proofs must make invariants explicit and establish formally their validity. Proving that invariants hold constitutes a fair amount of work. More generally, machine-checked proofs must justify all reasoning, including reasoning about side conditions and about elementary mathematics (groups, probabilities) in terms of basic definitions. In contrast to game transformations, for which suitable tactics have been designed, this form of reasoning is not always amenable to automation, and thus accounts for a substantial amount of the effort and of the size of the proofs. Indeed, we estimate that about a third of the proof scripts are devoted to facts about probabilities. In spite of this, the size of machine-checked proofs remains reasonable: the formalizations of Bellare-Rogaway and Coron proofs are about 3,000 lines each. While the length of our proofs might look prohibitive in comparison to published proofs, we expect that machine-checked proofs will shrink substantially as CertiCrypt (and its underlying libraries) mature.

It must be noted that much of the proof lies outside of the trusted base: in order to trust the proofs of FDH, it is sufficient to trust our formalization of the scheme and of the security statement, the formalization of probabilistic programs provided by CertiCrypt, and the proof checker of Coq. In particular, trusting the proofs of FDH does not require trusting the sequence of games, nor the proofs of transitions, nor the proofs of invariants. In this respect, CertiCrypt provides the highest possible level of assurance for the security of a cryptographic scheme, and breaks the symmetry between the effort of writing and checking a cryptographic proof. Both usually require a lot of expertise in cryptography, a lot of time, and a good understanding of the proof; in contrast, it is rather immediate and simple for a third party to check a proof in CertiCrypt.

6
Ciphertext Indistinguishability of OAEP

When we first discussed in Chapter 1 the definition of semantic security for public-key encryption schemes we observed that in order to achieve this notion of security an encryption scheme must be necessarily probabilistic. A deterministic asymmetric encryption scheme cannot be semantically secure because an adversary could trivially decide whether a given ciphertext is the encryption of a plaintext by encrypting the plaintext and comparing the resulting ciphertext to the one it was given. It is possible, however, to use a deterministic encryption scheme as a building block to construct a semantically secure probabilistic encryption scheme. The following definition gives a general way to construct a probabilistic encryption scheme from any family of trapdoor permutations.

Definition 6.1 (Padding-based encryption scheme). *Let $\mu, \rho, k : \mathbb{N} \to \mathbb{N}$ be three functions such that $\forall \eta.\ \mu(\eta) + \rho(\eta) \leq k(\eta)$. Consider a family of mappings*

$$\begin{aligned}
\pi_\eta &: \{0,1\}^{\mu(\eta)+\rho(\eta)} \to \{0,1\}^{k(\eta)} \\
\hat{\pi}_\eta &: \{0,1\}^{k(\eta)} \to \{0,1\}^{\mu(\eta)} \cup \{\perp\}
\end{aligned}$$

such that π_η is injective and the following consistency condition is satisfied

$$\forall \eta.\ m \in \{0,1\}^{\mu(\eta)},\ r \in \{0,1\}^{\rho(\eta)}.\ \hat{\pi}_\eta(\pi_\eta(m \parallel r)) = m$$

Then, given a family of trapdoor permutations $(\mathcal{KG}_f, f, f^{-1})$ on $\{0,1\}^{k(\eta)}$, one can construct a probabilistic padding-based encryption scheme $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ as follows:

- *Given $\eta : \mathbb{N}$, the key generation algorithm $\mathcal{KG}(\eta)$ runs the key generation algorithm $\mathcal{KG}_f(\eta)$ of the family of trapdoor permutations and returns the pair of keys (pk, sk) that it obtains as result. We assume that the description of $(\pi_\eta, \hat{\pi}_\eta)$ is public, so there is no need to return it as part of the key generation process.*
- *Given a public key pk and a message $m \in \{0,1\}^{\mu(\eta)}$ the encryption algorithm $\mathcal{E}(pk, m)$ chooses a uniformly random bitstring $r \in \{0,1\}^{\rho(\eta)}$ and returns a ciphertext computed as $f(pk, \pi_\eta(m \parallel r)) \in \{0,1\}^{k(\eta)}$.*

- *Given a secret key sk and a ciphertext $c \in \{0,1\}^{k(\eta)}$, the decryption algorithm $\mathcal{D}(sk, c)$ returns $m = \hat{\pi}_\eta(f^{-1}(sk, c))$, which is either a message in $\{0,1\}^{\mu(\eta)}$ or the special symbol $\perp$, meaning that the ciphertext is invalid.*

Many public-key encryption schemes can be viewed as particular instances of the above construction, including OAEP [Bellare and Rogaway 1994] and its variants OAEP$^+$ [Shoup 2001], SAEP, SAEP$^+$ [Boneh 2001], and REACT [Okamoto and Pointcheval 2001b]. In this chapter we will discuss the security of OAEP.

Optimal Asymmetric Encryption Padding (OAEP) [Bellare and Rogaway 1994] is a prominent public-key encryption scheme based on trapdoor permutations, most commonly used in combination with the RSA and Rabin functions. OAEP is widely deployed; many variants of OAEP are recommended by several standards, including IEEE P1363, PKCS, ISO 18033-2, ANSI X9, CRYPTREC and SET. Yet, the history of OAEP security is fraught with difficulties. The original paper of Bellare and Rogaway [1994] proves that, under the hypothesis that the underlying trapdoor permutation family is one-way, OAEP is semantically secure under chosen-ciphertext attacks. Shoup [2001] discovered later that this proof only established the security of OAEP against non-adaptive chosen-ciphertext attacks (IND-CCA), and not, as was believed at that time, against the stronger version of ciphertext indistinguishability that allows the adversary to adaptively obtain the decryption of ciphertexts of its choice (IND-CCA2). In response, Shoup suggested a modified scheme, secure against adaptive attacks under the one-wayness of the underlying permutation, and gave a proof of the adaptive security of the original scheme when it is used in combination with RSA with public exponent $e = 3$. Simultaneously, Fujisaki et al. [2004] proved that OAEP in its original formulation is indeed secure against adaptive attacks, but under the assumption that the underlying permutation family is *partial-domain* one-way. Since for the particular case of RSA this latter assumption is no stronger than (full-domain) one-wayness, this finally established the adaptive IND-CCA2 security of RSA-OAEP. Unfortunately, when one takes into account the additional cost of reducing the problem of inverting RSA to the problem of partially-inverting it, the security bound becomes less attractive. We note that there exist variants of OAEP that admit more efficient reductions when used in combination with the RSA and Rabin functions, notably SAEP, SAEP+ [Boneh 2001], and alternative schemes with tighter generic reductions, e.g. REACT [Okamoto and Pointcheval 2001b].

Here we focus on a machine-checked proof of the IND-CPA security of OAEP in the random oracle model, with a security bound that improves on the bound of Bellare and Rogaway [2006] game-based proof. We report as well on a significantly more challenging machine-checked proof of the IND-CCA2 security of OAEP in the random oracle model.

OAEP uses as a padding scheme a two-round Feistel network based on two hash functions G, H, and is commonly used together with the RSA and Rabin permutations. Figure 6.1 shows the Feistel network representation of the padding mappings $(\pi, \hat{\pi})$ used in OAEP for encryption and decryption.

During encryption, a message $m \in \{0,1\}^\mu$ is first padded with enough zeros to obtain a bitstring of length $k - \rho$, which is then fed to the encryption Feistel

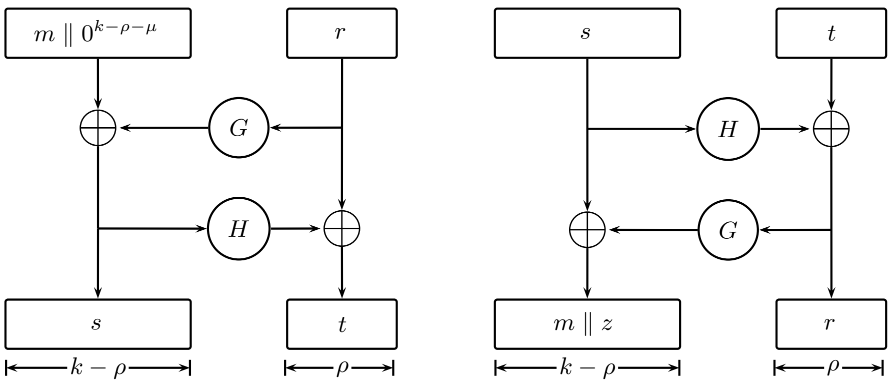


Fig. 6.1. Feistel network representation of the padding scheme used in OAEP for encryption (left) and decryption (right).

network together with a random bitstring $r \in \{0,1\}^\rho$; the ciphertext is the image under f of the resulting bitstring. To decrypt a ciphertext c, first compute its preimage $f^{-1}(sk, c)$ to obtain a bitstring of length k, and then run it through the reverse Feistel network to obtain a bitstring $(m \parallel z) \in \{0,1\}^{k-\rho}$. If z is not the all-zero bitstring, the ciphertext is rejected as invalid, otherwise, m is returned as its decryption. For concreteness, let us write down the definition of the OAEP scheme for a generic trapdoor permutation.

Definition 6.2 (OAEP encryption scheme). *Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor permutations on $\{0,1\}^k$, and let*

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

be two hash functions, with $\rho + \mu < k$. Let $k_1 = k - \rho - \mu$. The OAEP scheme is composed of the following triple of algorithms:

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

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

The way padding is handled plays a crucial role in the proof of IND-CCA2 security of the scheme in the random oracle model that we describe in Section 6.2. To prove that OAEP is IND-CCA2 secure, it is necessary to devise an efficient way to simulate the decryption oracle without having the trapdoor to the underlying permutation. It turns out that there is an efficient way to simulate the decryption oracle from the history of queries that an adversary made to the hash oracles G and H. Suppose the adversary queries the decryption oracle with a ciphertext $c = f(s \parallel t)$, then there are two possibilities:

1. Either s has been queried to H and $r = H(s) \oplus t$ has been queried to G. In that case, the corresponding plaintext can be found by inspecting the list of queries $(r', G(r'))$, $(s', H(s'))$ made to G and H respectively. For each pair of queries, check whether c coincides with $f(pk, s' \parallel (H(s') \oplus r'))$ and the $(k - \rho - \mu)$ least significant bits of $G(r') \oplus s'$ are all zeros. If one such pair of queries is found, the μ most significant bits of $G(r') \oplus s'$ form the corresponding plaintext;
2. Or else there is only a minute chance that the ciphertext be valid because it depends on a uniformly random value (either one of $G(r)$ and $H(s)$, or both). If the permutation is partial-domain one-way, the ciphertext may be safely rejected as invalid.

6.1 Indistinguishability under Chosen-Plaintext Attacks

Unsurprisingly, padding does not play any role in the proof of semantic security, because there is no need to simulate the decryption oracle. Therefore, in the rest of this section we assume without loss of generality that $\mu = k - \rho$.

Theorem 6.3 (IND-CPA security of OAEP). *Let $\mathcal{A}$ be an adversary against the semantic security of* OAEP *under chosen-message attacks that makes at most q_{G} queries to the hash oracle G and at most q_{H} queries to H. Suppose this adversary succeeds in guessing the hidden bit b with probability ϵ within time t. Then, there exists an inverter $\mathcal{I}$ that finds the preimage of an element uniformly drawn from the domain of the permutation f with probability ϵ' within time t', where*

$$\epsilon' \geq \epsilon - \left(\frac{1}{2} + \frac{q_{\mathsf{G}}}{2^\rho}\right) \tag{6.1}$$

$$t' \leq t + q_{\mathsf{G}}\ q_{\mathsf{H}}\ O(t_f) \tag{6.2}$$

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

Proof. We claim that the inverter $\mathcal{I}$ shown in the context of game G_{OW} in Figure 6.2 achieves the probability and time bounds in (6.1) and (6.2). The inverter runs the adversary $\mathcal{A}$ in a simulated environment. It intercepts queries to the hash oracles G and H and answers exactly as a random oracle would do, but keeps record of the queries and their responses. Instead of computing the challenge ciphertext to the IND-CPA adversary as an LR-oracle would do in the real IND-CPA game, it replaces the challenge ciphertext with its own challenge y. When adversary $\mathcal{A}$ halts, the inverter $\mathcal{I}$ inspects the history of queries that were made to the hash oracles G and H and tries to reconstruct a preimage of its challenge y under f. It turns out that the probability that the inverter succeeds in reconstructing such a preimage is closely related to the probability with which adversary $\mathcal{A}$ wins the IND-CPA game.

The upper bound on the execution time of the inverter can be justified by examining just the game G_{OW}. The only non-constant overhead incurred is in the

```
Game $\mathsf{G_{OW}}$ :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $y \xleftarrow{\$} \{0,1\}^k$;
  $x \leftarrow \mathcal{I}(pk, y)$

Adversary $\mathcal{I}(pk, y)$ :
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$;
  if $\exists r \in$ dom$(\boldsymbol{L}), (s, h) \in \boldsymbol{M}$.
      $f(pk, s \parallel h \oplus r) = y$
  then return $s \parallel (h \oplus r)$
  else return $0^k$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

Fig. 6.2. The inverter $\mathcal{I}$ in the context of the one-wayness game for the family of trapdoor permutations $(\mathcal{KG}_f, f, f^{-1})$.

reconstruction of the preimage of the challenge by inspecting the history of queries. In the worst case, it amounts to q_H computations of the permutation f for each of the q_G queries, from which (6.2) follows directly.

To prove that the inverter achieves the probability bound (6.1), we will exhibit a sequence of games relating the probability

$$\epsilon' = \Pr\left[\mathsf{G_{OW}} : x = f^{-1}(sk, y)\right]$$

of $\mathcal{I}$ successfully inverting f on its challenge y, to the probability

$$\epsilon = \Pr\left[\mathsf{G_{INDCPA}} : b = \tilde{b}\right]$$

of the adversary $\mathcal{A}$ correctly guessing the value of the hidden bit b in the IND-CPA game. We start from the IND-CPA game where oracles G and H are implemented as random oracles—this is justified by the random oracle model.

```
Game $\mathsf{G_{INDCPA}}$ :
  $(pk, sk) \leftarrow \mathcal{KG}()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $y \leftarrow \mathcal{E}(pk, m_b)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

The hypothesis on the bound on the number of queries the adversary makes to G can be readily encoded as:

$$\Pr\left[\mathsf{G}_{\mathsf{INDCPA}} : |L| \le q_{\mathsf{G}}\right] = 1$$

In the following game, we inline the key generation and the encryption of m_b, which uses a random seed $\hat{\boldsymbol{r}}$, and we eagerly sample the response that the random oracle G gives back if $\hat{\boldsymbol{r}}$ is ever queried:

```
Game G_1 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{\boldsymbol{g}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $g \leftarrow G(\hat{\boldsymbol{r}})$; $s \leftarrow g \oplus m_b$;
  $h \leftarrow H(s)$; $t \leftarrow h \oplus r$;
  $y \leftarrow f(pk, s \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle G(r) :
  if $r \notin$ dom($\boldsymbol{L}$) then
    if $r = \hat{\boldsymbol{r}}$ then $g \leftarrow \hat{\boldsymbol{g}}$
    else $g \xleftarrow{\$} \{0,1\}^{k-\rho}$
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle H(s) :
  if $s \notin$ dom($\boldsymbol{M}$) then
    $h \xleftarrow{\$} \{0,1\}^\rho$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

The resulting game G_1 is semantically equivalent to the initial game, and thus we have

$$\Pr\left[\mathsf{G}_{\mathsf{INDCPA}} : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_1 : b = \tilde{b}\right] \tag{6.3}$$

Our objective is now to eliminate $\hat{\boldsymbol{g}}$ from the code of the oracle G, because if we manage to do so, we will be able to make s completely random and remove the dependency of y on b.

In the next game we modify the oracle G so that if $\hat{\boldsymbol{r}}$ is ever queried, a flag **bad** is set to true and the answer is not recorded in the oracle memory.

```
Game G_2 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{\boldsymbol{g}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $s \leftarrow \hat{\boldsymbol{g}} \oplus m_b$;
  $h \leftarrow H(s)$; $t \leftarrow h \oplus \hat{\boldsymbol{r}}$;
  $y \leftarrow f(pk, s \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle G(r) :
  if $r = \hat{\boldsymbol{r}}$ then
    bad $\leftarrow$ true; return $\hat{\boldsymbol{g}}$
  else
    if $r \notin$ dom($\boldsymbol{L}$) then
      $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
      $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
    return $\boldsymbol{L}[r]$

Oracle H(s) :
  if $s \notin$ dom($\boldsymbol{M}$) then
    $h \xleftarrow{\$} \{0,1\}^\rho$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

In order to justify this transformation, we define the following relational invariant:

$$\phi_{12} \overset{\text{def}}{=} (\hat{\boldsymbol{r}} \in \mathsf{dom}(\boldsymbol{L}) \implies \boldsymbol{L}[\hat{\boldsymbol{r}}] = \hat{\boldsymbol{g}})\langle 1\rangle \wedge (\forall r.\ r \neq \hat{\boldsymbol{r}}\langle 1\rangle \implies \boldsymbol{L}[r]\langle 1\rangle = \boldsymbol{L}[r]\langle 2\rangle)$$

The above invariant allows us to prove that oracle G answers queries in the same way in games G_1 and G_2. We prove that this invariant is established after $\boldsymbol{L}$ is initialized in G_1 and G_2, and that the implementations of G and H preserve it. We know from the fact that $\mathcal{A}$ is well-formed that it cannot directly modify the global variables $\boldsymbol{L}$, $\hat{\boldsymbol{r}}$, and $\hat{\boldsymbol{g}}$. Therefore, the invariant ϕ_{12} is preserved through calls to the adversary. We then inline the call to G in the body of game G_1. At this point the invariant holds and either the adversary $\mathcal{A}$ has already queried $\hat{\boldsymbol{r}}$, in which case it follows from the invariant that the answer to the call is $\hat{\boldsymbol{g}}$, or it has never queried $\hat{\boldsymbol{r}}$, in which case the same follows directly by applying dead code elimination and constant propagation. Hence,

$$\Pr\left[\mathsf{G}_1 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_2 : b = \tilde{b}\right] \tag{6.4}$$

We will now use the Fundamental Lemma to remove $\hat{\boldsymbol{g}}$ altogether from the code of oracle G. We define a game G_3 syntactically identical up to **bad** to the previous game.

```
Game G_3 :
$(pk, sk) \leftarrow \mathcal{KG}_f()$;
$\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
$\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
$\hat{\boldsymbol{g}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
$(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
$b \xleftarrow{\$} \{0,1\}$;
$s \leftarrow \hat{\boldsymbol{g}} \oplus m_b$;
$h \leftarrow H(s)$; $t \leftarrow h \oplus \hat{\boldsymbol{r}}$;
$y \leftarrow f(pk, s \parallel t)$;
$\tilde{b} \leftarrow \mathcal{A}_2(y)$
```

```
Oracle $G(r)$ :
if $r = \hat{\boldsymbol{r}}$ then
  bad ← true;
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$
else
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ : ...
```

It follows from the Fundamental Lemma that

$$\left|\Pr\left[\mathsf{G}_2 : b = \tilde{b}\right] - \Pr\left[\mathsf{G}_3 : b = \tilde{b}\right]\right| \leq \Pr\left[\mathsf{G}_3 : \textbf{bad}\right] \tag{6.5}$$

Since $\hat{\boldsymbol{g}}$ no longer appears in G, we can now sample it later in the game. We also simplify the implementation of oracle G for the sake of readability by coalescing the portion of code appearing in both branches of the conditional.

```
Game G_4 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{\boldsymbol{g}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $s \leftarrow \hat{\boldsymbol{g}} \oplus m_b$;
  $h \leftarrow H(s)$; $t \leftarrow h \oplus \hat{\boldsymbol{r}}$;
  $y \leftarrow f(pk, s \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom($\boldsymbol{M}$) then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We have

$$\Pr\left[\mathsf{G}_3 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_4 : b = \tilde{b}\right] \tag{6.6}$$

$$\Pr\left[\mathsf{G}_3 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_4 : \mathbf{bad}\right] \tag{6.7}$$

We make now a transformation that relies on algebraic properties of the exclusive or operator: instead of sampling $\hat{\boldsymbol{g}}$ and defining s in terms of it, we can sample s and define $\hat{\boldsymbol{g}}$ in terms of s. This is justified by the following program equivalence:

$$\vdash \hat{\boldsymbol{g}} \xleftarrow{\$} \{0,1\}^{k-\rho};\ s \leftarrow \hat{\boldsymbol{g}} \oplus m_b \simeq^{\{m_b\}}_{\{\hat{\boldsymbol{g}}, s, m_b\}} s \xleftarrow{\$} \{0,1\}^{k-\rho};\ \hat{\boldsymbol{g}} \leftarrow s \oplus m_b$$

This transformation is sometimes called *optimistic sampling* and is a pattern that appears recurrently in game-based proofs; we gave a proof using the relational Hoare logic in Section 3.2.2.

We can now eliminate the assignment to $\hat{\boldsymbol{g}}$ as dead code, and sample s at the beginning of the game. Since y no longer depends on the hidden bit b, we can sample b at the end of the game. The resulting game G_5 is:

```
Game G_5 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $s \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $h \leftarrow H(s)$; $t \leftarrow h \oplus \hat{\boldsymbol{r}}$;
  $y \leftarrow f(pk, s \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$;
  $b \xleftarrow{\$} \{0,1\}$

Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom($\boldsymbol{M}$) then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We have $\Pr\left[\mathsf{G}_4 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_5 : \mathbf{bad}\right]$. It is obvious that, in the above game, $\tilde{b}$ and b are independent and thus

$$\Pr\left[\mathsf{G}_4 : b = \tilde{b}\right] = \Pr\left[\mathsf{G}_5 : b = \tilde{b}\right] = \frac{1}{2} \tag{6.8}$$

The objective now is to bound the probability of **bad** being set. We no longer care about b, so we remove the instruction sampling it. We reallocate s to a global variable $\hat{\boldsymbol{s}}$ and we eagerly sample the value of $H(\hat{\boldsymbol{s}})$.

```
Game G_6 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \xleftarrow{\$} \{0,1\}^\rho$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $h \leftarrow H(\hat{\boldsymbol{s}})$; $t \leftarrow h \oplus \hat{\boldsymbol{r}}$;
  $y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom($\boldsymbol{M}$) then
    if $s = \hat{\boldsymbol{s}}$ then $h \leftarrow \hat{\boldsymbol{h}}$
    else $h \xleftarrow{\$} \{0,1\}^\rho$
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We then modify oracle H so that it does not record the answer in its memory if $\hat{\boldsymbol{s}}$ is ever queried, and we inline the call to H in the body of the game. To prove that this modification to H is transparent to the adversary we prove the following relational invariant between G_6 and G_7:

$$(\hat{\boldsymbol{s}} \in \mathsf{dom}(\boldsymbol{M}) \implies \boldsymbol{M}[\hat{s}] = \hat{\boldsymbol{h}})\langle 1\rangle \ \wedge\ (\forall s.\ s \neq \hat{\boldsymbol{s}}\langle 1\rangle \implies \boldsymbol{M}[s]\langle 1\rangle = \boldsymbol{M}[s]\langle 2\rangle)$$

```
Game G_7 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \xleftarrow{\$} \{0,1\}^\rho$;
  $t \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s = \hat{\boldsymbol{s}}$ then return $\hat{\boldsymbol{h}}$
  else if $s \notin$ dom($\boldsymbol{M}$) then
    $h \xleftarrow{\$} \{0,1\}^\rho$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

Thus

$$\Pr\left[\mathsf{G}_5 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_6 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_7 : \mathbf{bad}\right] \tag{6.9}$$

We then revert to the previous implementation of oracle H, which stores the answer to a query $\hat{\boldsymbol{s}}$ in its memory. This allows us to bound the probability of **bad** being set by analyzing two different cases:

1. either the adversary queried $\hat{\boldsymbol{s}}$ to H before **bad** is set, in which case we set a flag $\mathbf{bad}_1$;
2. or it did not, in which case we set a flag $\mathbf{bad}_2$.

```
Game $\mathsf{G}_8$ :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \stackrel{\$}{\leftarrow} \{0,1\}^\rho$;
  $\hat{\boldsymbol{s}} \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \stackrel{\$}{\leftarrow} \{0,1\}^\rho$;
  $t \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$
```

```
Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}}$ then
    if $\hat{\boldsymbol{s}} \in$ dom$(\boldsymbol{M})$
    then $\mathbf{bad}_1 \leftarrow$ true
    else $\mathbf{bad}_2 \leftarrow$ true
  ...
Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{M})$ then
    if $s = \hat{\boldsymbol{s}}$ then $h \leftarrow \hat{\boldsymbol{h}}$
    else $h \stackrel{\$}{\leftarrow} \{0,1\}^\rho$
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We prove the following relational invariant between G_7 and G_8:

$$\mathbf{bad}\langle 1\rangle \implies (\mathbf{bad}_1 \vee \mathbf{bad}_2)\langle 2\rangle$$

Hence we have by the union bound,

$$\Pr\left[\mathsf{G}_7 : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_8 : \mathbf{bad}_1 \vee \mathbf{bad}_2\right] \leq \Pr\left[\mathsf{G}_8 : \mathbf{bad}_1\right] + \Pr\left[\mathsf{G}_8 : \mathbf{bad}_2\right] \quad (6.10)$$

We split the sequence of games and bound $\mathbf{bad}_1$ and $\mathbf{bad}_2$ separately; we deal with $\mathbf{bad}_1$ first.

We slice the assignment to $\mathbf{bad}_2$ off the code of G, we apply again the optimistic sampling transformation to sample t instead of $\hat{\boldsymbol{r}}$, and we reallocate t, y and pk to global variables, obtaining:

```
Game $\mathsf{P}_1$ :
  $(\boldsymbol{pk}, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{s}} \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{t}} \stackrel{\$}{\leftarrow} \{0,1\}^\rho$;
  $\hat{\boldsymbol{h}} \stackrel{\$}{\leftarrow} \{0,1\}^\rho$;
  $\hat{\boldsymbol{r}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{t}}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $\boldsymbol{y} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\boldsymbol{y})$
```

```
Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}} \wedge \hat{\boldsymbol{s}} \in$ dom$(\boldsymbol{M})$ then
    $\mathbf{bad}_1 \leftarrow$ true
  if $r \notin$ dom$(\boldsymbol{L})$ then
    $g \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$
Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{M})$ then
    if $s = \hat{\boldsymbol{s}}$ then $h \leftarrow \hat{\boldsymbol{h}}$
    else $h \stackrel{\$}{\leftarrow} \{0,1\}^\rho$
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We now replace the condition under which $\mathbf{bad}_1$ is set to true by an equivalent one. We prove that P_1 satisfies the invariant

$$(\forall (s,h) \in \boldsymbol{M}.\ s = \hat{\boldsymbol{s}} \implies h = \hat{\boldsymbol{h}}) \wedge (\hat{\boldsymbol{r}} = \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{t}} \ \wedge\ \boldsymbol{y} = f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}}))$$

From this invariant we have that

$$r = \hat{\boldsymbol{r}} \wedge \hat{\boldsymbol{s}} \in \mathsf{dom}(\boldsymbol{M}) \iff \exists (s,h) \in \boldsymbol{M}.\ f(\boldsymbol{pk}, s \parallel (h \oplus r)) = \boldsymbol{y}$$

Therefore we can reformulate the condition under which $\mathbf{bad}_1$ is set in G, and remove $\hat{\boldsymbol{r}}$ since it is no longer used.

```
Game P_2 :
  $(\boldsymbol{pk}, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{t}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{h}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $\boldsymbol{y} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\boldsymbol{y})$

Oracle G(r) :
  if $\exists (s,h) \in \boldsymbol{M}.$
     $f(\boldsymbol{pk}, s \parallel h \oplus r) = \boldsymbol{y}$ then
    $\mathbf{bad}_1 \leftarrow$ true
  if $r \notin \mathsf{dom}(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r,g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle H(s) :
  if $s \notin \mathsf{dom}(\boldsymbol{M})$ then
    if $s = \hat{\boldsymbol{s}}$ then $h \leftarrow \hat{\boldsymbol{h}}$
    else $h \xleftarrow{\$} \{0,1\}^{\rho}$
    $\boldsymbol{M} \leftarrow (s,h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We have

$$\Pr[\mathsf{G}_8 : \mathbf{bad}_1] = \Pr[\mathsf{P}_1 : \mathbf{bad}_1] = \Pr[\mathsf{P}_2 : \mathbf{bad}_1] \tag{6.11}$$

Next, we revert H to the original random oracle implementation and we slice away $\hat{\boldsymbol{h}}$ as dead code. In the resulting game $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ are used only to compute $\boldsymbol{y} = f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$. Since $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ are uniformly sampled bitstrings, their concatenation is uniformly distributed, and since f is a permutation, the value of $\boldsymbol{y}$ is uniformly distributed. This reasoning is summarized in the following program equivalence

$$\vdash \hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho};\ \hat{\boldsymbol{t}} \xleftarrow{\$} \{0,1\}^{\rho};\ \boldsymbol{y} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}}) \simeq^{\{\boldsymbol{pk}\}}_{\{\boldsymbol{pk},\boldsymbol{y}\}} \boldsymbol{y} \xleftarrow{\$} \{0,1\}^{k}$$

```
Game P_3 :
  $(\boldsymbol{pk}, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow \mathsf{nil}$;
  $\boldsymbol{y} \xleftarrow{\$} \{0,1\}^k$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\boldsymbol{y})$

Oracle $G(r)$ :
  if $\exists (s,h) \in \boldsymbol{M}.$
      $f(\boldsymbol{pk}, s \parallel h \oplus r) = \boldsymbol{y}$ then
    $\mathbf{bad}_1 \leftarrow \mathsf{true}$
  if $r \notin \mathsf{dom}(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r,g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin \mathsf{dom}(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s,h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We now revert G to its original implementation and prove the following relational invariant between P_3 and P_4 which gives a necessary condition for $\mathbf{bad}_1$ to be set:

$$\mathbf{bad}_1\langle 1\rangle \implies (\exists r \in \mathsf{dom}(\boldsymbol{L}), (s,h) \in \boldsymbol{M}.\ f(\boldsymbol{pk}, s \parallel (h \oplus r)) = \boldsymbol{y})\langle 2\rangle$$

Thus,

$$\Pr[\mathsf{P}_3 : \mathbf{bad}_1] \le \Pr[\mathsf{P}_4 : \exists r \in \mathsf{dom}(\boldsymbol{L}), (s,h) \in \boldsymbol{M}.\ f(\boldsymbol{pk}, s \parallel (h \oplus r)) = \boldsymbol{y}] \quad (6.12)$$

```
Game P_4 :
  $(\boldsymbol{pk}, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow \mathsf{nil}$;
  $\boldsymbol{y} \xleftarrow{\$} \{0,1\}^k$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\boldsymbol{y})$

Oracle $G(r)$ :
  if $r \notin \mathsf{dom}(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r,g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin \mathsf{dom}(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s,h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

We reallocate variables $\boldsymbol{pk}$ and $\boldsymbol{y}$ to local variables, and at the end of the game compute x as the inverter in Figure 6.2 would do.

```
Game P_5 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $y \xleftarrow{\$} \{0,1\}^k$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$;
  if $\exists r \in \mathsf{dom}(\boldsymbol{L}), (s,h) \in \boldsymbol{M}.$
        $f(pk, s \parallel h \oplus r) = y$
  then $x \leftarrow s \parallel (h \oplus r)$
  else  $x \leftarrow 0^k$

Oracle $G(r)$ :
  if $r \notin \mathsf{dom}(\boldsymbol{L})$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin \mathsf{dom}(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

Compare this last game to game $\mathsf{G_{OW}}$ in Figure 6.2: both games semantically equivalent once the call to $\mathcal{I}$ in $\mathsf{G_{OW}}$ is inlined. Furthermore,

$$\begin{aligned}\Pr[\mathsf{P}_4 : \exists r \in \mathsf{dom}(\boldsymbol{L}), (s,h) \in \boldsymbol{M}.\ f(\boldsymbol{pk}, s \parallel (h \oplus r)) = \boldsymbol{y}]&\\ \leq \Pr\left[\mathsf{P}_5 : x = f^{-1}(sk, y)\right]&\\ = \Pr\left[\mathsf{G_{OW}} : x = f^{-1}(sk, y)\right]&\end{aligned} \tag{6.13}$$

From (6.11)–(6.13) we get

$$\Pr[\mathsf{G}_8 : \mathbf{bad}_1] \leq \Pr\left[\mathsf{G_{OW}} : x = f^{-1}(sk, y)\right] \tag{6.14}$$

We are now left with the problem of bounding $\mathbf{bad}_2$ in game G_8. Define the game Q parametrized by two instructions c_1 and c_2 as follows

```
Game Q($c_1, c_2$) :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $t \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$

Oracle $G(r)$ :
  if $r = \hat{\boldsymbol{r}} \wedge \hat{\boldsymbol{s}} \notin \mathsf{dom}(\boldsymbol{M})$ then
    $\mathbf{bad}_2 \leftarrow$ true
  ...

Oracle $H(s)$ :
  if $s \notin \mathsf{dom}(\boldsymbol{M})$ then
    $h \xleftarrow{\$} \{0,1\}^{\rho}$;
    if $s = \hat{\boldsymbol{s}} \wedge \mathbf{bad}_2$ then
      $\mathbf{bad}_2, \mathbf{bad}_3 \leftarrow$ true;  $c_1$
    if $s = \hat{\boldsymbol{s}} \wedge \neg\mathbf{bad}_2$ then
      $\mathbf{bad}_4 \leftarrow$ true;  $c_2$
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

Flag $\mathbf{bad}_3$ is set when $\hat{\boldsymbol{s}}$ is first queried to H and $G(\hat{\boldsymbol{r}})$ has already been queried. Flag $\mathbf{bad}_4$ is set when $\hat{\boldsymbol{s}}$ is first queried to H and $G(\hat{\boldsymbol{r}})$ has not yet been queried. Consider the games

$$\begin{aligned}\mathsf{Q}_1 &\stackrel{\text{def}}{=} \mathsf{Q}(h \leftarrow \hat{\boldsymbol{h}}\qquad\quad, h \leftarrow \hat{\boldsymbol{h}})\\ \mathsf{Q}_2 &\stackrel{\text{def}}{=} \mathsf{Q}(h \xleftarrow{\$} \{0,1\}^{\rho}, h \leftarrow \hat{\boldsymbol{h}})\\ \mathsf{Q}_3 &\stackrel{\text{def}}{=} \mathsf{Q}(h \xleftarrow{\$} \{0,1\}^{\rho}, h \xleftarrow{\$} \{0,1\}^{\rho})\end{aligned}$$

Using the fundamental lemma together with appropriate progam invariants, we obtain

$$
\begin{aligned}
\Pr[\mathsf{G}_8 : \mathbf{bad}_2] &= \Pr[\mathsf{Q}_1 : \mathbf{bad}_2] \\
&= \Pr[\mathsf{Q}_2 : \mathbf{bad}_2] && (\text{FL}, \mathbf{bad}_2) \\
&= \Pr[\mathsf{Q}_2 : \mathbf{bad}_2 \wedge \mathbf{bad}_4] + \Pr[\mathsf{Q}_2 : \mathbf{bad}_2 \wedge \neg\mathbf{bad}_4] \\
&= \Pr[\mathsf{Q}_2 : \mathbf{bad}_2 \wedge \mathbf{bad}_4] + \Pr[\mathsf{Q}_3 : \mathbf{bad}_2 \wedge \neg\mathbf{bad}_4] && (\text{FL}, \mathbf{bad}_4) \\
&= 0 \qquad\qquad\qquad\qquad\quad + \Pr[\mathsf{Q}_3 : \mathbf{bad}_2 \wedge \neg\mathbf{bad}_4] && (\mathbf{bad}_4 \Longrightarrow \neg\mathbf{bad}_2) \\
&= \Pr[\mathsf{Q}_3 : \mathbf{bad}_2 \wedge \mathbf{bad}_4] + \Pr[\mathsf{Q}_3 : \mathbf{bad}_2 \wedge \neg\mathbf{bad}_4] && (\mathbf{bad}_4 \Longrightarrow \neg\mathbf{bad}_2) \\
&= \Pr[\mathsf{Q}_3 : \mathbf{bad}_2]
\end{aligned}
$$

We use optimistic sampling and the fact that f is a permutation to prove the following equivalence:

$$
\begin{array}{l}
\hat{\boldsymbol{r}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}; \\
\hat{\boldsymbol{s}} \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}; \\
\hat{\boldsymbol{h}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}; \\
t \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}; \\
y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)
\end{array}
\simeq^{\emptyset}_{\{\hat{\boldsymbol{r}},y\}}
\begin{array}{l}
\hat{\boldsymbol{r}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}; \\
\hat{\boldsymbol{s}} \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}; \\
t \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}; \\
\hat{\boldsymbol{h}} \leftarrow t \oplus \hat{\boldsymbol{r}}; \\
y \leftarrow f(pk, \hat{\boldsymbol{s}} \parallel t)
\end{array}
\simeq^{\emptyset}_{\{\hat{\boldsymbol{r}},y\}}
\begin{array}{l}
\hat{\boldsymbol{r}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}; \\
y \stackrel{\$}{\leftarrow} \{0,1\}^{k};
\end{array}
$$

In the next game, we apply the above equivalence and we revert oracles G and H to the original random oracles,

```
Game Q_4 :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\boldsymbol{L}, \boldsymbol{M} \leftarrow$ nil;
  $y \stackrel{\$}{\leftarrow} \{0,1\}^k$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$;
  $\hat{\boldsymbol{r}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}$;
```

```
Oracle $G(r)$ :
  if $r \notin$ dom($\boldsymbol{L}$) then
    $g \stackrel{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L} \leftarrow (r, g) :: \boldsymbol{L}$
  return $\boldsymbol{L}[r]$

Oracle $H(s)$ :
  if $s \notin$ dom($\boldsymbol{M}$) then
    $h \stackrel{\$}{\leftarrow} \{0,1\}^{\rho}$;
    $\boldsymbol{M} \leftarrow (s, h) :: \boldsymbol{M}$
  return $\boldsymbol{M}[s]$
```

If $\mathbf{bad}_2$ is set to true in Q_3 then $\hat{\boldsymbol{r}}$ must be in $\boldsymbol{L}$, i.e. the following is a relational invariant of Q_3 and Q_4,

$$\mathbf{bad}_2\langle 1\rangle \implies (\hat{\boldsymbol{r}} \in \mathsf{dom}(\boldsymbol{L}))\langle 2\rangle$$

and we can readily bound the probability of $\hat{\boldsymbol{r}}$ being in the domain of $\boldsymbol{L}$ in Q_4 because we know that the adversary makes at most q_{G} calls to G,

$$\Pr[\mathsf{G}_8 : \mathbf{bad}_2] = \Pr[\mathsf{Q}_3 : \mathbf{bad}_2] \leq \Pr[\mathsf{Q}_4 : \hat{\boldsymbol{r}} \in \mathsf{dom}(\boldsymbol{L})] \leq \frac{q_{\mathsf{G}}}{2^{\rho}} \tag{6.15}$$

Putting (6.10), (6.14) and (6.15) together,

$$\Pr\left[\mathsf{G}_7 : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_{\mathsf{OW}} : x = f^{-1}(sk, y)\right] + \frac{q_{\mathsf{G}}}{2^{\rho}} \tag{6.16}$$

Finally from (6.3)–(6.9) and (6.16),

$$\left|\Pr\left[\mathsf{G}_{\mathsf{INDCPA}} : b = \tilde{b}\right] - \frac{1}{2}\right| \leq \Pr\left[\mathsf{G}_{\mathsf{OW}} : x = f^{-1}(sk, y)\right] + \frac{q_{\mathsf{G}}}{2^{\rho}} \tag{6.17}$$

From which follows the bound (6.1). □

6.2 Indistinguishability under Chosen-Ciphertext Attacks

We use as a starting point the proof of Pointcheval [2005], which fills several gaps in the reduction of Fujisaki et al. [2004], resulting in a weaker security bound. Our proof unveils minor glitches in the proof of Pointcheval [2005] and marginally improves on its exact security bound (reducing the coefficients) by performing an aggressive analysis of oracle queries earlier in the sequence of games. The initial and final games of the reduction appear in Figure 6.3; the proof in CertiCrypt is about 10,000 lines long.

Theorem 6.4 (IND-CCA2 security of OAEP). *Let $\mathcal{A}$ be an adversary against the ciphertext indistinguishability of OAEP under an adaptive chosen-ciphertext attack that makes at most q_{G} and q_{H} queries to the hash oracles G and H, respectively, and at most $q_{\mathcal{D}}$ queries to the decryption oracle $\mathcal{D}$.*[1] *Suppose $\mathcal{A}$ achieves an advantage ϵ within time t during game $\mathsf{G}_{\mathsf{INDCCA}}$. Then, there exists an inverter $\mathcal{I}$ that finds a partial preimage (the $k - \rho$ most significant bits of a preimage) of an element uniformly drawn from the domain of f with probability ϵ' within time t' during experiment $\mathsf{G}_{\mathsf{PDOW}}$, where*

$$t' \leq t + q_{\mathsf{G}}\ q_{\mathsf{H}}\ q_{\mathcal{D}}\ O(t_f) \qquad\qquad \epsilon' \geq \frac{1}{q_{\mathsf{H}}}\left(\epsilon - \frac{3q_{\mathcal{D}}q_{\mathsf{G}} + q_{\mathcal{D}}^2 + 4q_{\mathcal{D}} + q_{\mathsf{G}}}{2^{\rho}} - \frac{2q_{\mathcal{D}}}{2^{k_1}}\right)$$

6.2.1 Proof Outline

Figure 6.4 outlines the structure of the proof; the first step from $\mathsf{G}_{\mathsf{INDCCA}}$ to G_1 and the final step from G_5 to $\mathsf{G}_{\mathsf{PDOW}}$ are not displayed. The reduction successively eliminates all situations in which the plaintext extractor used by the inverter to simulate decryption may fail.

Starting from game $\mathsf{G}_{\mathsf{INDCCA}}$, we use the logic of swapping statements to fix the hash $\hat{\boldsymbol{g}}$ that G gives in response to the random seed of the challenge ciphertext; the computation of the challenge ciphertext unfolds to:

[1] The machine-checked proof slightly relaxes this condition; it requires that the length of $\boldsymbol{L}_G$ be at most $q_{\mathcal{D}} + q_{\mathsf{G}}$, so that the adversary could trade queries to $\mathcal{D}$ for queries to G.

```
Game $\mathsf{G}_{\mathsf{INDCCA}}$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow \mathsf{nil}$;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}()$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{\boldsymbol{c}} \leftarrow \mathcal{E}(m_b)$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow \mathsf{true}$;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $G(r)$ :
  if $r \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G \leftarrow (r, g) :: \boldsymbol{L}_G$
  return $\boldsymbol{L}_G[r]$

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

Oracle $\mathcal{D}(c)$ :
  $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$;
  $(s, t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
  $h \leftarrow H(s)$;
  $r \leftarrow t \oplus h$;
  $g \leftarrow G(r)$;
  if $[s \oplus g]_{k_1} = 0^{k_1}$ then
    return $[s \oplus g]^{\mu}$
  else return $\perp$
```

```
Game $\mathsf{G}_{\mathsf{PDOW}}$ :
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $s \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $t \xleftarrow{\$} \{0,1\}^{\rho}$;
  $\tilde{s} \leftarrow \mathcal{I}(pk, f(pk, s \parallel t))$

Adversary $\mathcal{I}(pk, y)$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H \leftarrow \mathsf{nil}$;
  $\hat{\boldsymbol{pk}} \leftarrow pk$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $\tilde{b} \leftarrow \mathcal{A}_2(y)$;
  $s \xleftarrow{\$} \mathsf{dom}(\boldsymbol{L}_H)$;
  return $s$

Oracle $G(r)$ :
  if $r \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G \leftarrow (r, g) :: \boldsymbol{L}_G$
  return $\boldsymbol{L}_G[r]$

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

Oracle $\mathcal{D}(c)$ :
  if $\exists (s, h) \in \boldsymbol{L}_H, (r, g) \in \boldsymbol{L}_G.$
    $c = f(\hat{\boldsymbol{pk}}, s \parallel (r \oplus h)) \wedge$
    $[s \oplus g]_{k_1} = 0^{k_1}$
  then return $[s \oplus g]^{\mu}$
  else return $\perp$
```

Fig. 6.3. Initial and final games in the reduction of the IND-CCA2 security of OAEP to the problem of partially inverting the underlying permutation. We exclude cheating adversaries who query the decryption oracle with the challenge ciphertext during the second phase of the experiment by requiring $(\mathsf{true}, \hat{\boldsymbol{c}}) \notin \boldsymbol{L}_{\mathcal{D}}$ to be a post-condition of the initial game.

$$\begin{aligned}
&\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho};\\
&\hat{\boldsymbol{s}} \leftarrow \hat{\boldsymbol{g}} \oplus (m_b \parallel 0^{k_1});\\
&\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}});\\
&\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}};\\
&\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})
\end{aligned}$$

where $\hat{\boldsymbol{g}}$ is sampled from $\{0,1\}^{k-\rho}$ before the first call to $\mathcal{A}$. We then make G respond to an adversary query $\hat{\boldsymbol{r}}$ with a freshly sampled value instead of $\hat{\boldsymbol{g}}$; this only makes a difference if flag **bad** is set in game G_1. Since at this point $\hat{\boldsymbol{g}}$ is uniformly distributed and independent from the adversary's view, the value $\hat{\boldsymbol{s}}$ computed as $\hat{\boldsymbol{g}} \oplus (m_b \parallel 0^{k_1})$ is as well uniformly distributed and independent from the adversary's view. This removes the dependence of the adversary output on the hidden bit b, and thus the probability of a correct guess is exactly 1/2. Using the Fundamental Lemma we obtain the bound:

$$\begin{aligned}
\Pr\left[\mathsf{G}_{\mathsf{IND\text{-}CCA2}} : \tilde{b} = b\right] - \Pr\left[\mathsf{G}_1 : \tilde{b} = b\right] &= \Pr\left[\mathsf{G}_{\mathsf{IND\text{-}CCA2}} : \tilde{b} = b\right] - \frac{1}{2}\\
&\leq \Pr\left[\mathsf{G}_1 : \mathbf{bad}\right] \qquad (6.18)
\end{aligned}$$

```
Game $G_1$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $\hat{\boldsymbol{r}} \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \overset{\$}{\leftarrow} \{0,1\}$;
  $\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}})$;
  $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L}_G)$ then
    if $r = \hat{\boldsymbol{r}}$ then
      bad $\leftarrow$ true;
    $g \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G[r] \leftarrow g$
  else $g \leftarrow \boldsymbol{L}_G[r]$
  return $g$

Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{L}_H)$ then
    $h \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
    $\boldsymbol{L}_H[s] \leftarrow h$
  else $h \leftarrow \boldsymbol{L}_H[s]$
  return $h$

Oracle $\mathcal{D}(c)$ :
  if $(\hat{\boldsymbol{c}}_{\mathsf{def}} \wedge \hat{\boldsymbol{c}} = c) \vee q_{\mathcal{D}} < |\boldsymbol{L}_{\mathcal{D}}| \vee q_{\mathcal{D}} + q_{\mathsf{G}} < |\boldsymbol{L}_G|$ then
    return $\bot$
  else
    $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$;
    $(s,t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
    $r \leftarrow t \oplus H(s)$;
    $g \leftarrow G(r)$;
    if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^{\mu}$ else return $\bot$
```

$\Pr[\mathsf{G}_1 : \mathbf{bad}] \leq \Pr[\mathsf{G}_2 : \mathbf{bad}] + \dfrac{q_{\mathcal{D}}^2 + q_{\mathcal{D}} q_{\mathsf{G}} + q_{\mathcal{D}}}{2^{\rho}} + \dfrac{q_{\mathcal{D}}}{2^{k_1}}$

Inline G and case analysis on $s \in$ dom$(\boldsymbol{L}_H)$ in $\mathcal{D}$. Reject ciphertexts with a fresh g or h

```
Game $G_2$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $\hat{\boldsymbol{r}} \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \overset{\$}{\leftarrow} \{0,1\}$;
  $\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}})$;
  $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L}_G)$ then
    if $r = \hat{\boldsymbol{r}}$ then
      bad $\leftarrow$ true;
    $g \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G[r] \leftarrow g$
  else $g \leftarrow \boldsymbol{L}_G[r]$
  return $g$

Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{L}_H)$ then
    $h \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
    $\boldsymbol{L}_H[s] \leftarrow h$
  else $h \leftarrow \boldsymbol{L}_H[s]$
  return $h$

Oracle $\mathcal{D}(c)$ :
  if $(\hat{\boldsymbol{c}}_{\mathsf{def}} \wedge \hat{\boldsymbol{c}} = c) \vee q_{\mathcal{D}} < |\boldsymbol{L}_{\mathcal{D}}| \vee q_{\mathcal{D}} + q_{\mathsf{G}} < |\boldsymbol{L}_G|$ then
    return $\bot$
  else
    $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$;
    $(s,t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
    if $s \in$ dom$(\boldsymbol{L}_H)$ then
      $r \leftarrow t \oplus H(s)$;
      if $r \in$ dom$(\boldsymbol{L}_G)$ then
        $g \leftarrow \boldsymbol{L}_G[r]$;
        if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^{\mu}$ else return $\bot$
      else
        if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true;
        $g \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow g$; return $\bot$
    else
      $r \leftarrow t \oplus H(s)$;
      if $r \notin$ dom$(\boldsymbol{L}_G)$ then $g \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow g$
      return $\bot$
```

$\Pr[\mathsf{G}_2 : \mathbf{bad}] \leq \Pr[\mathsf{G}_3 : \mathbf{bad}] + \dfrac{q_{\mathcal{D}}}{2^{k_1}}$

Eliminate assignments to $\boldsymbol{L}_G$ in $\mathcal{D}$
Update $\mathcal{D}$ to enforce new bound on $\boldsymbol{L}_G$

```
Game $G_3$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $\hat{\boldsymbol{r}} \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \overset{\$}{\leftarrow} \{0,1\}$;
  $\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}})$;
  $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L}_G)$ then
    if $r = \hat{\boldsymbol{r}}$ then
      bad $\leftarrow$ true;
    $g \overset{\$}{\leftarrow} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G[r] \leftarrow g$
  else $g \leftarrow \boldsymbol{L}_G[r]$
  return $g$

Oracle $H(s)$ :
  if $s \notin$ dom$(\boldsymbol{L}_H)$ then
    $h \overset{\$}{\leftarrow} \{0,1\}^{\rho}$;
    $\boldsymbol{L}_H[s] \leftarrow h$
  else $h \leftarrow \boldsymbol{L}_H[s]$
  return $h$

Oracle $\mathcal{D}(c)$ :
  if $(\hat{\boldsymbol{c}}_{\mathsf{def}} \wedge \hat{\boldsymbol{c}} = c) \vee q_{\mathcal{D}} < |\boldsymbol{L}_{\mathcal{D}}| \vee q_{\mathsf{G}} < |\boldsymbol{L}_G|$ then
    return $\bot$
  else
    $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$;
    $(s,t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
    if $s \in$ dom$(\boldsymbol{L}_H)$ then
      $r \leftarrow t \oplus H(s)$;
      if $r \in$ dom$(\boldsymbol{L}_G)$ then
        $g \leftarrow \boldsymbol{L}_G[r]$;
        if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^{\mu}$ else return $\bot$
      else
        if $r = \hat{\boldsymbol{r}}$ then bad $\leftarrow$ true;
        return $\bot$
    else
      $r \leftarrow t \oplus H(s)$; return $\bot$
```

Fig. 6.4. Outline of the reduction showing the *lossy* transitions. Fragments of code that change between games are highlighted on a gray background.

$$\Pr[\mathsf{G}_3 : \mathbf{bad}] \le \Pr[\mathsf{G}_4 : \mathbf{bad}] + \frac{q_{\mathcal{D}} q_{\mathsf{G}} + q_{\mathcal{D}}}{2^\rho}$$

Inline calls to H in $\mathcal{D}$
Eliminate assignments to $\boldsymbol{L}_H$ in $\mathcal{D}$

```
Game G_4 :
  $L_G, L_H, L_D \leftarrow$ nil;
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\hat{r} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{s} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{h} \leftarrow H(\hat{s})$;
  $\hat{t} \leftarrow \hat{h} \oplus \hat{r}$;
  $\hat{c} \leftarrow f(pk, \hat{s} \parallel \hat{t})$;
  $\hat{c}_{def} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{c})$

Oracle G(r) :
  if $r \notin$ dom$(L_G)$ then
    if $r = \hat{r}$ then
      bad $\leftarrow$ true;
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $L_G[r] \leftarrow g$
  else $g \leftarrow L_G[r]$
  return $g$

Oracle H(s) :
  if $s \notin$ dom$(L_H)$ then
    $h \xleftarrow{\$} \{0,1\}^\rho$;
    $L_H[s] \leftarrow h$
  else $h \leftarrow L_H[s]$
  return $h$

Oracle D(c) :
  if $(\hat{c}_{def} \wedge \hat{c} = c) \vee q_D < |L_D| \vee q_G < |L_G|$ then
    return $\perp$
  else
    $L_D \leftarrow (\hat{c}_{def}, c) :: L_D$;
    $(s, t) \leftarrow f^{-1}(sk, c)$;
    if $s \in$ dom$(L_H)$ then
      $h \leftarrow L_H[s]$; $r \leftarrow t \oplus h$;
      if $r \in$ dom$(L_G)$ then
        $g \leftarrow L_G[r]$;
        if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^\mu$ else return $\perp$
      else
        if $r = \hat{r}$ then bad $\leftarrow$ true;
        return $\perp$
    else return $\perp$
```

$$\Pr[\mathsf{G}_4 : \mathbf{bad}] \le \Pr[\mathsf{G}_5 : \mathbf{bad}_H] + \frac{q_{\mathcal{D}} q_{\mathsf{G}} + 2q_{\mathcal{D}} + q_{\mathsf{G}}}{2^\rho}$$

Eagerly sample the value of $\hat{\boldsymbol{h}}$
Introduce $\mathbf{bad}_H$ in H
Bound $\mathbf{bad}$ in terms of $\mathbf{bad}_H$

```
Game G_5 :
  $L_G, L_H, L_D \leftarrow$ nil;
  $(pk, sk) \leftarrow \mathcal{KG}_f()$;
  $\hat{r} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{s} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{h} \xleftarrow{\$} \{0,1\}^\rho$;
  $\hat{t} \leftarrow \hat{h} \oplus \hat{r}$;
  $\hat{c} \leftarrow f(pk, \hat{s} \parallel \hat{t})$;
  $\hat{c}_{def} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{c})$

Oracle G(r) :
  if $r \notin$ dom$(L_G)$ then
    if $r = \hat{r}$ then
      bad $\leftarrow$ true;
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $L_G[r] \leftarrow g$
  else $g \leftarrow L_G[r]$
  return $g$

Oracle H(s) :
  if $s \notin$ dom$(L_H)$ then
    if $s = \hat{s}$ then
      bad_H $\leftarrow$ true;
    $h \xleftarrow{\$} \{0,1\}^\rho$;
    $L_H[s] \leftarrow h$
  else $h \leftarrow L_H[s]$
  return $h$

Oracle D(c) :
  if $(\hat{c}_{def} \wedge \hat{c} = c) \vee q_D < |L_D| \vee q_G < |L_G|$ then
    return $\perp$
  else
    $L_D \leftarrow (\hat{c}_{def}, c) :: L_D$;
    $(s, t) \leftarrow f^{-1}(sk, c)$;
    if $s \in$ dom$(L_H)$ then
      $h \leftarrow L_H[s]$; $r \leftarrow t \oplus h$;
      if $r \in$ dom$(L_G)$ then
        $g \leftarrow L_G[r]$;
        if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^\mu$ else return $\perp$
      else return $\perp$
    else return $\perp$
```

Fig. 6.4. Outline of the reduction showing the *lossy* transitions. Fragments of code that change between games are highlighted on a gray background.

The transition from G_1 to G_2 modifies the decryption oracle successively by inlining the call to G, and by applying the Fundamental and Failure Event lemmas to reject the ciphertext when there is a negligible chance it matches the padding. Overall, we prove:

$$\Pr[\mathsf{G}_1 : \mathbf{bad}] \le \Pr[\mathsf{G}_2 : \mathbf{bad}] + \frac{q_{\mathcal{D}}^2 + q_{\mathcal{D}} q_{\mathsf{G}} + q_{\mathcal{D}}}{2^\rho} + \frac{q_{\mathcal{D}}}{2^{k_1}} \qquad (6.19)$$

Next, we eliminate fresh calls to G in the decryption oracle. These calls correspond to the two assignments $\boldsymbol{L}_G[r] \leftarrow g$, since calls to G have been inlined previously. We perform an aggressive elimination and remove both calls. As a result, in game G_3 the length of list $\boldsymbol{L}_G$ (i.e. the number of calls to G) is bounded by q_{G} rather than $q_{\mathcal{D}} + q_{\mathsf{G}}$. This is the key to improve on the security bound of Pointcheval

[2005], who only removes the second call. The proof relies on the logic of swapping statements to show that values of discarded calls are "uniformly distributed and independent from the adversary's view". Details appear below. Overall, we prove:

$$\Pr[\mathsf{G}_2 : \mathbf{bad}] \leq \Pr[\mathsf{G}_3 : \mathbf{bad}] + \frac{q_{\mathcal{D}}}{2^{k_1}} \tag{6.20}$$

Likewise, we eliminate calls to H in $\mathcal{D}$, yielding a new game G_4 in which the decryption oracle does not add any new values to the memories of G and H. Using the Fundamental and Failure Event lemmas, we obtain:

$$\Pr[\mathsf{G}_3 : \mathbf{bad}] \leq \Pr[\mathsf{G}_4 : \mathbf{bad}] + \frac{q_{\mathcal{D}} q_{\mathsf{G}} + q_{\mathcal{D}}}{2^{\rho}} \tag{6.21}$$

We next fix the value $\hat{\boldsymbol{h}}$ that oracle H gives in response to $\hat{\boldsymbol{s}}$, and then make H return a freshly sampled value instead of $\hat{\boldsymbol{h}}$. This allows us to bound the probability of **bad** in terms of the probability of a newly introduced event $\mathbf{bad}_H$. The proof uses the hypothesis that $\mathcal{A}_2$ cannot query the decryption oracle with the challenge ciphertext, and yields:

$$\Pr[\mathsf{G}_4 : \mathbf{bad}] \leq \Pr[\mathsf{G}_5 : \mathbf{bad}_H] + \frac{q_{\mathcal{D}} q_{\mathsf{G}} + 2q_{\mathcal{D}} + q_{\mathsf{G}}}{2^{\rho}} \tag{6.22}$$

Finally, we prove that the probability of $\mathbf{bad}_H$ in G_5 is upper bounded by the probability that the inverter $\mathcal{I}$ succeeds in partially inverting the permutation f. The proof uses the (standard, non-relational) invariant on G_5:

$$\mathbf{bad}_H \implies \hat{\boldsymbol{s}} \in \mathsf{dom}(\boldsymbol{L}_H)$$

The inverter $\mathcal{I}$ that we build (shown in Figure 6.3) gives its own challenge y as the challenge ciphertext to the IND-CCA2 adversary $\mathcal{A}$ and returns a random element in the list of queries made to H. Thus,

$$\Pr[\mathsf{G}_5 : \mathbf{bad}_H] \leq \Pr[\mathsf{G}_5 : \hat{\boldsymbol{s}} \in \mathsf{dom}(\boldsymbol{L}_H)] \leq \frac{1}{q_{\mathsf{H}}} \Pr[\mathsf{G}_{\mathsf{PDOW}} : s = \tilde{s}] \tag{6.23}$$

Where the last inequality follows from the bound on the number of queries to oracle H and an instance of the optimistic sampling equivalence:

$$\vdash \hat{\boldsymbol{h}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho};\ \hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}} \simeq^{\{\hat{\boldsymbol{r}}\}}_{\{\hat{\boldsymbol{h}},\hat{\boldsymbol{t}},\hat{\boldsymbol{r}}\}} \hat{\boldsymbol{t}} \stackrel{\$}{\leftarrow} \{0,1\}^{\rho};\ \hat{\boldsymbol{h}} \leftarrow \hat{\boldsymbol{t}} \oplus \hat{\boldsymbol{r}}$$

Putting together (6.18)–(6.23) concludes the proof of the statement in Theorem 6.4.

Detailed proof of the transition from G_2 to G_3

We use the five intermediate games shown in Figure 6.5. The first transition from G_2 to G_2^1 consists in adding a tag to queries in the memory of G indicating whether the query has been made directly by the adversary or indirectly, through the decryption oracle. The decryption oracle tests this tag when accessing the memory of G: if the

```
Game $\dbox{\mathsf{G}_2^1}$ $\boxed{\mathsf{G}_2^2}$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}})$;
  $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L}_G)$ then
    if $r = \hat{\boldsymbol{r}}$ then
      $\mathbf{bad} \leftarrow$ true
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G[r] \leftarrow$ (false, $g$)
  else
    $(d, g) \leftarrow \boldsymbol{L}_G[r]$;
    $\boldsymbol{L}_G[r] \leftarrow$ (false, $g$)
  return $g$

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

Oracle $\mathcal{D}(c)$ :
  if $(\hat{\boldsymbol{c}}_{\mathsf{def}} \wedge \hat{\boldsymbol{c}} = c) \vee q_{\mathcal{D}} < |\boldsymbol{L}_{\mathcal{D}}| \vee q_{\mathcal{D}} + q_{\mathsf{G}} < |\boldsymbol{L}_G|$ then
    return $\perp$
  else
    $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$; $(s, t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
    if $s \in$ dom$(\boldsymbol{L}_H)$ then
      $r \leftarrow t \oplus H(s)$;
      if $r \in$ dom$(\boldsymbol{L}_G)$ then
        $(d, g) \leftarrow \boldsymbol{L}_G[r]$;
        if $d =$ true then
          if $[s \oplus g]_{k_1} = 0^{k_1}$ then
            $\mathbf{bad}_1 \leftarrow$ true; $\dbox{\text{return } [s \oplus g]^{\mu}}$ $\boxed{\text{return } \perp}$
          else return $\perp$
        else
          if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^{\mu}$
          else return $\perp$
      else
        if $r = \hat{\boldsymbol{r}}$ then $\mathbf{bad} \leftarrow$ true;
        $g \xleftarrow{\$} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow$ (true, $g$); return $\perp$
    else
      $r \leftarrow t \oplus H(s)$;
      if $r \notin$ dom$(\boldsymbol{L}_G)$ then
        $g \xleftarrow{\$} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow$ (true, $g$);
      return $\perp$
```

```
Game $\mathsf{G}_2^3$ $\dbox{\mathsf{G}_2^4}$ $\boxed{\mathsf{G}_2^5}$ :
  $\boldsymbol{L}_G, \boldsymbol{L}_H, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil;
  $(\boldsymbol{pk}, \boldsymbol{sk}) \leftarrow \mathcal{KG}_f()$;
  $(m_0, m_1) \leftarrow \mathcal{A}_1(\boldsymbol{pk})$;
  $b \xleftarrow{\$} \{0,1\}$;
  $\hat{\boldsymbol{r}} \xleftarrow{\$} \{0,1\}^{\rho}$;
  $\hat{\boldsymbol{s}} \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  $\hat{\boldsymbol{h}} \leftarrow H(\hat{\boldsymbol{s}})$;
  $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{h}} \oplus \hat{\boldsymbol{r}}$;
  $\hat{\boldsymbol{c}} \leftarrow f(\boldsymbol{pk}, \hat{\boldsymbol{s}} \parallel \hat{\boldsymbol{t}})$;
  $\hat{\boldsymbol{c}}_{\mathsf{def}} \leftarrow$ true;
  $\tilde{b} \leftarrow \mathcal{A}_2(\hat{\boldsymbol{c}})$
  ┆ $L \leftarrow \boldsymbol{L}_G$;
  ┆ while $L \neq$ nil do
  ┆   $(r, (b, g)) \leftarrow$ hd$(L)$;
  ┆   if $b =$ true then
  ┆     $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
  ┆     $\boldsymbol{L}_G[r] \leftarrow$ (true, $g$)
  ┆   $L \leftarrow$ tl$(L)$

Oracle $G(r)$ :
  if $r \notin$ dom$(\boldsymbol{L}_G)$ then
    if $r = \hat{\boldsymbol{r}}$ then
      $\mathbf{bad} \leftarrow$ true
    $g \xleftarrow{\$} \{0,1\}^{k-\rho}$;
    $\boldsymbol{L}_G[r] \leftarrow$ (false, $g$)
  else
    $(d, g) \leftarrow \boldsymbol{L}_G[r]$;
    if $d =$ true then
      $\dbox{g \xleftarrow{\$} \{0,1\}^{k-\rho};}$
      $\boxed{g \xleftarrow{\$} \{0,1\}^{k-\rho};}$
      $\boldsymbol{L}_G[r] \leftarrow$ (false, $g$);
      $\mathbf{bad}_2 \leftarrow P(g, r)$
  return $g$

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

Oracle $\mathcal{D}(c)$ :
  if $(\hat{\boldsymbol{c}}_{\mathsf{def}} \wedge \hat{\boldsymbol{c}} = c) \vee q_{\mathcal{D}} < |\boldsymbol{L}_{\mathcal{D}}| \vee q_{\mathcal{D}} + q_{\mathsf{G}} < |\boldsymbol{L}_G|$ then
    return $\perp$
  else
    $\boldsymbol{L}_{\mathcal{D}} \leftarrow (\hat{\boldsymbol{c}}_{\mathsf{def}}, c) :: \boldsymbol{L}_{\mathcal{D}}$; $(s, t) \leftarrow f^{-1}(\boldsymbol{sk}, c)$;
    if $s \in$ dom$(\boldsymbol{L}_H)$ then
      $r \leftarrow t \oplus H(s)$;
      if $r \in$ dom$(\boldsymbol{L}_G)$ then
        $(d, g) \leftarrow \boldsymbol{L}_G[r]$;
        if $d =$ true then return $\perp$
        else
          if $[s \oplus g]_{k_1} = 0^{k_1}$ then return $[s \oplus g]^{\mu}$
          else return $\perp$
      else
        if $r = \hat{\boldsymbol{r}}$ then $\mathbf{bad} \leftarrow$ true;
        $g \xleftarrow{\$} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow$ (true, $g$); return $\perp$
    else
      $r \leftarrow t \oplus H(s)$;
      if $r \notin$ dom$(\boldsymbol{L}_G)$ then
        $g \xleftarrow{\$} \{0,1\}^{k-\rho}$; $\boldsymbol{L}_G[r] \leftarrow$ (true, $g$);
      return $\perp$
```

$P(g, r) \stackrel{\text{def}}{=} \exists (d, c) \in \boldsymbol{L}_{\mathcal{D}}.$ let $(s, t) = f^{-1}(\boldsymbol{sk}, c)$ in $s \in \mathsf{dom}(\boldsymbol{L}_H) \wedge r = t \oplus \boldsymbol{L}_H[s] \wedge [s \oplus g]_{k_1} = 0^{k_1}$

Fig. 6.5. Games in the transition from G_2 to G_3. Fragments of code inside a box appear only in the game whose name is surrounded by the matching box.

ciphertext queried is valid and its random seed appeared in a previous decryption query, but not yet in a direct query to G, the decryption oracle raises a flag $\mathbf{bad}_1$. We show that this can happen with probability 2^{-k_1} for any single query, since the random seed is uniformly distributed and independent from the adversary's view. In this case, the decryption oracle can safely reject the ciphertext, as done in game G_2^2. The proof proceeds in two steps: We show that game G_2 is observationally equivalent to game G_2^1 using the relational invariant

$$\boldsymbol{L}_G\langle 1\rangle = (\mathsf{map}\ (\lambda(r,(b,g)).(r,g))\ \boldsymbol{L}_G)\langle 2\rangle$$

Therefore $\Pr\left[\mathsf{G}_2 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_2^1 : \mathbf{bad}\right]$. Game G_2^2 is identical to G_2^1, except that it rejects ciphertexts that raise the $\mathbf{bad}_1$ flag. Applying the Fundamental Lemma (i.e. Lemma 3.3), we show that

$$\Pr\left[\mathsf{G}_2^1 : \mathbf{bad}\right] \le \Pr\left[\mathsf{G}_2^2 : \mathbf{bad}\right] + \Pr\left[\mathsf{G}_2^2 : \mathbf{bad}_1\right]$$

Our next goal is to show that answers to queries tagged as true can be resampled. However, one cannot apply the logic of swapping statements at this stage to resample these answers in G, because flag $\mathbf{bad}_1$ is set on $\mathcal{D}$ and depends on them. The solution is to introduce a new game G_2^3 that sets another flag $\mathbf{bad}_2$ in the code of G instead of setting $\mathbf{bad}_1$ in the decryption oracle[2]. Flag $\mathbf{bad}_2$ is raised whenever the adversary queries G with the random seed of a valid ciphertext previously submitted to the decryption oracle. We prove that games G_2^2 and G_2^3 satisfy the relational invariant:

$$\mathbf{bad}_1\langle 1\rangle \implies (\mathbf{bad}_2 \vee \phi)\langle 2\rangle$$

where

$$\begin{aligned}\phi \stackrel{\text{def}}{=}\ & \exists (d,c) \in \boldsymbol{L}_{\mathcal{D}}.\ \mathsf{let}\ (s,t) = f^{-1}(\boldsymbol{sk},c),\ r = t \oplus \boldsymbol{L}_H[s]\ \mathsf{in} \\ & r \in \mathsf{dom}(\boldsymbol{L}_G) \wedge s \in \mathsf{dom}(\boldsymbol{L}_H) \wedge \mathsf{fst}(\boldsymbol{L}_G[r]) = \mathsf{false} \wedge [s \oplus \mathsf{snd}(\boldsymbol{L}_G[r])]_{k_1} = 0^{k_1}\end{aligned}$$

Therefore:

$$\Pr\left[\mathsf{G}_2^2 : \mathbf{bad}\right] + \Pr\left[\mathsf{G}_2^2 : \mathbf{bad}_1\right] \le \Pr\left[\mathsf{G}_2^3 : \mathbf{bad}\right] + \Pr\left[\mathsf{G}_2^3 : \mathbf{bad}_2 \vee \phi\right]$$

We now consider game G_2^4 where oracle G resamples the answers to queries previously sampled in the decryption oracle. As such answers are uniformly distributed and independent from the adversary's view, the logic for swapping statements can be used to establish that this transformation preserves semantics. Hence:

$$\Pr\left[\mathsf{G}_2^3 : \mathbf{bad}\right] + \Pr\left[\mathsf{G}_2^3 : \mathbf{bad}_2 \vee \phi\right] = \Pr\left[\mathsf{G}_2^4 : \mathbf{bad}\right] + \Pr\left[\mathsf{G}_2^4 : \mathbf{bad}_2 \vee \phi\right]$$

Note that in order to prove semantic equivalence we need to resample the values in $\boldsymbol{L}_G$ associated to queries tagged as true—made by the $\mathcal{D}$—at the end of the game. Using Lemma 3.4, we upper bound the probability of $\mathbf{bad}_2 \vee \phi$ in G_2^4:

[2] As $\mathbf{bad}_1$ is not set anymore, we simplify the code of $\mathcal{D}$ by coalescing branches in the innermost conditional.

$$\Pr\left[\mathsf{G}_2^4 : \mathbf{bad}_2 \vee \phi\right] \leq \frac{q_{\mathcal{D}}}{2^{k_1}}$$

We are now only interested in bounding **bad**, so we can remove as dead code the fragment of code at the end of G_2^4 that resamples values in $\boldsymbol{L_G}$, obtaining G_2^5, and prove that

$$\Pr\left[\mathsf{G}_2^4 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_2^5 : \mathbf{bad}\right]$$

We finally prove that game G_2^5 is observationally equivalent to G_3, in which the code for the oracle G is reverted to its original form and the decryption oracle no longer tampers with the memory of G. Thus,

$$\Pr\left[\mathsf{G}_2 : \mathbf{bad}\right] \leq \Pr\left[\mathsf{G}_2^5 : \mathbf{bad}\right] + \frac{q_{\mathcal{D}}}{2^{k_1}} = \Pr\left[\mathsf{G}_3 : \mathbf{bad}\right] + \frac{q_{\mathcal{D}}}{2^{k_1}}$$

□

Comparison with the security bound of Pointcheval [2005]

Pointcheval obtains a slightly different bound:

$$\epsilon' \geq \frac{1}{q_{\mathsf{H}}}\left(\epsilon - \frac{4q_{\mathcal{D}}q_{\mathsf{G}} + 2q_{\mathcal{D}}^2 + 4q_{\mathcal{D}} + 8q_{\mathsf{G}}}{2^{\rho}} - \frac{3q_{\mathcal{D}}}{2^{k_1}}\right)$$

We marginally improve on this bound by reducing the coefficients. As previously mentioned, the improvement stems from the transition from G_2 to G_3, where we eliminate both calls to G in the decryption oracle, whereas only one of them is eliminated in [Pointcheval 2005]. In fact, eliminating both calls is not only needed to give a better bound, but is essential for the correctness of the proof. Indeed, the transition from G_3 to G_4 would not be possible if $\mathcal{D}$ modified the memory of G. Concretely, the justification of Equation (27) in [Pointcheval 2005] contains two minor glitches: firstly, the remark "which just cancels r' from $\boldsymbol{L}_G$" oversees the possibility of this removal having an impact on future queries. Secondly, "the probability for r' to be in $\boldsymbol{L}_G$ is less than $q_{\mathsf{G}}/2^{\rho}$" oversees that the length of $\boldsymbol{L}_G$ is upper bounded by $q_{\mathsf{G}} + q_{\mathcal{D}}$ rather than just q_{G}, as the decryption oracle still adds values to $\boldsymbol{L}_G$; a correct bound for this probability in [Pointcheval 2005] is $(q_{\mathsf{G}} + q_{\mathcal{D}})/2^{\rho}$.

6.2.2 Notes about the Proved Security Bound

We note that although we exhibit an explicit inverter that achieves the advantage in the statement of Theorem 6.4, we do not prove formally that it executes within the given time bound. We would be loath to say that the proof is incomplete, because the time complexity of the inverter is evident from its formulation. Nonetheless, we can prove that the inverter executes in probabilistic polynomial-time (under a reasonable cost model for constructions in the language) and thus the asymptotic security of OAEP under the hypothesis that the underlying permutation family is partial-domain one-way. Formally, if the winning probability $\Pr\left[\mathsf{G}_{\mathsf{PDOW}} : s = \tilde{s}\right]$ of any PPT inverter is negligible, we can prove that the advantage of any IND-CCA2 adversary is also negligible, provided k_1 and η increase at least linearly with the security parameter, i.e.,

$$\exists\alpha.\ \exists\eta_0.\ \forall\eta \geq \eta_0.\ \rho \geq \alpha\ \eta\ \wedge\ k_1 \geq \alpha\ \eta$$

Moreover, we observe that by using a standard time-space trade-off argument (cf. [Pointcheval 2005]) one can reduce the factor $q_{\mathsf{G}} q_{\mathsf{H}} q_{\mathcal{D}}$ in the time bound of the inverter to $q_{\mathsf{G}} q_{\mathsf{H}}$.

6.3 About the Security of OAEP in the Standard Model

From a practical point of view, in reductionist arguments like the ones we studied in this chapter, the tightness of the security bound makes a whole world of difference. The tighter the bound is, the closer the problem of breaking the security of the scheme is to the problem of solving the computational hard problem under consideration. Another aspect of practical importance is the model where the proof is carried out. A proof in the random oracle model, like the proofs we presented, only rules out *generic attacks* that do not exploit the implementation details of the hash functions G and H. In contrast, a proof in the standard model of cryptography considers the possibility that an adversary might be able to exploit weaknesses in the actual implementation of the hash functions to attack the security of the scheme.

Several authors have studied the possibility of proving OAEP IND-CCA2 secure in the standard model, either when used in conjunction with the RSA function or with a trapdoor permutation satisfying particular classes of properties. Shoup [2001] showed the impossibility of finding a blackbox reduction from the IND-CCA2 security of OAEP to the one-wayness of the underlying trapdoor permutation, either in the standard or random oracle model, but he exhibited a blackbox reduction in the random oracle model to the problem of partially inverting the permutation. Brown [2006] showed a pathological instantiation of the hash functions that would render OAEP insecure, which means that a security proof should at least assume some property about the hash functions. He ruled out as well the possibility of proving the IND-CCA2 security of OAEP in the standard model using certain types of reductions. Finally, Kiltz and Pietrzak [2009] prove a blackbox separation result: no padding-based encryption scheme can be proved IND-CCA2 secure in the standard model, even assuming the underlying trapdoor permutation is *ideal*. This means in particular, that is impossible to prove the IND-CCA2 security of OAEP in the standard model under most standard assumptions about the trapdoor permutation, including one-wayness, partial-domain one-wayness, or claw-freeness.

In contrast, Kiltz et al. [2010] recently proved the IND-CPA security of generic OAEP in the standard model under non-interactive and non-interdependent assumptions on the underlying trapdoor permutation and the hash functions. In particular, they prove that RSA-OAEP is IND-CPA secure when instantiated using a t-wise independent hash function (for an appopriate t) provided RSA is a *lossy* trapdoor permutation. This latter condition can be proved under the phi-hiding assumption, a number-theoretic assumption slightly stronger than the RSA assumption.

Despite the above negative results, most people agree that a security proof of a scheme in the random oracle model gives strong evidence about its practical security, and that it is very unlikely that someone comes out with an attack exploiting details of the concrete instantiations of random oracles.

7 Machine-Checked Formalization of Zero-Knowledge Protocols

PROOFS of knowledge [Goldreich 2002; Goldwasser et al. 1989] are two-party interactive protocols where one party, called the *prover*, convinces the other one, called the *verifier*, that it knows *something.* Typically, both parties share a common input x and *something* refers to a witness w of membership of the input x to an $\mathcal{NP}$ language. Proofs of knowledge are useful to enforce honest behavior of potentially malicious parties [Backes et al. 2009]: the knowledge witness acts as an authentication token used to establish that the prover is a legitimate user of a service provided by the verifier, or as evidence that a message sent by the prover has been generated in accordance to the rules of a protocol. Proofs of knowledge must be complete, so that a prover that has indeed knowledge of a witness can convince a honest verifier, and sound, so that a dishonest prover has little chance of being convincing. In addition, practical applications often require to preserve secrecy or anonymity; in such cases the proof should not leak anything about the witness. Zero-knowledge proofs are computationally convincing proofs of knowledge that achieve this goal, i.e. they are convincing and yet the verifier does not learn anything from interacting with the prover beyond the fact that the prover knows a witness for their common input. This property has an elegant formulation: a protocol is said to be zero-knowledge when transcripts of protocol runs between a prover P and a (possibly dishonest) verifier V can be efficiently simulated without ever interacting with the prover—but with access to the strategy of V. In particular, this implies that proofs are not transferable; a conversation is convincing only for the verifier interacting with the prover and cannot be replayed to convince a third party.

In his PhD dissertation, Cramer [1996] introduced the concept of Σ-protocols , a class of three-move interactive protocols that are suitable as a basis for the design of many efficient and secure cryptographic services. Cramer described Σ-protocols as abstract modules and showed that they are realizable when instantiated for most computational assumptions commonly considered in cryptography, including the difficulty of computing discrete logarithms or factoring integers, or the existence of some abstract function families (e.g. one-way group homomorphisms). In addition, he gave an effective method to combine Σ-protocols to obtain zero-knowledge proofs of any Boolean formula constructed using the AND and OR operators from formulæ for which Σ-protocols exist. This means that Σ-protocols can be used in a practical setting as building blocks to achieve various cryptographic goals. Applications of Σ-protocols notably include secure multi-party computation, identification schemes, secret-ballot electronic voting, and anonymous attestation credentials.

This chapter reports on a fully machine-checked formalization of a comprehensive theory of Σ-protocols using CertiCrypt. The formalization consists of more than 10,000 lines of Coq code, and covers the basics of Σ-protocols: definitions, relations between different notions of security, general constructions and composability theorems. We show its applicability by formalizing several well-known protocols, including the Schnorr, Guillou-Quisquater, Okamoto, and Feige-Fiat-Shamir protocols. The highlight of the formalization is a generic account of Σ^ϕ-protocols, that prove knowledge of a preimage under a group homomorphism ϕ. We use the module system of Coq to define and relate the classes of Σ^ϕ- and Σ-protocols. Our formalization of Σ^ϕ-protocols provides sufficient conditions (the so-called specialness conditions) on the group homomorphism ϕ so that every Σ^ϕ-protocol can be construed as a Σ-protocol. Moreover, we show that special homomorphisms are closed under direct product, which yields a cheap way of AND-combining Σ^ϕ proofs. We exploit the generality of our result to achieve short proofs of completeness, special soundness, and (honest verifier) zero-knowledge for many protocols in the literature.

7.1 Sigma-Protocols

A Σ-protocol is a 3-step interactive protocol where a prover P interacts with a verifier V. Both parties have access to a common input x, and the goal of the prover is to convince the verifier that it knows some value w suitably related to x, without revealing anything beyond this assertion. The protocol begins with the prover sending a commitment r to the verifier, who responds by sending back a random challenge c; the prover then computes a response s to the challenge and sends it to the verifier, who either accepts or rejects the conversation. Figure 7.1 shows a diagram of a run of a Σ-protocol.

Formally, a Σ-protocol is defined with respect to a knowledge relation R. This terminology comes from interpreting the proof system as proving membership of the common input to an $\mathcal{NP}$ language L. Each $\mathcal{NP}$ language admits an efficient membership verification procedure via a polynomial-time recognizable relation R_L such that

$$L = \{x \mid \exists w.\ R_L(x, w)\}$$

Proving that x belongs to the language amounts to proving knowledge of a witness w related to x via R_L. In CertiCrypt, the class of Σ-protocols is formalized as a

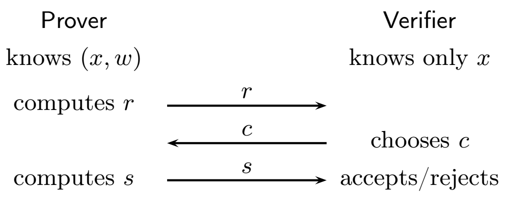


Fig. 7.1. Characteristic 3-step interaction in a run of a Σ-protocol.

module type parametrized over a knowledge relation R , and a number of procedures specifying the different phases of the prover and the verifier; the module type specifies as well the properties that any given protocol instance must satisfy. In the remainder of this section we describe in detail our formal definition of Σ-protocols and comment on an alternative—but in some sense equivalent—specification of the zero-knowledge property.

Definition 7.1 (Σ-protocol). *A Σ-protocol for a knowledge relation R is a 3-step protocol between a prover P and a verifier V, whose interaction is described by the following parametrized program:*

```
**Protocol**$(x, w)$ :
  $(r, state) \leftarrow \mathsf{P}_1(x, w)$;
  $c \leftarrow \mathsf{V}_1(x, r)$;
  $s \leftarrow \mathsf{P}_2(x, w, state, c)$;
  $b \leftarrow \mathsf{V}_2(x, r, c, s)$
```

In the above program specifying a Σ-protocol, the two phases of the prover are described by the procedures P_1 and P_2, while the phases of the verifier are described by V_1 and V_2. Note that the protocol explicitly passes state between the phases of the participants; we could have used instead global variables shared between P_1 and P_2 on one hand, and V_1 and V_2 on the other, but that would unnecessarily complicate the proofs because we would need to specify that the procedures representing one party do not have access to the shared state of the other party. All the protocols that we consider in the following are public-coin, *meaning that a honest verifier chooses the challenge uniformly from some predefined set C. A Σ-protocol must satisfy the following three properties,*

1. *Completeness: Given a public input x together with a witness w such that $R(x, w)$, the prover is always able to convince the verifier:*

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

2. *Special Honest Verifier Zero-Knowledge (sHVZK): There exists a probabilistic polynomial-time simulator S that given $x \in \mathsf{dom}(R)$ and a challenge c, computes triples (r, c, s) with the same distribution as a valid conversation. The property is formalized in terms of a version of the protocol where the challenge c is fixed as a parameter,*

$$\boxed{\begin{array}{l}\textbf{Protocol}(x,w,c):\\ \quad (r, state) \leftarrow \mathsf{P}_1(x,w);\\ \quad s \leftarrow \mathsf{P}_2(x,w,state,c);\\ \quad b \leftarrow \mathsf{V}_2(x,r,c,s)\end{array}} \simeq^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} (r,s) \leftarrow \mathsf{S}(x,c)$$

3. *Special Soundness: Given two accepting conversations (r, c_1, s_1), (r, c_2, s_2) for an input x, with the same commitment r, but with different challenges, there exists a PPT knowledge extractor* KE *that computes a witness w such that $R(x, w)$. Formally, for any memory m,*

$$\left.\begin{array}{l} m(c_1) \neq m(c_2)\\ \Pr\left[b \leftarrow \mathsf{V}_2(x,r,c_1,s_1), m : b = \mathsf{true}\right] = 1\\ \Pr\left[b \leftarrow \mathsf{V}_2(x,r,c_2,s_2), m : b = \mathsf{true}\right] = 1\end{array}\right\} \implies$$
$$\Pr\left[w \leftarrow \mathsf{KE}(x,r,c_1,c_2,s_1,s_2), m : R(x,w)\right] = 1$$

Special soundness might seem a relatively weak property at first sight. It can be shown using a rewinding argument (although we did not formalize this result in Coq) that thanks to special soundness, any public-coin Σ-protocol with challenge set C can be seen as a *proof of knowledge* with *soundness error* $|C|^{-1}$ [Damgård and Pfitzmann 1998]. Informally, this means that any efficient prover (possibly dishonest) that manages to convince a honest verifier for a public input x with a probability greater than $|C|^{-1}$ can be turned into an efficient procedure that computes a witness for x.

7.1.1 Relation between sHVZK and HVZK

Some authors require that Σ-protocols satisfy a somewhat weaker property known as honest verifier zero-knowledge rather than the *special* version of this property mentioned above. The difference is that in the former the simulator is allowed to choose the challenge while in the latter the challenge is fixed. In other words, plain HVZK requires that there exists a PPT simulator S that given just $x \in \mathsf{dom}(R)$ computes a triple (r, c, s) with the same distribution as the verifier's view of a conversation. The relation between the two notions has been studied by Cramer [1996]. As an illustration of the use of CertiCrypt and the Σ-protocol framework, the formalization of this relation is discussed below. Without loss of generality we assume that the challenge set of the protocols we consider is $\{0, 1\}^k$.

Theorem 7.2 (sHVZK implies HVZK). *If a Σ-protocol satisfies sHVZK, it also satisfies HVZK.*

Proof. A HVZK simulator S′ can be built from the sHVZK simulator S:

$$\textbf{Simulator } \mathsf{S}'(x):\ c \xleftarrow{\$} \{0,1\}^k;\ (r,s) \leftarrow \mathsf{S}(x,c);\ \mathsf{return}\ (r,c,s)$$

The fact that S′ perfectly simulates conversations of the protocol can be proved by means of the following sequence of games:

$$\begin{aligned}
\mathsf{Protocol}(x,w) &\simeq^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} c \xleftarrow{\$} \{0,1\}^k;\ \mathsf{Protocol}(x,w,c)\\
&\simeq^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} c \xleftarrow{\$} \{0,1\}^k;\ (r,s) \leftarrow \mathsf{S}(x,c)\\
&\simeq^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} (r,c,s) \leftarrow \mathsf{S}'(x)
\end{aligned}$$

The first and last equivalences are easily proved by inlining procedure calls using the tactic `inline`, and reordering instructions in the resulting programs using `swap`. To prove the second equivalence, the tactic `eqobs_hd` is used to get rid of the instruction $c \xleftarrow{\$} \{0,1\}^k$ that is common to both games; the resulting goal matches exactly the definition of sHVZK for S. □

In a sense, sHVZK is a stronger property than HVZK, because a protocol satisfying sHVZK can be shown to satisfy HVZK, while the converse is not generally true. However, from every protocol (P,V) that satisfies HVZK it is possible to construct a protocol $(\mathsf{P}',\mathsf{V}')$ that satisfies sHVZK and is nearly as efficient as the original protocol:

$$\begin{aligned}
\mathsf{P}'_1(x,w) &\overset{\text{def}}{=} (r, \mathit{state}) \leftarrow \mathsf{P}_1(x,w);\ c' \xleftarrow{\$} \{0,1\}^k;\ \mathsf{return}\ ((r,c'),(\mathit{state},c')))\\
\mathsf{P}'_2(x,w,(\mathit{state},c'),c) &\overset{\text{def}}{=} s \leftarrow \mathsf{P}_2(x,w,\mathit{state},c\oplus c');\ \mathsf{return}\ s\\
\mathsf{V}'_1(x,(r,c')) &\overset{\text{def}}{=} c \leftarrow \mathsf{V}_1(x,r);\ \mathsf{return}\ (c\oplus c')\\
\mathsf{V}'_2(x,(r,c'),c,s) &\overset{\text{def}}{=} b \leftarrow \mathsf{V}_2(x,r,c\oplus c',s);\ \mathsf{return}\ b
\end{aligned}$$

Essentially, the construction creates a new protocol for which HVZK and sHVZK coincide. The difference is that in the new protocol the challenge that the verifier chooses is xor-ed with a random bitstring sampled by the prover at the beginning of the protocol.

Theorem 7.3 (sHVZK from HVZK). *If a protocol (P,V) is a Σ-protocol as in Definition 7.1 but satisfying HVZK instead of sHVZK, then the protocol $(\mathsf{P}',\mathsf{V}')$ defined above is a Σ-protocol.*

Proof.

Completeness

Follows easily from the completeness of protocol (P,V) and the absorption property of the exclusive or operator, i.e., $(c\oplus c')\oplus c' = c$.

Special Honest Verifier Zero-Knowledge

The following is a sHVZK simulator for the protocol

$$\textbf{Simulator}\ \mathsf{S}'(x,c):\ (\hat{r},\hat{c},\hat{s}) \leftarrow \mathsf{S}(x);\ \mathsf{return}\ ((\hat{r}, c\oplus\hat{c}),\hat{s})$$

(The variables of the original protocol are decorated with a hat.) We prove this by means of a sequence of program equivalences,

$$\begin{aligned}
\mathsf{Protocol}'(x,w,c) &\simeq^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} \mathsf{Protocol}(x,w);\ r \leftarrow (\hat{r}, c\oplus\hat{c});\ s \leftarrow \hat{s}\\
&\simeq^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} (\hat{r},\hat{c},s) \leftarrow \mathsf{S}(x);\ r \leftarrow (\hat{r}, c\oplus\hat{c})\\
&\simeq^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} (r,s) \leftarrow \mathsf{S}'(x,c)
\end{aligned}$$

The first and last equivalences are proved without much difficulty using the program transformation tactics described in Chapter 3, while the second can be reduced to the HVZK of S using the `alloc` and `eqobs_tl` tactics to simplify the goal.

Soundness

From a conversation $((r,c'),(c\oplus c'),s)$ of $(\mathsf{P}',\mathsf{V}')$ a conversation (r,c,s) of the original protocol can be trivially recovered. Thus, the following knowledge extractor proves special soundness of $(\mathsf{P}',\mathsf{V}')$:

$$\mathsf{KE}'(x,(r,c'),c_1,c_2,s_1,s_2):\ w \leftarrow \mathsf{KE}(x,r,c'\oplus c_1,c'\oplus c_2,s_1,s_2);\ \mathsf{return}\ w$$

□

7.2 Sigma Protocols Based on Special Homomorphisms

An important class of Σ-protocols are the so-called Σ^ϕ-protocols, that prove knowledge of a preimage under a homomorphism. The Schnorr protocol [Schnorr 1991], one of the most archetypal zero-knowledge proofs, is an instance of a Σ^ϕ-protocol that proves knowledge of a discrete logarithm in a cyclic group, i.e. the homomorphism is in this case exponentiation, $\phi(x) = g^x$, where g is a generator of the group.

Our formalization of Σ^ϕ-protocols is constructive. We provide a functor that, given a homomorphism ϕ together with proofs that it satisfies certain properties, builds a concrete Σ-protocol for proving knowledge of a preimage under ϕ. This protocol comes with proofs of completeness, soundness, and sHVZK. Thus, all that it takes to build an instance of a Σ^ϕ-protocol is to specify a homomorphism and prove that it has the necessary properties. In this way, we give several examples of Σ^ϕ-protocols, including the Schnorr, Guillou-Quisquater and Feige-Fiat-Shamir protocols. Although using the Σ^ϕ construction spares us the hassle of proving each time the properties in Definition 7.1, these instantiations remain non-trivial because one needs to formalize the homomorphisms themselves, which in turn requires to give representations of the groups over which they are defined.

In the remaining of this subsection we let $(G,\oplus)$ be a finite additive group and $(H,\otimes)$ a multiplicative group.

Definition 7.4 (Σ^ϕ-protocol). *Let $\phi : G \to H$ be a homomorphism, and define $R \stackrel{def}{=} \{(x,w) \mid x = \phi(w)\}$. The Σ^ϕ-protocol for relation R with challenge set $C \subset \mathbb{N}$ is the Σ-protocol (P,V) defined as follows:*

$$
\begin{array}{lcl}
\mathsf{P}_1(x,w) & \stackrel{def}{=} & y \stackrel{\$}{\leftarrow} G;\ \mathsf{return}\ (\phi(y), y) \\
\mathsf{P}_2(x,w,y,c) & \stackrel{def}{=} & \mathsf{return}\ (y \oplus cw) \\
\mathsf{V}_1(x,r) & \stackrel{def}{=} & c \stackrel{\$}{\leftarrow} C;\ \mathsf{return}\ c \\
\mathsf{V}_2(x,r,c,s) & \stackrel{def}{=} & \mathsf{return}\ (\phi(s) = r \otimes x^c)
\end{array}
$$

It can be shown that the above protocol satisfies the properties of a Σ-protocol when $C = \{0,1\}$. However, a cheating prover could convince the verifier with probability $1/2$; this probability may be reduced to $1/2^n$ (at the cost of efficiency) by repeating the protocol n rounds. We will see that a certain class of homomorphisms defined below admits a much larger challenge set, and thus achieves a lower soundness error in a single execution of the protocol.

Definition 7.5 (Special homomorphism). *An homomorphism $\phi : G \to H$ is special if there exists a value $v \in \mathbb{Z} \setminus \{0\}$ (called* special exponent*) and a PPT algorithm that given $x \in H$ computes $u \in G$ such that $\phi(u) = x^v$.*

To formalize Σ^ϕ-protocols, we extended the language of CertiCrypt with types for the groups G, H and operators for computing the group operation, exponentiation/product, and inverse; we also added operators $\phi(\cdot)$, $u(\cdot)$, and a constant expression v denoting the special exponent of the homomorphism as in Definition 7.5. In addition, we wrote an expression normalizer that simplifies arithmetic expressions by applying the homomorphic property of ϕ; normalization is done as part of the `ep` tactic.

A Σ^ϕ-protocol built from a special homomorphism admits as a challenge set any natural interval of the form $[0..c^+]$, where c^+ is smaller than the smallest prime divisor of the special exponent v.

Theorem 7.6 (Σ^ϕ-protocols for special homomorphisms). *If ϕ is special and c^+ is smaller than any prime divisor of the special exponent v, then the protocol in Definition 7.4 is a Σ-protocol with challenge set $C = [0..c^+]$.*

Proof.

Completeness

We must prove that a honest prover always succeeds in convincing a verifier, i.e.

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

Note that this can be reformulated in terms of a program equivalence as follows

$$\mathsf{Protocol}(x,w) \simeq_{\{b\}}^{R(x,w)} b \leftarrow \mathsf{true}$$

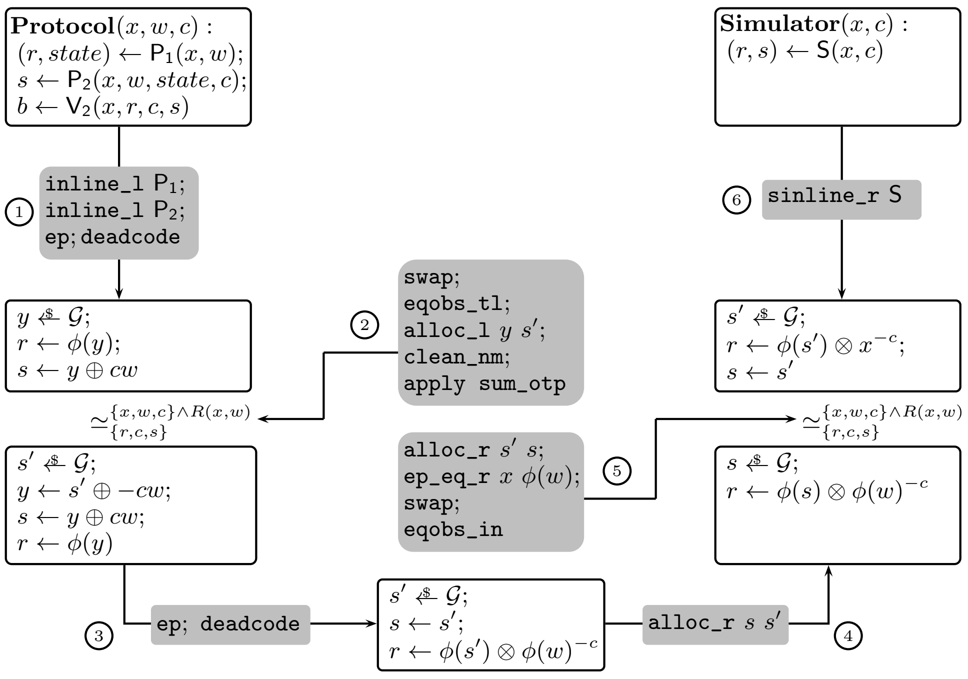
To prove this, we inline all procedure calls in the protocol and simplify the resulting program performing expression propagation, normalization, and dead code elimination. We use the following proof script:

```
inline P₁; inline P₂; inline V₁; inline V₂; ep; deadcode.
```

The resulting goal has the form

$$y \xleftarrow{\$} G;\ c \xleftarrow{\$} [0..c^{+}];\ b \leftarrow (\phi(y) \otimes \phi(w)^{c} = \phi(y) \otimes x^{c}) \quad \simeq_{\{b\}}^{\phi(w)=x} \quad b \leftarrow \mathsf{true}$$

We use the tactic `ep_eq` $x\ \phi(w)$ to simplify the last instruction in the game on the left hand side to $b \leftarrow \mathsf{true}$, tactic `deadcode` to remove the first two instructions that are no longer relevant, and `eqobs_in` to conclude.



Protocol(x, w, c) :
$(r, state) \leftarrow \mathsf{P}_1(x, w)$;
$s \leftarrow \mathsf{P}_2(x, w, state, c)$;
$b \leftarrow \mathsf{V}_2(x, r, c, s)$
Simulator(x, c) :
$(r, s) \leftarrow \mathsf{S}(x, c)$
1
inline_l P_1; inline_l P_2; ep; deadcode
6
sinline_r S
$y \xleftarrow{\$} \mathcal{G}$; $r \leftarrow \phi(y)$; $s \leftarrow y \oplus cw$
2
swap; eqobs_tl; alloc_l y s'; clean_nm; apply sum_otp
$s' \xleftarrow{\$} \mathcal{G}$; $r \leftarrow \phi(s') \otimes x^{-c}$; $s \leftarrow s'$
$\simeq_{\{r,c,s\}}^{\{x,w,c\} \wedge R(x,w)}$
$\simeq_{\{r,c,s\}}^{\{x,w,c\} \wedge R(x,w)}$
$s' \xleftarrow{\$} \mathcal{G}$; $y \leftarrow s' \oplus -cw$; $s \leftarrow y \oplus cw$; $r \leftarrow \phi(y)$
alloc_r s' s; ep_eq_r x $\phi(w)$; swap; eqobs_in
5
$s \xleftarrow{\$} \mathcal{G}$; $r \leftarrow \phi(s) \otimes \phi(w)^{-c}$
3
ep; deadcode
$s' \xleftarrow{\$} \mathcal{G}$; $s \leftarrow s'$; $r \leftarrow \phi(s') \otimes \phi(w)^{-c}$
alloc_r s s'
4

Fig. 7.2. A game-based proof that S is a sHVZK simulator for the Σ^{ϕ}-protocol in Theorem 7.6.

Special Honest Verifier Zero-Knowledge

The following is a sHVZK simulator for the protocol:

$$\textbf{Simulator } \mathsf{S}(x, c) :\ s \xleftarrow{\$} G;\ r \leftarrow \phi(s) \otimes x^{-c};\ \mathsf{return}\ (r, c, s)$$

A proof that S perfectly simulates conversations of the protocol is illustrated in Figure 7.2; we briefly explain the numbered steps in the figure.

1. Similarly to the proof above, we inline calls to P_1 and P_2, and simplify the goal using tactics `ep` and `deadcode`.

2. We introduce an intermediate game using the transitivity of observational equivalence. To prove that the new game is observationally equivalent to the previous one, we first reorder the instructions using `swap` to obtain a common suffix which we then remove using the tactic `eqobs_tl`. The resulting goal is

$$\vdash y \xleftarrow{\$} G \simeq^{\{x,w,c\} \wedge R(x,w)}_{\{y,w,c\}} s' \xleftarrow{\$} G;\ y \leftarrow s' \oplus -cw$$

   Since variables w and c are not modified, we can remove them from the output set using tactic `clean_nm`. We next use tactic `alloc` y s' to sample s' instead of y in the game on the left, and we weaken the pre-condition to `true`, which results in the goal

$$\vdash s' \xleftarrow{\$} G;\ y \leftarrow s' \simeq_{\{y\}} s' \xleftarrow{\$} G;\ y \leftarrow s' \oplus -cw$$

   This equivalence holds because $-cw$ acts as a one-time pad; we have proved this as a lemma called `sum_otp` that we apply to conclude the proof.
3. Using `ep`, we propagate throughout the code the value assigned to y and then remove the assignment using `deadcode`. The expression normalizer automatically simplifies $(s' \oplus -cw) \oplus cw$ to s', and $\phi(s' \oplus -cw)$ to $\phi(s') \otimes \phi(w)^{-c}$ using the homomorphic property of ϕ.
4. We introduce a new intermediate game; to prove that is equivalent to the previous one, we allocate variable s into s'; the resulting game is identical to the one on the left hand side.
5. We substitute variable s for s' in the game on the right hand side of the equivalence, and use the pre-condition $R(x, w)$—which boils down to $x = \phi(w)$—to substitute x by $\phi(w)$. The resulting games are identical modulo reordering of instructions.
6. We conclude by inlining the call to S in the simulation.

Soundness

Soundness requires the existence of an algorithm KE that given two accepting conversations (x, r, c_1, s_1), (x, r, c_2, s_2), with $c_1 \neq c_2$, efficiently computes a w such that $x = \phi(w)$. We propose the following knowledge extractor:

```
KE($x, c_1, c_2, s_1, s_2$) :
  $(a, b, d) \leftarrow$ extended_gcd($c_1 - c_2, v$);
  $w \leftarrow a(s_1 \oplus -s_2) \oplus b\ u(x)$;
  return $w$
```

where extended_gcd efficiently implements the extended Euclidean algorithm. For integers a, b, extended_gcd(a, b) computes a triple of integers (x, y, d) such that d is the greatest common divisor of a and b, and x, y satisfy the Bézout's identity

$$ax + by = \gcd(a, b) = d$$

Since all computations done by the knowledge extractor can be efficiently implemented, KE is a PPT algorithm. We have to prove as well that KE computes a preimage of the public input x. For two accepting conversations (x, r, c_1, s_1) and (x, r, c_2, s_2), we have

$$\phi(s_1) = r \otimes x^{c_1} \ \wedge \ \phi(s_2) = r \otimes x^{c_2}$$

and thus

$$x^{c_1 - c_2} = \phi(s_1 \oplus -s_2) \tag{7.1}$$

Furthermore, since ϕ is special we can efficiently compute u such that $x^v = \phi(u)$. The triple (a, b, d) given by the extended Euclidean algorithm satisfies the Bézout's identity

$$a(c_1 - c_2) + bv = \gcd(c_1 - c_2, v) = d \tag{7.2}$$

Both c_1 and c_2 are bounded by c^+, which is in turn smaller than the smallest prime that divides v. Thus, no divisor of $|c_1 - c_2|$ can divide v and

$$d = \gcd(c_1 - c_2, v) = 1$$

In addition, since ϕ is a homomorphism, from (7.1) and (7.2) we conclude

$$\phi(w) = \phi(a(s_1 \oplus -s_2) \oplus bu) = x^{a(c_1 - c_2)} \otimes x^{bv} = x$$

□

7.2.1 Concrete Instances of Sigma-Phi Protocols

We have formalized several Σ^ϕ-protocols using the functor described in the previous section. For each protocol, we specify the groups G, H and the underlying special homomorphism $\phi : G \to H$, and provide appropriate interpretations for the operator $u(\cdot)$ and the constant special exponent v. Table 7.1 summarizes all the protocols that we have formalized.

Table 7.1. Special homomorphisms in selected Σ^ϕ-protocols. In the table, $\mathbb{Z}_q^+$ stands for the additive group of integers modulo q, $\mathbb{Z}_p^*$ for the multiplicative group of integers modulo p; N is an RSA modulus and e a public RSA exponent coprime with $\varphi(N)$.

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

The Schnorr [1991] and Okamoto [1993] protocols are based on the discrete logarithm problem. For prime numbers p and q such that q divides $p - 1$, a Schnorr group is a multiplicative subgroup of $\mathbb{Z}_p^*$ of order q with generator g. A Σ-protocol for proving knowledge of discrete logarithms in the Schnorr group is obtained by instantiating the construction of Definition 7.4 with the homomorphism

$$\begin{aligned} \phi &: \mathbb{Z}_q^+ \to \mathbb{Z}_p^* \\ \phi(x) &= g^x \end{aligned}$$

Since the order q of the Schnorr group is known, it suffices to take q as the special exponent of the homomorphism, and $u(x) = 0$ for all $x \in \mathbb{Z}_q^+$. The Okamoto protocol is similar to Schnorr protocol but it works with two Schnorr subgroups of $\mathbb{Z}_p^*$ with generators g_1 and g_2, respectively (it can be naturally generalized to any number of generators). In this case ϕ maps a pair (x_1, x_2) to $g_1^{x_1} \otimes g_2^{x_2}$.

Let N be an RSA modulus with prime factors p and q, and let e be a public exponent; e must be co-prime with the totient $\varphi = (p-1)(q-1)$ (i.e. $\gcd(e, \varphi(N)) = 1$). The Guillou-Quisquater [Guillou and Quisquater 1988], Fiat-Shamir [Fiat and Shamir 1987], and Feige-Fiat-Shamir [Feige et al. 1988] protocols are based on the difficulty of solving the RSA problem: given N, e and $y \equiv x^e \mod N$, compute x, the e^{th}-root of y modulo N.

The Guillou-Quisquater protocol is obtained by taking

$$\begin{aligned} \phi &: \mathbb{Z}_N^* \to \mathbb{Z}_N^* \\ \phi(x) &= x^e \end{aligned}$$

The Fiat-Shamir protocol is obtained as a special case when $e = 2$. The Feige-Fiat-Shamir is obtained by taking

$$\begin{aligned} \phi &: \{-1, 1\} \times \mathbb{Z}_N^* \to \mathbb{Z}_N^* \\ \phi(s, x) &= s\, x^2 \end{aligned}$$

Remark

We note that our results hold independently of any computational assumption. Certainly, it is the difficulty of inverting the underlying homomorphism what makes a Σ^ϕ-protocol interesting, but this is inessential for establishing the properties we prove about the protocol.

7.2.2 Composition of Sigma-Phi Protocols

Let $\phi_1 : G_1 \to H_1$ and $\phi_2 : G_2 \to H_2$ be two special homomorphisms with special exponents v_1, v_2 and associated algorithms u_1, u_2, respectively. We give below two useful ways of combining the Σ^ϕ-protocols induced by these homomorphisms.

Theorem 7.7 (Direct product of special homomorphisms). *The following homomorphism from the direct product of G_1 and G_2 to the direct product of H_1 and H_2 is a special homomorphism:*

$$\begin{aligned} \phi &: G_1 \times G_2 \to H_1 \times H_2 \\ \phi(x_1, x_2) &\overset{def}{=} (\phi_1(x_1), \phi_2(x_2)) \end{aligned}$$

Proof. It suffices to take

$$\begin{array}{lcl} v & \stackrel{\text{def}}{=} & \operatorname{lcm}(v_1, v_2) \\ u(x_1, x_2) & \stackrel{\text{def}}{=} & (u_1(x_1)^{v/v_1}, u_2(x_2)^{v/v_2}) \end{array}$$

Indeed,

$$\begin{aligned} \phi(u(x_1, x_2)) &= (\phi_1(u_1(x_1)^{v/v_1}), \phi_2(u_2(x_2)^{v/v_2})) \\ &= (x_1^{v_1 v/v_1}, x_2^{v_2 v/v_2}) \\ &= (x_1, x_2)^v \end{aligned}$$

□

This yields an effective means of AND-combining assertions proved by Σ^ϕ-protocols. The result generalizes the protocol of Maurer [2009,Theorem 6.2]; we do not require that the special exponent be the same.

Theorem 7.8 (Equality of preimages). *Suppose that the domain of both homomorphisms is the same, $G_1 = G_2 = G$, $v_1 = v_2$, and u_1, u_2 are such that*

$$\forall x_1 \in H_1,\ x_2 \in H_2.\ u_1(x_1) = u_2(x_2)$$

Then, the following homomorphism from G to the direct product of H_1 and H_2 is a special homomorphism:

$$\begin{array}{lcl} \phi & : & G \to H_1 \times H_2 \\ \phi(x) & \stackrel{\text{def}}{=} & (\phi_1(x), \phi_2(x)) \end{array}$$

Proof. Take $v \stackrel{\text{def}}{=} v_1$ and $u(x_1, x_2) \stackrel{\text{def}}{=} u_1(x_1)$,

$$\begin{aligned} \phi(u(x_1, x_2)) &= (\phi_1(u_1(x_1)), \phi_2(u_2(x_2))) \\ &= (x_1^{v_1}, x_2^{v_2}) \\ &= (x_1, x_2)^v \end{aligned}$$

□

We can use this latter theorem to construct a Σ-protocol that proves correctness of Diffie-Hellman keys. Given a group with prime order q and a generator g, this amounts to prove that triples of group elements of the form (α, β, γ) are Diffie-Hellman triples, i.e. that if $\alpha = g^a$ and $\beta = g^b$, then $\gamma = g^{ab}$. We instantiate the above construction for homomorphisms $\phi_1(x) = g^x$, and $\phi_2(x) = \beta^x$. Knowledge of a preimage a of (α, γ) implies that (α, β, γ) is a Diffie-Hellman triple (and thus that γ is a valid Diffie-Hellman shared key).

7.3 Sigma Protocols Based on Claw-Free Permutations

This section describes a general construction in the same flavor as the Σ^ϕ construction discussed in the previous section, but based on pairs of *claw-free* permutations [Cramer 1996] rather than on special homomorphisms.

Definition 7.9 (Claw-free permutation pair). *A pair of trapdoor permutations $f = (f_0, f_1)$ on the same domain D is claw-free if it is unfeasible to compute $x, y \in D$ such that $f_0(x, pk) = f_1(y, pk)$.*

Given a claw-free permutation pair f, and a bitstring $a \in \{0,1\}^k$, we define

$$f_{[a]}(b) \stackrel{\text{def}}{=} f_{a_1}(f_{a_2}(\dots(f_{a_k}(b))\dots))$$

where a_i denotes the i^{th} bit of a.

Theorem 7.10 (Σ-protocol based on claw-free permutations). *Let (f_0, f_1) be a pair of claw-free permutations on D and let R be such that*

$$R(pk, sk) \iff \forall x.\ f_0^{-1}(sk, f_0(pk, x)) = x\ \wedge\ f_1^{-1}(sk, f_1(pk, x)) = x$$

The following protocol is a Σ-protocol for relation R:

$$\begin{array}{lcl}
\mathsf{P}_1(pk, sk) & \stackrel{def}{=} & y \stackrel{\$}{\leftarrow} D;\ \mathsf{return}\ (y, y) \\
\mathsf{P}_2(pk, sk, y, c) & \stackrel{def}{=} & \mathsf{return}\ f_{[c]}^{-1}(sk, y) \\
\mathsf{V}_1(pk, r) & \stackrel{def}{=} & c \stackrel{\$}{\leftarrow} \{0,1\}^k;\ \mathsf{return}\ c \\
\mathsf{V}_2(pk, r, c, s) & \stackrel{def}{=} & \mathsf{return}\ \big(f_{[c]}(pk, s) = r\big)
\end{array}$$

except that it might not satisfy the knowledge soundness property.

Proof.

Completeness

The proof follows almost the same structure as the completeness proof for Σ^{ϕ}-protocols. After inlining procedure calls in the protocol, we are left with the goal

$$b \leftarrow f_{[c]}(pk, f_{[c]}^{-1}(sk, y)) = y\ \simeq_{\{b\}}^{R(pk,sk)}\ b \leftarrow \mathsf{true}$$

We use the fact that the pair (pk, sk) is in R to prove that $f_{[c]}(pk, f_{[c]}^{-1}(sk, y)) = y$ by induction on c.

Special Honest Verifier Zero-Knowledge

The following is a sHVZK simulator for the protocol,

$$\textbf{Simulator}\ \mathsf{S}(pk, c):\ s \stackrel{\$}{\leftarrow} D;\ r \leftarrow f_{[c]}(pk, s);\ \mathsf{return}\ (r, s)$$

To prove that

$$\mathsf{Protocol}(pk, sk, c) \simeq_{\{r,c,s\}}^{\{pk,c\} \wedge R(pk,sk)} (r, s) \leftarrow \mathsf{S}(pk, c)$$

we inline every procedure call in both games and perform expression propagation and dead code elimination, we are left with the following goal:

$$\vdash r \xleftarrow{\$} D;\ s \leftarrow f^{-1}_{[c]}(sk, r) \simeq^{\{pk,sk,c\}}_{\{r,s\}} r \leftarrow f_{[c]}(pk, s)$$

which is provable from the fact that f is a permutation pair. □

We observed that the above protocol does not necessarily satisfy the special soundness property. Instead, it satisfies a property known as *collision intractability*: no efficient algorithm can find two accepting conversations with different challenges but same commitment (a collision) with non-negligible probability. Interactive proof protocols that are complete, sHVZK but only satisfy *collision intractability* have important applications as signature protocols.

Theorem 7.11. *It is unfeasible to find a collision for the protocol in Theorem 7.10.*

Proof. By contradiction. Assume two accepting conversations (r, c_1, s_1), (r, c_2, s_2) for a public input pk with $c_1 \neq c_2$. We show that it is possible to efficiently compute a claw (b, b') such that $f_0(b) = f_1(b')$. Since the two conversations are accepting,

$$f_{[c_1]}(pk, s_1) = f_{[c_2]}(pk, s_2) = r$$

The following algorithm computes a claw

```
find_claw($s_1, c_1, s_2, c_2$) :
  if head($c_1$) = head($c_2$)
  then find_claw(tail($c_1$), $s_1$, tail($c_2$), $s_2$)
  else if head($c_1$) = 0
         then($f_{[\mathsf{tail}(c_1)]}(pk, s_1), f_{[\mathsf{tail}(c_2)]}(pk, s_2)$)
         else ($f_{[\mathsf{tail}(c_2)]}(pk, s_2), f_{[\mathsf{tail}(c_1)]}(pk, s_1)$)
```

The algorithm executes in polynomial-time provided permutations f_0 and f_1 can be evaluated in polynomial time, and c_1, c_2 are polynomially bounded. For a polynomially bounded challenge set, this contradicts the assumption that (f_0, f_1) is claw-free. □

7.4 Combination of Sigma-Protocols

There are two immediate, but essential, ways of combining two Σ-protocols (P^1, V^1) and (P^2, V^2) with knowledge relations R_1 and R_2 respectively: AND-combination, and OR-combination. The former allows a prover to prove knowledge of witnesses w_1, w_2 such that $R_1(x_1, w_1)$ and $R_2(x_2, w_2)$. The latter allows a prover to prove knowledge of a witness w such that either $R_1(x_1, w)$ or $R_2(x_2, w)$, without revealing which is the case. This can be naturally extended to proofs of any monotone Boolean formula by nested combination (although there exist a direct, more efficient construction based on secret-sharing schemes, cf. [Cramer 1996]). Even though simple, such constructions are incredibly powerful and form the basis of many practical protocols, like secure electronic voting protocols.

7.4.1 AND-Combination

Two Σ-protocols can be combined into a Σ-protocol that proves simultaneous knowledge of witnesses for both underlying knowledge relations, i.e. a Σ-protocol with a knowledge relation:

$$R \stackrel{\text{def}}{=} \{((x_1, x_2), (w_1, w_2)) \mid R_1(x_1, w_1) \wedge R_2(x_2, w_2)\}$$

We have formalized a functor AND, that combines two public-coin Σ-protocols (P^1, V^1) and (P^2, V^2) in this form. Without loss of generality, we assume that both protocols mandate that honest verifiers choose challenges uniformly from a set of bitstrings of a certain length k. The construction is straightforward; the combination is essentially a parallel composition of the two sub-protocols using the same randomly chosen challenge:

```
$\mathsf{P}_1((x_1, x_2), (w_1, w_2)) \stackrel{\text{def}}{=}$
  $(r_1, state_1) \leftarrow \mathsf{P}^1_1(x_1, w_1);$
  $(r_2, state_2) \leftarrow \mathsf{P}^2_1(x_2, w_2);$
  $\mathsf{return}\ ((r_1, r_2), (state_1, state_2))$

$\mathsf{P}_2((x_1, x_2), (w_1, w_2), state_1, state_2, c) \stackrel{\text{def}}{=}$
  $s_1 \leftarrow \mathsf{P}^1_2(x_1, w_1, state_1, c);$
  $s_2 \leftarrow \mathsf{P}^2_2(x_2, w_2, state_1, c);$
  $\mathsf{return}\ (s_1, s_2)$

$\mathsf{V}_1((x_1, x_2), (r_1, r_2)) \stackrel{\text{def}}{=} c \stackrel{\$}{\leftarrow} \{0,1\}^k;\ \mathsf{return}\ c$

$\mathsf{V}_2((x_1, x_2), (r_1, r_2), c, (s_1, s_2)) \stackrel{\text{def}}{=}$
  $b_1 \leftarrow \mathsf{V}^1_2(x_1, r_1, c, s_1)$
  $b_2 \leftarrow \mathsf{V}^2_2(x_2, r_2, c, s_2)$
  $\mathsf{return}\ (b_1 = \mathsf{true} \wedge b_2 = \mathsf{true})$
```

Observe that V_1 is not built from V^1_1 and V^2_1. The reason for this is that in order to prove soundness, two runs of the protocol for the same public input x with the same commitment r, but with different challenges $c \neq c'$, must yield two runs of each of the sub-protocols with distinct challenges. If the challenge for the main protocol were built from the challenges computed by V^1_1 and V^2_1, e.g. by concatenating them, we would not be able to conclude that the challenges in each pair of conversations extracted for the sub-protocols are different—one could only conclude that this is the case for one of the sub-protocols. Instead, we make use of the public-coin property and simply draw in V_1 a new random challenge that is used in both sub-protocols. This solves the above problem, but also requires that the sub-protocols satisfy the special honest verifier zero-knowledge property, since we need to be able to simulate the sub-protocols for any fixed challenge.

Since AND combination essentially amounts to pairing the two sub-protocols while respecting the structure of a Σ-protocol, all proofs have the same general form: procedure calls are first inlined, and then the goal is manipulated using program transformations to put it in a form where the properties of the sub-protocols can be applied to conclude. We give below a proof sketch of sHVZK and special soundness; a more detailed proof of these properties and a proof of completeness can be found in [Barthe et al. 2010b].

Special Honest-Verifier Zero-Knowledge

The sHVZK simulator for the protocol simply runs the simulators of the sub-protocols to obtain a conversation for each sub-protocol with the same challenge c, the conversations are then combined to obtain a conversation of the main protocol:

```
**Simulator** $\mathsf{S}((x_1, x_2), c)$ :
$(r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c)$;
$(r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c)$;
$\mathsf{return}((r_1, r_2), (s_1, s_2))$
```

Soundness

Soundness requires us to give a PPT knowledge extractor that computes a witness for the knowledge relation R from two accepting runs of the protocol with different challenges but the same commitment. This amounts to computing a witness for each of the sub-protocols and can be done using the corresponding knowledge extractors as follows:

```
$\mathsf{KE}((x_1, x_2), (r_1, r_2), c, c', (s_1, s_2), (s'_1, s'_2))$ :
$w_1 \leftarrow \mathsf{KE}^1(x_1, r_1, c, c', s_1, s'_1)$;
$w_2 \leftarrow \mathsf{KE}^2(x_2, r_2, c, c', s_2, s'_2)$;
$\mathsf{return}\ (w_1, w_2)$
```

Note that an accepting conversation of the main protocol yields an accepting conversation for each one of the sub-protocols. Moreover, since the challenge of the main protocol is used as the challenge in both sub-protocols and $c \neq c'$, the extracted conversations have different challenges. Concretely, from

$$\Pr\left[b \leftarrow \mathsf{V}_2((x_1, x_2), (r_1, r_2), c\ , (s_1, s_2)), m : b = \mathsf{true}\right] = 1$$
$$\Pr\left[b \leftarrow \mathsf{V}_2((x_1, x_2), (r_1, r_2), c', (s'_1, s'_2)), m : b = \mathsf{true}\right] = 1$$

we can prove that for $i = 1, 2$,

$$\Pr\left[w_i \leftarrow \mathsf{KE}^i(x_i, r_i, c, c', s_i, s'_i), m : R_i(x_i, w_i)\right] = 1$$

from the soundness of the sub-protocols and from the fact that

$$\Pr\left[b_i \leftarrow \mathsf{V}_2^i(x_i, r_i, c\ , s_i), m : b_i = \mathsf{true}\right] = 1$$
$$\Pr\left[b_i \leftarrow \mathsf{V}_2^i(x_i, r_i, c', s'_i), m : b_i = \mathsf{true}\right] = 1$$

□

7.4.2 OR-Combination

Two Σ-protocols can also be combined to obtain a protocol that proves knowledge of a witness for the knowledge relation of one of the sub-protocols, but without revealing which. The construction relies on the ability to simulate accepting runs; the basic idea is that the prover runs the real protocol for which it knows a witness, and uses the simulator to generate a run of the other protocol. The knowledge relation suggested by, e.g. Damgård [2010],

$$\hat{R} \stackrel{\text{def}}{=} \{((x_1, x_2), w) \mid R_1(x_1, w) \;\vee\; R_2(x_2, w)\}$$

suffers from placing unrealistic demands on the simulator. As pointed out by Cramer [1996], it is important to allow the simulator to fail on an input $x \notin \mathsf{dom}(R)$. However, in order to prove completeness for the above relation, the simulator must be able to perfectly simulate outside the domain of the respective knowledge relation. Instead, we can prove completeness (and sHVZK) of the combination with respect to a knowledge relation whose domain is restricted to the Cartesian product of the domains of the knowledge relations of the sub-protocols, i.e.

$$R \stackrel{\text{def}}{=} \left\{ ((x_1, x_2), w) \;\middle|\; \begin{array}{l} (R_1(x_1, w) \;\wedge\; x_2 \in \mathsf{dom}(R_2)) \;\vee \\ (R_2(x_2, w) \;\wedge\; x_1 \in \mathsf{dom}(R_1)) \end{array} \right\}$$

Unfortunately, we cannot prove soundness with respect to R, we can only prove it with respect to $\hat{R}$. The reason for this is that an accepting run of the combined protocol only guarantees the existence of a witness for the public input of one of the protocols, the simulation of the other protocol may succeed even if the input is not in the domain of the respective relation. Otherwise said, from two accepting runs of the combined protocol with distinct challenges we might not be able to extract two accepting runs with distinct challenges for each of the sub-protocols; we can only guarantee we can do that for one of them. Observe that we do not really lose anything by proving completeness with respect to the smaller relation R. If we admitted pairs (x_1, x_2) as public input where one component does not belong to the domain of the corresponding knowledge relation, we would not be able to say anything about the success of the simulator. The simulator might as well fail, trivially revealing that the prover could not have known a witness for the corresponding input, and rendering the protocol pointless for such inputs.

Compared to the AND combination, the OR combination is harder to fit into the structure of a Σ-protocol. The reason for this is that the first phase of the prover needs to use the simulator of one of the sub-protocols, which results in a full (accepting) conversation that has to be passed over to the second phase of the prover. Given $R_1(x_1, w)$, the OR prover runs the prover of the first protocol and the simulator of the second, and returns as a commitment a pair with the commitments of each protocol; it passes over in the state the challenge and the reply of the simulated conversation,

```
$\mathsf{P}_1((x_1, x_2), w) \stackrel{\text{def}}{=}$
    if $(x_1, w) \in R_1$ then
        $(r_1, state_1) \leftarrow \mathsf{P}_1^1(x_1, w)$;
        $c_2 \xleftarrow{\$} \{0,1\}^k$;
        $(r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c_2)$;
        $state \leftarrow (state_1, c_2, s_2)$
    else
        $(r_2, state_2) \leftarrow \mathsf{P}_1^2(x_2, w)$;
        $c_1 \xleftarrow{\$} \{0,1\}^k$;
        $(r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c_1)$;
        $state \leftarrow (state_2, c_1, s_1)$
    return $((r_1, r_2), state)$
```

Above, the test $(x_1, w) \in R_1$ is an encoding of the fact that the prover knows to which knowledge relation corresponds the witness w, and thus which protocol it can run for real, while simulating the other one. The commitment (r_1, r_2) is passed along to the verifier that simply replies by returning a randomly chosen bitstring to the prover, the combination is a public-coin protocol,

$$\mathsf{V}_1((x_1, x_2), (r_1, r_2)) \stackrel{\text{def}}{=} c \xleftarrow{\$} \{0,1\}^k;\ \mathsf{return}\ c$$

Assume without loss of generality that $R_1(x_1, w)$. In the second phase the prover constructs the challenge for the first protocol by xor-ing the challenge c of the OR protocol with the challenge used in the simulation of the second protocol in the first phase. It then runs the second phase of the prover of the first protocol to compute a reply. The result of the second phase is constructed from the challenges for each protocol and the prover replies (the ones coming from the simulated protocol are recovered from the state):

```
$\mathsf{P}_2((x_1, x_2), w, (state, c', s), c) \stackrel{\text{def}}{=}$
    if $(x_1, w) \in R_1$ then
        $state_1 \leftarrow state;\ c_2 \leftarrow c';\ s_2 \leftarrow s$;
        $c_1 \leftarrow c_2 \oplus c$;
        $s_1 \leftarrow \mathsf{P}_2^1(x_1, w, state_1, c_1)$
    else
        $state_2 \leftarrow state;\ c_1 \leftarrow c';\ s_1 \leftarrow s$;
        $c_2 \leftarrow c_1 \oplus c$;
        $s_2 \leftarrow \mathsf{P}_2^2(x_2, w, state_2, c_2)$
    return $((c_1, s_1), (c_2, s_2))$
```

The verifier accepts a conversation when the runs of both protocols are accepting and the challenge is the xor of the challenges used in each of the combined protocols,

```
$\mathsf{V}_2((x_1, x_2), (r_1, r_2), c, ((c_1, s_1), (c_2, s_2))) \stackrel{\text{def}}{=}$
    $b_1 \leftarrow \mathsf{V}_2^1(x_1, r_1, c_1, s_1)$;
    $b_2 \leftarrow \mathsf{V}_2^2(x_2, r_2, c_2, s_2)$;
    return $(c = c_1 \oplus c_2 \wedge b_1 = \mathsf{true} \wedge b_2 = \mathsf{true})$
```

Completeness

The proof is slightly more involved than the proof for the AND combination, since only one of the protocols is run for real, while the other is just simulated, and this depends on the knowledge of the prover. Thus, the proof is split into two cases:

- **case** $(x_1, w) \in R_1$: the outline of the proof is as follows:

$$\begin{aligned}\mathsf{Protocol}((x_1,x_2),w) &\simeq \mathsf{Protocol}_1(x_1,w);\ c_2 \xleftarrow{\$} \{0,1\}^k;\ (r_2,s_2) \leftarrow \mathsf{S}_2(x_2,c_2)\\ &\simeq \mathsf{Protocol}_1(x_1,w);\ \mathsf{Protocol}_2(x_2,w')\end{aligned}$$

  The first equivalence is immediate from inlining procedure calls and simplification. The second equivalence follows from the fact that $\exists w'.\ R_2(x_2, w')$ and the sHVZK property of the second protocol. The proof concludes by application of the completeness property of each of the sub-protocols.
- **case** $(x_2, w) \in R_2$: Idem.

Special Honest-Verifier Zero-Knowledge

The simulator for the OR combination is easily built from the simulators of the sub-protocols:

```
**Simulator** $\mathsf{S}((x_1, x_2), c)$ :
$c_2 \xleftarrow{\$} \{0,1\}^k$;
$c_1 \leftarrow c \oplus c_2$;
$(r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c_1)$;
$(r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c_2)$;
return $((r_1, r_2), ((c_1, s_1), (c_2, s_2)))$
```

As before, the proof is split into two cases:

- **case** $(x_1, w) \in R_1$:

$$\begin{aligned}\mathsf{Protocol}((x_1,x_2),w) &\simeq \mathsf{Protocol}_1(x_1,w);\ \mathsf{S}_2(x_2)\\ &\simeq \mathsf{S}_1(x_1);\ \mathsf{S}_2(x_2)\\ &\simeq \mathsf{S}((x_1,x_2),c)\end{aligned}$$

  Where the first and last steps are immediate from inlining, and simplification, whereas the second step is a direct application of the HVZK property of S_1 (which follows from sHVZK by Theorem 7.2).
- **case** $(x_2, w) \in R_2$: Idem.

Soundness

(With respect to $\hat{R}$). Unlike the AND combination, the OR combination does not have the property that runs with distinct challenges guarantee that the challenges used in the sub-protocols are also distinct. This is not as problematic as in the case of the AND combination, since it suffices to compute a w such that either $R_1(x_1, w)$ or $R_2(x_2, w)$. Furthermore, from

$$c = c_1 \oplus c_2 \neq c' = c'_1 \oplus c'_2$$

we have either $c_1 \neq c'_1$ or else $c_1 = c'_1$, in which case necessarily $c_2 \neq c'_2$. Thus, the knowledge extractor simply needs to do a case analysis:

```
KE((x_1, x_2), (r_1, r_2), c, c',
    ((c_1, s_1), (c_2, s_2)), ((c'_1, s'_1), (c'_2, s'_2))) :
  if $c_1 \neq c'_1$ then
    $w \leftarrow \mathsf{KE}^1(x_1, r_1, c_1, c'_1, s_1, s'_1)$
  else
    $w \leftarrow \mathsf{KE}^2(x_2, r_2, c_2, c'_2, s_2, s'_2)$
  return $w$
```

Assume two accepting runs of the combined protocol with the same commitment and $c \neq c'$:

$$((x_1, x_2), (r_1, r_2), c, ((c_1, s_1), (c_2, s_2)))$$
$$((x_1, x_2), (r_1, r_2), c', ((c'_1, s'_1), (c'_2, s'_2)))$$

We have to establish that for an $i \in 1, 2$,

$$\Pr\left[w_i \leftarrow \mathsf{KE}^i(x_i, r_i, c_i, c'_i, s_i, s'_i) : R_i(x_i, w_i)\right] = 1$$

depending on whether $c_1 \neq c'_1$ or $c_1 = c'_1 \wedge c_2 \neq c'_2$,

- **case** $c_1 \neq c'_1$: From the special soundness of $\mathsf{Protocol}_1$,

$$\Pr\left[w_1 \leftarrow \mathsf{KE}^1(x_1, r_1, c_1, c'_1, s_1, s'_1) : R_1(x_1, w_1)\right] = 1$$

- **case** $c_1 = c'_1$ (and thus $c_2 \neq c'_2$): From the special soundness of $\mathsf{Protocol}_2$

$$\Pr\left[w_2 \leftarrow \mathsf{KE}^2(x_2, r_2, c_2, c'_2, s_2, s'_2) : R_2(x_2, w_2)\right] = 1$$

□

7.5 Related Work

Our work participates to an upsurge of interest in Σ-protocols, and shares some motivations and commonalities with recently published papers. Specifically, our account of Σ^ϕ-protocols coincides with Maurer's [2009] unifying treatment of proofs of knowledge for preimages of group homomorphisms. Concretely, Maurer exhibits a main protocol that uses a group homomorphism—which in our setting corresponds to the definition of the module of Σ^ϕ-protocols in Section 7.2—and shows (in his Theorem 3) that under suitable hypotheses the main protocol is a Σ-protocol. He gives several instances of the main protocol by picking suitable group homomorphisms and showing that they satisfy these hypotheses.

Our work is also closely connected to the recent effort of Bangerter et al. [2008, 2010] to design and implement efficient zero-knowledge proofs of knowledge. They provide both a set of sufficient conditions on a homomorphism ϕ under which the corresponding Σ^ϕ-protocol can be viewed as a Σ-protocol [Bangerter et al. 2008,Theorem 1], and a generalization that allows to consider sets of linear relations among preimages of group homomorphisms [Bangerter et al. 2008,Theorem 2]. The latter result is used to justify the soundness of a compiler that generates efficient code from high-level descriptions of protocols. As future work, the authors of [Bangerter et al. 2008] mention that they plan to make their compiler certifying, so that it would generate proofs accompanying the code. Doing so from scratch remains a daunting task. By building on CertiCrypt, our formalization could be readily used as a stepping stone for a modular certifying compiler, in which (high-level, unoptimized) code is certified, and then compiled to efficient code using a certified or certifying compiler; see e.g. [Barthe et al. 2009b; Leroy 2006] for an instance of applying ideas from certified/certifying compilation to cryptography.

Cryptographic primitives need not only be secure; they must also be used correctly. In a series of papers, Backes, Hritcu, and Maffei [2008b]; Backes, Maffei, and Unruh [2008c] develop sound analysis methods for protocols that use zero-knowledge proofs, and apply their analyses to verify the authentication and secrecy properties of the Direct Anonymous Attestation Protocol. One extremely ambitious objective would be to use their results, which complement ours, to fully certify the security of the protocol in the computational model. Intermediate results would involve formalizing computational soundness results [Abadi and Rogaway 2002; Cortier and Warinschi 2005], which represents a substantial amount of work on its own.

7.6 Perspectives

We have presented a formalization of Σ-protocols in CertiCrypt. The highlights of our formalization are its generic account of the class of Σ^{ϕ}-protocols and the detailed treatment of the AND/OR composition. Our work complements recent advances in the field, and takes a first but important step towards formalizing a rich theory of zero-knowledge proofs. In our opinion, and judging by the myriad of small variations in definitions we have found in the literature, this effort would be worth pursuing for a field that strives for definitional clarity and consistency.

Compared to other applications of CertiCrypt, like the verification of security proofs of encryption and signature schemes discussed in previous chapters, the formalization presented here imposes challenges of a different nature to the user. In contrast to earlier case studies, for which we have developed a mature set of techniques that mechanize most of the reasoning patterns appearing in proofs, we found that the formalization of Σ-protocols does not require as much complex reasoning, but is more demanding with respect to the compositionality of proofs. This led us to revise some design choices of CertiCrypt and has given us ideas on how to improve the framework so that results can be reused and composed more easily. For instance, when composing proofs of observational equivalence statements the user often needs to manually rename variables to match the names of the context where the proof is being reused; currently the user has to appeal to the `alloc` tactic to do this, but a simply heuristic may suffice in most cases.

We can build on the existing formalization to verify other important results about zero-knowledge proofs. These include other means of composing protocols: sequential [Goldreich and Oren 1994] and concurrent [Damgård 2000; Garay et al.

2006] composition; transforming public-coin zero-knowledge proofs in general zero-knowledge proofs [Goldreich 2002], or different formulations like non-interactive zero-knowledge proofs [Blum et al. 1988] or properties, i.e. statistical zero-knowledge and computational zero-knowledge instead of perfect zero-knowledge. Moreover, Σ-protocols form the base for a number of important and intriguing protocols for electronic voting schemes [Cramer 1996], identity schemes [Cramer 1996], and commitment schemes [Cramer 1996; Damgård 1990]. All are prime targets for future formalizations.

8 Related Work and Conclusion

8.1 Related Work

Cryptographic protocol verification is an established area of formal methods, and a wealth of automated and deductive methods have been developed to the purpose of verifying that protocols provide the expected level of security [Meadows 2003]. Traditionally, protocols have been verified in a symbolic model, for which effective decision procedures exist under suitable hypotheses [Abadi and Cortier 2006]. Although the symbolic model assumes perfect cryptography, soundness results such as [Abadi and Rogaway 2002]—see [Cortier et al. 2010] for a recent survey—relate the symbolic model with the computational model, provided the cryptographic primitives satisfy adequate notions of security. It is possible to combine symbolic methods and soundness proofs to achieve guarantees in the computational model, as done e.g. in [Backes and Laud 2006; Backes et al. 2010; Sprenger and Basin 2008]. One drawback of this approach is that the security proof relies on intricate soundness proofs and hypotheses that unduly restrict the usage of primitives. Besides, it is not clear whether computational soundness results will always exist to allow factoring verification through symbolic methods [Backes and Pfitzmann 2005]. Consequently, some authors attempt to provide guarantees directly at the computational level [Blanchet 2008; Laud 2001; Roy et al. 2008].

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

Blanchet and Pointcheval [2006] were among the first to use verification tools to carry out game-based proofs of cryptographic schemes. They used CryptoVerif to prove the existential unforgeability of the FDH signature scheme, with the original security bound given in Section 5.1, which is much looser than the one given in Section 5.2. CryptoVerif has also been used to verify the security of many protocols, including Kerberos [Blanchet et al. 2008]. It is difficult to assess CryptoVerif ability to handle automatically more complex cryptographic proofs (or tighter security bounds), e.g. for schemes such as OAEP; on the other hand, compiling CryptoVerif sequences of games in CertiCrypt is an interesting research direction that would increase automation in CertiCrypt and confidence in CryptoVerif—by generating independently verifiable proofs.

Impagliazzo and Kapron [2006] were the first to develop a logic to reason about indistinguishability. Their logic is built upon a more general logic whose soundness relies on non-standard arithmetic; they show the correctness of a pseudo-random generator and that next-bit unpredictability implies pseudo-randomness. Recently, Zhang [2009] developed a similar logic on top of Hofmann's SLR system [Hofmann 1998] and reconstructed the examples of Impagliazzo and Kapron [2006]. These logics have limited applicability because they lack support for oracles or adaptive adversaries and so cannot capture many of the the standard patterns for reasoning about cryptographic schemes. More recently Barthe et al. [2010a] developed a general logic, called Computational Indistinguishability Logic (CIL), that captures reasoning patterns that are common in provable security, such as simulation and reduction, and deals with oracles and adaptive adversaries. They use CIL to prove the security of the Probabilistic Signature Scheme, a widely used signature scheme that forms part of the PKCS standard [Bellare and Rogaway 1996]. CIL subsumes an earlier logic by Courant et al. [2008], who developed a form of strongest post-condition calculus that can establish automatically asymptotic security (IND-CPA and IND-CCA2) of encryption schemes that use one-way functions and hash functions modeled as random oracles. They show soundness and provide a prototype implementation that covers many examples in the literature.

In parallel, several authors have initiated formalizations of game-based proofs in proof assistants and shown the security of basic examples. Nowak [2007] gives a game-based proof of ElGamal semantic security in Coq. Nowak uses a shallow embedding to model games; his framework ignores complexity issues and has limited support for proof automation: because there is no special syntax for writing games, mechanizing syntactic transformations becomes very difficult. Affeldt et al. [2007] formalize a game-based proof of the PRP/PRF switching lemma in Coq. However, their formalization is tailored towards the particular example they consider, which substantially simplifies their task and hinders generality. They deal with a weak (non-adaptive) adversary model and ignore complexity. In another attempt to build a system supporting provable security, Backes et al. [2008a] formalize a language for games in the Isabelle proof assistant and prove the Fundamental Lemma; however, no examples are reported. All in all, these works appear like preliminary experiments that are not likely to scale.

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

8.2 Conclusion

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

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

In addition, it would be useful to formalize cryptographic meta-results such as the equivalence between IND-CPA and IND-CCA2 under plaintext awareness, or the transformation of an IND-CPA-secure scheme into an IND-CCA2-secure scheme [Fujisaki and Okamoto 1999]. Another direction would be to formalize proofs of computational soundness of the symbolic model, see e.g. [Abadi and Rogaway 2002] and proofs of automated methods for proving security of primitives and protocols, see e.g. [Courant et al. 2008; Laud 2001]. Finally, it would also be worthwhile to explore applications of CertiCrypt outside the realm cryptography, in particular to randomized algorithms and complexity.

Complexity and termination analysis of probabilistic programs

CertiCrypt provides the necessary ingredients to reason about termination and complexity of programs. Yet cryptographic applications only make a limited use of them; e.g. we only use simple closure properties of PPT programs. It would be instructive to extend our formalization to define standard complexity classes and to prove the complexity of well-known probabilistic algorithms. More generally, we are interested in developing automated methods to carry such analyses for programs with loops and recursive calls.

Reasoning about probabilistic programs

The pWHILE language is sufficiently powerful to program widely used randomized algorithms, and it would be attractive to endow the formal semantics in Coq with a mechanized program logic that allows proving formally properties of these algorithms, in the spirit of the work of Hurd et al. [2005].

References

M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci.*, 367(1-2):2–32, 2006. Cited in page 133.

M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002. Cited in pages 131, 133, and 135.

R. Affeldt, M. Tanaka, and N. Marti. Formal proof of provable security by game-playing in a proof assistant. In *1st International conference on Provable Security, ProvSec 2007*, volume 4784 of *Lecture Notes in Computer Science*, pages 151–168, Berlin, 2007. Springer. Cited in pages 66 and 134.

T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2006*, pages 91–102, New York, 2006. ACM. Cited in page 46.

P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009. Cited in pages 2, 15, 17, and 134.

M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *13th ACM conference on Computer and Communications Security, CCS 2006*, pages 370–379, New York, 2006. ACM. Cited in page 133.

M. Backes and B. Pfitzmann. Limits of the cryptographic realization of Dolev-Yao-style XOR. In *Computer Security – ESORICS 2005, 10th European symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 178–196, Berlin, 2005. Springer. Cited in page 133.

M. Backes, M. Berg, and D. Unruh. A formal language for cryptographic pseudocode. In *15th International conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 353–376. Springer, 2008a. Cited in page 134.

M. Backes, C. Hritcu, and M. Maffei. Type-checking zero-knowledge. In *15th ACM conference on Computer and Communications Security, CCS 2008*, pages 357–370. ACM, 2008b. Cited in page 131.

M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the Direct Anonymous Attestation protocol. In *29th IEEE symposium on Security and Privacy, S&P 2008*, pages 202 –215. IEEE Computer Society, 2008c. Cited in page 131.

M. Backes, M. P. Grochulla, C. Hritcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *22nd IEEE Computer Security Foundations symposium, CSF 2009*, pages 308–323. IEEE Computer Society, 2009. Cited in page 111.

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

York, 2010. ACM. Cited in page 133.

E. Bangerter, J. Camenisch, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. Cited in pages 130 and 131.

E. Bangerter, J. Camenisch, and S. Krenn. Efficiency limitations for Sigma-protocols for group homomorphisms. In *7th Theory of Cryptography conference, TCC 2010*, volume 5978 of *Lecture Notes in Computer Science*, pages 553–571. Springer, 2010. Cited in page 130.

G. Barthe, J. Cederquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In *Automated Reasoning, 2nd International Joint conference, IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 385–399, Berlin, 2004. Springer. Cited in page 133.

G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2009a. Springer. Cited in pages 14 and 25.

G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5):1–45, 2009b. Cited in page 131.

G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2009*, pages 90–101, New York, 2009c. ACM. Cited in pages 2, 25, 50, and 62.

G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *17th ACM conference on Computer and Communications Security, CCS 2010*, New York, 2010a. ACM. Cited in page 134.

G. Barthe, D. Hedin, S. Zanella Béguelin, B. Gregoire, and S. Heraud. A machine-checked formalization of Sigma-protocols. In *23rd IEEE Computer Security Foundations symposium, CSF 2010*, pages 246–260, Los Alamitos, Calif., 2010b. IEEE Computer Society. Cited in page 125.

M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *1st ACM conference on Computer and Communications Security, CCS 1993*, pages 62–73, New York, 1993. ACM. Cited in pages 33, 67, 69, and 83.

M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111, Berlin, 1994. Springer. Cited in page 88.

M. Bellare and P. Rogaway. The exact security of digital signatures – How to sign with RSA and Rabin. In *Advances in Cryptology – EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416, Berlin, 1996. Springer. Cited in pages 33, 67, 83, and 134.

M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Berlin, 2006. Springer. Cited in pages 1, 2, 20, 50, 65, and 88.

M. Bellare, D. Hofheinz, and E. Kiltz. Subtleties in the definition of IND-CCA: When and how should challenge-decryption be disallowed? Cryptology ePrint Archive, Report 2009/418, 2009. Cited in pages 33, 36, and 37.

N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2004*, pages 14–25, New York, 2004. ACM. Cited in pages 41, 46, and 48.

Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81, Berlin, 2006. Springer. Cited in page 48.

B. Blanchet. A computationally sound mechanized prover for security protocols. In *27th IEEE symposium on Security and Privacy, S&P 2006*, pages 140–154. IEEE Computer Society, 2006. Cited in page 85.

B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput.*, 5(4):193–207, 2008. Cited in page 133.

B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554, Berlin, 2006. Springer. Cited in pages 85 and 133.

B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *15th ACM conference on Computer and Communications Security, CCS 2008*, pages 87–99, New York, 2008. ACM. Cited in page 134.

M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. In *20th Annual ACM symposium on Theory of computing, STOC 1988*, pages 103–112. ACM, 1988. Cited in page 132.

D. Boneh. Simplified OAEP for the RSA and Rabin functions. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 275–291, Berlin, 2001. Springer. Cited in page 88.

D. Brown. What hashes make RSA-OAEP secure? Cryptology ePrint Archive, Report 2006/223, 2006. Cited in page 109.

S. Cavallar, B. Dodson, A. Lenstra, W. Lioen, P. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA modulus. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18, Berlin, 2000. Springer. Cited in page 81.

J.-S. Coron. On the exact security of Full Domain Hash. In *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2000. Cited in pages 75 and 83.

J.-S. Coron. Optimal security proofs for PSS and other signature schemes. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287, Berlin, 2002. Springer. Cited in page 83.

V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *Programming Languages and Systems, 14th European symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005. Cited in page 131.

V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, pages 1–35, 2010. Cited in page 133.

J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *15th ACM conference on Computer and Communications Security, CCS 2008*, pages 371–380, New York, 2008. ACM. Cited in pages 134 and 135.

R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI and Uni. of Amsterdam, 1996. Cited in pages 111, 114, 122, 124, 127, and 132.

I. Damgård. On the existence of bit commitment schemes and zero-knowledge proofs. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 17–27. Springer, 1990. Cited in page 132.

I. Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 418–430. Springer, 2000. Cited in page 131.

I. Damgård. On sigma-protocols. Lecture Notes on Cryptologic Protocol Theory, 2010. Cited in page 127.

I. Damgård and B. Pfitzmann. Sequential iteration of interactive arguments and an efficient zero-knowledge argument for NP. In *Automata, Languages and Programming, 25th International Colloquiumm, ICALP 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 772–783. Springer, 1998. Cited in page 114.

J. den Hartog. Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic Hoare style logic. *Sci. Comput. Program.*, 74(1-2):52–63, 2008. Cited in page 133.

U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptology*, 1(2): 77–94, 1988. Cited in page 121.

A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO 1986*, pages 186–194. Springer, 1987. Cited in page 121.

E. Fujisaki and T. Okamoto. How to enhance the security of public-key encryption at minimum cost. In *2nd International workshop on Practice and Theory in Public Key Cryptography, PKC 1999*, volume 1560 of *Lecture Notes in Computer Science*, pages 634–634, Berlin, 1999. Springer. Cited in page 135.

E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. *J. Cryptology*, 17(2):81–104, 2004. Cited in pages 88 and 101.

J. A. Garay, P. MacKenzie, and K. Yang. Strengthening zero-knowledge protocols using signatures. *J. Cryptology*, 19:169–209, 2006. Cited in page 131.

O. Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001. Cited in page 28.

O. Goldreich. Zero-knowledge twenty years after its invention. Technical Report TR02-063, Electronic Colloquium on Computational Complexity, 2002. Cited in pages 111 and 132.

O. Goldreich and Y. Oren. Definitions and properties of zero-knowledge proof systems. *J. Cryptology*, 7(1):1–32, 1994. Cited in page 131.

O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986. Cited in page 58.

S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2): 270–299, 1984. Cited in pages 1 and 6.

S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989. Cited in page 111.

G. Gonthier. Formal Proof — The Four Colour Theorem. *Notices of the AMS*, 55(11): 1382–1393, 2008. Cited in page 4.

L. Guillou and J.-J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Advances in Cryptology – EUROCRYPT 1988*, volume 330 of *Lecture Notes in Computer Science*, pages 123–128. Springer, 1988. Cited in page 121.

T. Hales. Formal Proof. *Notices of the AMS*, 55(11):1370–1380, 2008. Cited in page 4.

T. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete and Computational Geometry*, 44(1): 1–34, 2010. Cited in page 4.

S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. Cited in pages 1 and 2.

C. Hall, D. Wagner, J. Kelsey, and B. Schneier. Building PRFs from PRPs. In *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 370–389. Springer, 1998. Cited in page 62.

J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999. Cited in page 14.

M. Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *11th International workshop on Computer Science Logic, CSL 1997*, volume 1414 of *Lecture Notes in Computer Science*, pages 275–294, Berlin, 1998. Springer. Cited in page 134.

J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1):96–112, 2005. Cited in pages 135 and 136.

R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. *J. Comput. Syst. Sci.*, 72(2):286–320, 2006. Cited in page 134.

R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *21st Annual ACM symposium on Theory of Computing, 1989*, pages 44–61, New York, 1989. ACM. Cited in pages 63 and 65.

B. Jonsson, W. Yi, and K. G. Larsen. Probabilistic extensions of process algebras. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 685–710. Elsevier, Amsterdam, 2001. Cited in pages 17, 18, and 19.

J. Katz and N. Wang. Efficiency improvements for signature schemes with tight security reductions. In *10th ACM conference on Computer and Communications Security, CCS 2003*, pages 155–164, New York, 2003. ACM. Cited in pages 33 and 84.

E. Kiltz and K. Pietrzak. On the security of padding-based encryption schemes — or — why we cannot prove OAEP secure in the standard model. In *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 389–406, Berlin, 2009. Springer. Cited in page 109.

E. Kiltz, A. O'Neill, and A. Smith. Instantiability of RSA-OAEP under chosen-plaintext attack. In *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in*

Computer Science, pages 295–313, Berlin, 2010. Springer. Cited in page 109.

G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006. Cited in page 4.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *22nd ACM symposium on Operating Systems Principles, SOSP 2009*, pages 207–220. ACM Press, 2009. Cited in page 4.

T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology — CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Berlin, 2010. Cited in page 81.

D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981. Cited in page 17.

P. Laud. Semantics and program analysis of computationally secure information flow. In *Programming Languages and Systems, 10th European symposium on Programming, ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91, Berlin, 2001. Springer. Cited in pages 133 and 135.

A. K. Lenstra and H. W. L. Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer, Berlin, 1993. Cited in page 81.

A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *J. Cryptology*, 14(4): 255–293, 2001. Cited in page 81.

A. K. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. Leyland. Factoring estimates for a 1024-bit RSA modulus. In *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 55–74, Berlin, 2003. Springer. Cited in page 82.

X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2006*, pages 42–54, New York, 2006. ACM. Cited in pages 4, 48, and 131.

M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988. Cited in pages 59 and 62.

U. Maurer. Unifying zero-knowledge proofs of knowledge. In *Progress in Cryptology – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2009. Cited in pages 122 and 130.

C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE J. Sel. Areas Commun.*, 21(1):44–54, 2003. Cited in page 133.

D. Nowak. A framework for game-based security proofs. In *9th International conference on Information and Communications Security, ICICS 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333, Berlin, 2007. Springer. Cited in page 134.

T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *Advances in Cryptology – CRYPTO 1992*, volume 740 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 1993. Cited in page 120.

T. Okamoto and D. Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In *4th International workshop on Practice and Theory in Public Key Cryptography, PKC 2001*, volume 1992 of *Lecture Notes in Computer Science*, pages 104–118, Berlin, 2001a. Springer. Cited in page 10.

T. Okamoto and D. Pointcheval. REACT: Rapid Enhanced-Security Asymmetric Cryptosystem Transform. In *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 159–174, Berlin, 2001b. Springer. Cited in page 88.

L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. of Comput. Secur.*, 6(1-2):85–128, 1998. Cited in page 4.

D. Pointcheval. Provable security for public key schemes. In *Advanced Courses on Contemporary Cryptology*, chapter D, pages 133–189. Birkhäuser Basel, 2005. Cited in pages 85, 101, 104, 108, and 109.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2002*, pages 154–165, New York, 2002. ACM. Cited in page 17.

A. Roy, A. Datta, A. Derek, and J. Mitchell. Inductive proofs of computational secrecy. In *Computer Security – ESORICS 2007, 12th European symposium on Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 219–234, Berlin, 2008. Springer. Cited in page 133.

RSA Data Security, Inc. PKCS #1 v2.1: RSA encryption standard, June 2002. Cited in page 83.

A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001. Cited in page 18.

C.-P. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991. Cited in pages 116 and 120.

A. Shamir and E. Tromer. On the cost of factoring RSA-1024. *RSA CryptoBytes*, 6:10–19, 2003. Cited in page 82.

V. Shoup. OAEP reconsidered. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259, Berlin, 2001. Springer. Cited in pages 88 and 109.

V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. Cited in pages 1, 7, 41, and 65.

C. Sprenger and D. Basin. Cryptographically-sound protocol-model abstractions. In *21st IEEE Computer Security Foundations symposium, CSF 2008*, pages 115–129, Los Alamitos, Calif., 2008. IEEE Computer Society. Cited in page 133.

J. Stern. Why provable security matters? In *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 644–644, Berlin, 2003. Springer. Cited in page 1.

The Coq development team. The Coq Proof Assistant Reference Manual Version 8.2. Online – `http://coq.inria.fr`, 2009. Cited in pages 2 and 21.

Y. Zhang. The computational SLR: A logic for reasoning about computational indistinguishability. In *8th International conference on Typed Lambda Calculi and Applications, TLCA 2008*, volume 5608 of *Lecture Notes in Computer Science*, pages 401–415, Berlin, 2009. Springer. Cited in page 134.

Index

Certification Formelle de Preuves Cryptographiques Basées sur les Séquences de Jeux

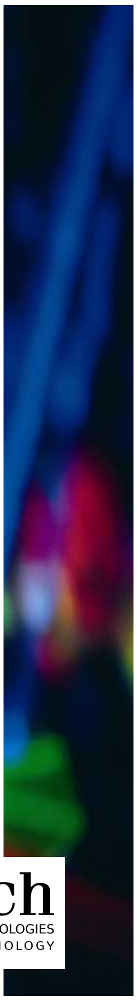
Résumé : Les séquences de jeux sont une méthodologie établie pour structurer les preuves cryptographiques. De telles preuves peuvent être formalisées rigoureusement en regardant les jeux comme des programmes probabilistes et en utilisant des méthodes de vérification de programmes. Cette thèse décrit CertiCrypt, un outil permettant la construction et vérification automatique de preuves basées sur les jeux. CertiCrypt est implementé dans l'assistant à la preuve Coq, et repose sur de nombreux domaines, en particulier les probabilités, la complexité, l'algèbre, et la sémantique des langages de programmation. CertiCrypt fournit des outils certifiés pour raisonner sur l'équivalence de programmes probabilistes, en particulier une logique de Hoare relationnelle, une théorie équationnelle pour l'équivalence observationnelle, une bibliothèque de transformations de programme, et des techniques propres aux preuves cryptographiques, permettant de raisonner sur les évènements. Nous validons l'outil en formalisant les preuves de sécurité de plusieurs exemples emblématiques, notamment le schéma de chiffrement OAEP et le schéma de signature FDH.

Mots clés : Cryptographie, langages de programmation, complexité, vérification de programmes, preuves formelles, assistant à la preuve Coq

Formal Certification of Game-Based Cryptographic Proofs

Abstract: The game-based approach is a popular methodology for structuring cryptographic proofs as sequences of games. Game-based proofs can be rigorously formalized by taking a code-centric view of games as probabilistic programs and relying on programming language techniques to justify proof steps. In this dissertation we present CertiCrypt, a framework that enables the machine-checked construction and verification of game-based cryptographic proofs. CertiCrypt is built upon the general-purpose proof assistant Coq, from which it inherits the ability to provide independently verifiable evidence that proofs are correct, and draws on many areas, including probability and complexity theory, algebra, and semantics of programming languages. The framework provides certified tools to reason about the equivalence of probabilistic programs, including a relational Hoare logic, a theory of observational equivalence, verified program transformations, and ad-hoc programming language techniques of particular interest in cryptographic proofs, such as reasoning about failure events. We validate our framework through the formalization of several significant case studies, including proofs of security of the Optimal Asymmetric Encryption Padding scheme against adaptive chosen-ciphertext attacks, and of existential unforgeability of Full-Domain Hash signatures.

Keywords: Cryptography, programming languages, complexity, program verification, formal proofs, Coq proof assistant



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY