An abridged version of this paper will appear in the proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012. ACM Press, 2012. This is the full version.

Verified Security of Redundancy-Free Encryption from Rabin and RSA

Gilles Barthe[*] David Pointcheval[†] Santiago Zanella-Béguelin[‡]

August 2012

Abstract

Verified security provides a firm foundation for cryptographic proofs by means of rigorous programming language techniques and verification methods. EasyCrypt is a framework that realizes the verified security paradigm and supports the machine-checked construction and verification of cryptographic proofs using state-of-the-art SMT solvers, automated theorem provers and interactive proof assistants. Previous experiments have shown that EasyCrypt is effective for a posteriori validation of cryptographic systems. In this paper, we report on the first application of verified security to a novel cryptographic construction, with strong security properties and interesting practical features. Specifically, we use EasyCrypt to prove in the Random Oracle Model the IND-CCA security of a redundancy-free public-key encryption scheme based on trapdoor one-way permutations. Somewhat surprisingly, we show that even with a zero-length redundancy, Boneh's SAEP scheme (an OAEP-like construction with a single-round Feistel network rather than two) converts a trapdoor one-way permutation into an IND-CCA-secure scheme, provided the permutation satisfies two additional properties. We then prove that the Rabin function and RSA with short exponent enjoy these properties, and thus can be used to instantiate the construction we propose to obtain efficient encryption schemes. The reduction that justifies the security of our construction is tight enough to achieve practical security with reasonable key sizes.

[*]IMDEA Software Institute, Madrid, Spain. E-mail: gilles.barthe@imdea.org
[†]École Normale Supérieure, Paris, France. E-mail: david.pointcheval@ens.fr
[‡]Microsoft Research, Cambridge, UK. E-mail: santiago@microsoft.com

Contents

1 Introduction

More than three decades after its inception by Rivest, Shamir and Adleman, the RSA algorithm [39] has become a recommendation of several international standards for public-key cryptography and is widely used in practical cryptosystems. In order to achieve the level of security mandated by modern cryptography, RSA is used for instantiating cryptographic systems based on trapdoor one-way functions, rather than as a standalone primitive. The prevailing definition of security for public-key encryption schemes is the notion of ciphertext indistinguishability against chosen-ciphertext attacks (IND-CCA) [38], which requires that no efficient adversary with access to a decryption oracle be able to distinguish between the ciphertexts resulting from encrypting two messages of its choice. Since IND-CCA security cannot be achieved by deterministic encryption algorithms like RSA, encryption systems adopt the *encode-then-encrypt* paradigm, in which a message is pre-processed and randomized before encryption. For instance, the PKCS standard recommends that the RSA algorithm be used together with the Optimal Asymmetric Encryption Padding [10] scheme (OAEP), a two-round Feistel construction due to Bellare and Rogaway. In OAEP, redundancy is added during the encoding phase with the goal of achieving plaintext-awareness, that is, of making infeasible for an adversary to obtain a valid ciphertext other than by encrypting a known plaintext. Although the formalization of plaintext-awareness has unveiled subtleties (see Section 6 for a brief discussion), it is an appealing notion satisfied by many prominent encryption schemes. Furthermore, plaintext-awareness is achieved by cryptographic transformations [25, 26, 35] that convert encryption schemes that are just semantically secure under chosen-plaintext attacks [28] into IND-CCA-secure schemes. As a consequence, it was a widespread belief that plaintext-awareness was necessary to achieve IND-CCA security. In 2003, Phan and Pointcheval [36] proved this intuition wrong, by proposing the first IND-CCA-secure encryption schemes without redundancy, both in the ideal-cipher model and the random oracle model. They showed that a trapdoor one-way permutation combined with a full-domain random permutation, in a similar way to the FDH signature scheme [11], suffice to build a redundancy-free IND-CCA-secure scheme. In addition, Phan and Pointcheval showed that a 3-round version of OAEP together with a partial-domain one-way permutation would not require redundancy, as in the classical OAEP construction [10, 27]. This result was later improved when it was shown that (full-domain) one-wayness on its own is actually enough to eliminate redundancy in a 3-round version of OAEP [37]. Abe et al. [2] construct a redundancy-free scheme based on a 4-round Feistel network that achieves optimal ciphertext overhead (but that imposes a minimal message size). This line of work was further developed in a series of papers, including [19, 32], in the context of identity-based encryption and DL-based cryptosystems.

In this paper, we revisit the problem of designing redundancy-free IND-CCA-secure schemes based on trapdoor one-way functions. Our starting point is the SAEP and SAEP+ padding schemes, put forward by Boneh [18] in 2001. SAEP and SAEP+ are basically one-round OAEP-like paddings, that when combined with the Rabin function and RSA with exponent 3, yield encryption schemes with efficient security reductions. We generalize Boneh's construction to an arbitrary trapdoor one-way function and we show that SAEP padding without redundancy, which we call ZAEP (Zero-Redundancy Asymmetric Encryption Padding), achieves IND-CCA security in the Random Oracle Model for a class of trapdoor one-way functions that satisfy two novel properties: Common Input Extractability (CIE), and Second Input Extractability (SIE). Informally, CIE allows us to efficiently extract the plaintexts and randomness from two different ciphertexts that share the same randomness, whereas SIE allows us to efficiently extract the plaintext from a ciphertext and its randomness—in both cases, without knowing the trapdoor to the underlying one-way function. Using Coppersmith algorithm [20], we then show that the original Rabin function and RSA with short exponent satisfy these two properties. We thus obtain two efficient encryption algorithms, that are well-suited to encapsulate AES keys at a very low cost, with classical RSA moduli, either under the integer factoring assumption or the RSA assumption with exponent 3.

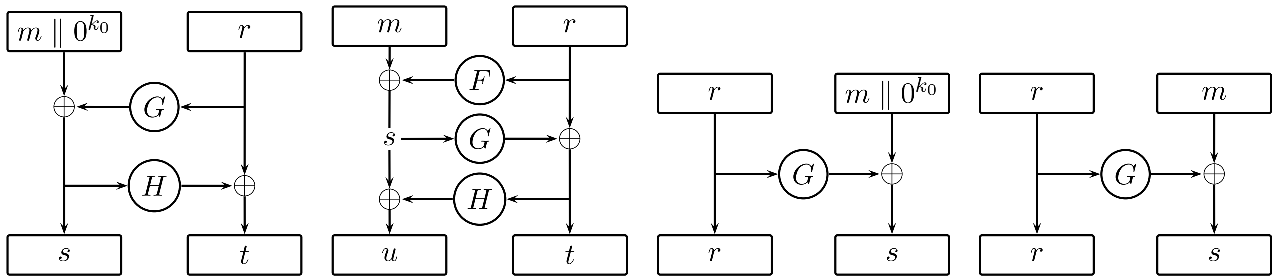
Our result is remarkable in two respects. First, ZAEP is surprisingly simple in comparison to the previous redundancy-free 3-round variant of OAEP that was shown to achieve IND-CCA security. Second, it constitutes the first application of verified security to a novel cryptographic construction. Specifically, we formally verify the security reduction (and the exact probability bound) of ZAEP using the EasyCrypt framework [4], which aims to make machine-checkable security proofs accessible to the working cryptographer by leveraging state-of-the-art methods and tools for program verification. Quite pleasingly, the functionalities and expressive power of EasyCrypt proved adequate for converting an incomplete and intuitive argument into a machine-checked proof. In less than a week, we were able to flesh out the details of the proof, including the new security assumptions, concrete security bound, and sequence of games, and to build a machine-checked proof. As further developed in Section 7, our work contributes to evidencing that, as anticipated by Halevi [29], computer-aided security proofs may become commonplace in the near future.

Organization of the paper We introduce the ZAEP redundancy-free scheme in Section 2 and present necessary background on verified security and the EasyCrypt framework in Section 3. We give an overview of the verified security reduction of ZAEP in Section 4 and discuss possible instantiations in Section 5. We conclude with a discussion on related work in Section 6, and an analysis of the significance of our results in Section 7. The EasyCrypt input file corresponding to the proof presented in Section 4 appears in the Appendix; all the infrastructure needed to machine-check this proof can be made available on request.

2 Redundancy-Free Encryption

In 1994, Bellare and Rogaway [10] proposed the padding scheme OAEP (see Fig. 1(a)), that in combination with a trapdoor permutation (e.g. RSA) yields an efficient encryption scheme. When encrypting using OAEP, a random value r is first expanded by a hash function G and then xor-ed with the redundancy-padded input message. The resulting value s is then hashed under an independent function H and the result xor-ed with r to obtain t. The ciphertext is computed by applying the permutation to the concatenation of s and t. OAEP was proved IND-CCA-secure by Fujisaki et al. [27] under the assumption that the underlying trapdoor permutation is partial-domain one-way. This is in general a stronger assumption than just one-wayness, but fortunately both assumptions are equivalent in particular for RSA. The reduction from the security of OAEP to the RSA problem is not tight for two reasons: (1) the generic reduction from OAEP security to the partial-domain one-wayness of the underlying permutation is itself not tight, and (2) the reduction from RSA partial-domain one-wayness to the RSA problem introduces an extra security gap. In order to obtain a direct reduction to the RSA problem (or the one-wayness of the underlying permutation), one needs to add a third round to the Feistel network used in OAEP [37]. Although this latter reduction is still not tight, the redundancy resulting from padding the input message can be removed without breaking the proof.

Boneh [18] showed that by exploiting Coppersmith algorithm [20], it is possible to shave off one round of OAEP without compromising security. Encryption in the resulting scheme, SAEP (see Fig. 1(c)), works by choosing a random value r, hashing it under a function G and xor-ing it with the message padded with a zero-bitstring of length k_0. The resulting value s is then concatenated with the random value r and fed to the RSA function. However, an efficient reduction is possible only if a small RSA public exponent is used, or if the Rabin function is used instead. The security reduction of SAEP is quite tight, but the redundancy



(a) OAEP (b) 3-round OAEP (c) SAEP (d) ZAEP

Figure 1: Asymmetric Encryption Paddings

introduced when padding the input message is essential and cannot be removed—as a by-product, SAEP achieves plaintext-awareness. We revisit SAEP with zero-length redundancy (i.e., letting $k_0 = 0$) and show that a reduction to the one-wayness of the underlying trapdoor permutation is still possible under additional (but achievable) assumptions.

2.1 A Novel Redundancy-Free Scheme

We recall the SAEP construction [18] with zero-length redundancy (see Fig. 1(d)). We use k to denote the length of the random value used during encryption and ℓ to denote the length of input messages. Let $(\mathcal{KG}_f, f, f^{-1})$ be a family of trapdoor one-way permutations on $\{0,1\}^n$, where $n = k + \ell$. For any pair of keys (pk, sk) output by the key generation algorithm $\mathcal{KG}_f$, $f_{pk}(\cdot)$ and $f_{sk}^{-1}(\cdot)$ are permutations on $\{0,1\}^n$ and inverses of each other. We model f_{pk} and f_{sk}^{-1} as two-input functions from $\{0,1\}^k \times \{0,1\}^\ell$ onto $\{0,1\}^n$. Let in addition $G : \{0,1\}^k \to \{0,1\}^\ell$ be a hash function, which we model as a random oracle in the reduction [9]. The ZAEP encryption scheme is composed of the triple of algorithms $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ defined as follows:

Key Generation $\mathcal{KG}$ is the same as the key generation algorithm $\mathcal{KG}_f$ of the underlying trapdoor permutation;

Encryption Given a public key pk and an input message $m \in \{0,1\}^\ell$, the encryption algorithm $\mathcal{E}_{pk}(m)$ chooses uniformly at random a value $r \in \{0,1\}^k$ and outputs the ciphertext $c = f_{pk}(r, G(r) \oplus m)$;

Decryption Given a secret key sk and a ciphertext c, the decryption algorithm $\mathcal{D}_{sk}(c)$ computes $(r, s) = f_{sk}^{-1}(c)$ and outputs $m = s \oplus G(r)$. No additional check is required because all ciphertexts are valid.

2.2 Adaptive Security of ZAEP

We recall the usual definitions of trapdoor one-way function and IND-CCA security for public-key encryption schemes.

Definition 1 (Trapdoor one-way function)**.** *Consider a family of trapdoor functions $(\mathcal{KG}, f, f^{-1})$ on $\{0,1\}^n$. The success probability $\mathbf{Succ}_f^{\mathsf{OW}}(\mathcal{I})$ of an algorithm $\mathcal{I}$ in inverting f_{pk} on a freshly generated public-key pk and a uniformly chosen input is defined as follows:*

$$\Pr\left[\begin{array}{l}(pk, sk) \leftarrow \mathcal{KG}(1^\eta);\\ x \xleftarrow{\$} \{0,1\}^n;\ x' \leftarrow \mathcal{I}(f_{pk}(x))\end{array} : f_{pk}(x) = f_{pk}(x')\right]$$

In an asymptotic setting, a family of trapdoor functions is one-way if this probability is negligible on the security parameter η for any efficient (probabilistic polynomial-time) algorithm $\mathcal{I}$.

Definition 2 (IND-CCA security)**.** *The advantage of an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the IND-CCA security of an asymmetric encryption scheme $\Pi = (\mathcal{KG}, \mathcal{E}, \mathcal{D})$, $\mathbf{Adv}^{\mathsf{CCA}}_{\Pi}(\mathcal{A})$, is defined as follows:*

$$\left| \Pr\left[\begin{array}{l} (pk, sk) \leftarrow \mathcal{KG}(1^\eta); \\ (m_0, m_1, \sigma) \leftarrow \mathcal{A}_1^{\mathcal{D}_{sk}(\cdot)}(pk); \\ b \overset{\$}{\leftarrow} \{0,1\};\ \boldsymbol{c}^* \leftarrow \mathcal{E}_{pk}(m_b); \\ b' \leftarrow \mathcal{A}_2^{\mathcal{D}_{sk}(\cdot \neq \boldsymbol{c}^*)}(\boldsymbol{c}^*, \sigma) \end{array} : b = b' \right] - \frac{1}{2} \right|$$

In both stages of the experiment the adversary has access to a decryption oracle, but in the second stage $\mathcal{A}_2$ cannot query for the decryption of the challenge ciphertext $\mathbf{c}^*$. In an asymptotic setting, Π is IND-CCA-secure if all efficient adversaries have a negligible advantage.

In order to prove the IND-CCA security of ZAEP, we require that the underlying trapdoor function satisfy the two properties defined below.

Definition 3 (Second-Input Extractability)**.** *A family of trapdoor functions $(\mathcal{KG}, f, f^{-1})$ satisfies SIE if there exists an efficient algorithm* sie *that given a public key pk, $c \in \{0,1\}^\ell$ and $r \in \{0,1\}^k$, outputs s if $c = f_{pk}(r, s)$ or $\perp$ otherwise.*

Observe that Second-Input Extractability collapses the distinction between one-wayness and partial one-wayness. If a family of one-way functions satisfies Second-Input Extractability, then it is also partial-domain one-way over its first input.

Definition 4 (Common-Input Extractability)**.** *A family of trapdoor functions $(\mathcal{KG}, f, f^{-1})$ satisfies CIE if there exists an efficient algorithm* cie *that given a public key pk and $c_1, c_2 \in \{0,1\}^\ell$, outputs (r, s_1, s_2) if $c_1 = f_{pk}(r, s_1)$, $c_2 = f_{pk}(r, s_2)$ and $s_1 \neq s_2$, or $\perp$ otherwise.*

Since we conduct our proof in a concrete security setting rather than in an asymptotic setting, and we prove exact probability and time bounds, we fix the security parameter and omit in the remainder. We prove the following security result for ZAEP.

Theorem 1 (Security of ZAEP)**.** *Let $(\mathcal{KG}, f, f^{-1})$ be a family of trapdoor permutations satisfying both SIE and CIE properties. Let $\mathcal{A}$ be an adversary against the IND-CCA security of ZAEP instantiated with $(\mathcal{KG}, f, f^{-1})$ that runs within time $t_\mathcal{A}$ and makes at most q_G queries to the random oracle G and at most $q_\mathcal{D}$ queries to the decryption oracle. Then, there exists an algorithm $\mathcal{I}$ running within time $t_\mathcal{I}$ such that*

$$t_\mathcal{I} \leq t_\mathcal{A} + 2q_\mathsf{G}q_\mathcal{D}\ t_\mathsf{sie} + q_\mathcal{D}^2\ t_\mathsf{cie}$$
$$\mathbf{Succ}^{\mathsf{OW}}_f(\mathcal{I}) \geq \mathbf{Adv}^{\mathsf{CCA}}_{\mathsf{ZAEP}}(\mathcal{A}) - \frac{q_\mathcal{D}}{2^n}$$

where t_cie (resp. t_sie) is an upper bound on the execution time of the algorithm cie *(resp.* sie*) for $(\mathcal{KG}, f, f^{-1})$.*

In Section 4 we give an overview of a machine-checked reductionist proof of the above theorem in EasyCrypt. We observe that while ZAEP can be cast as an instance of SAEP by setting the length of the padding $k_0 = 0$, our reduction is different from Boneh's reduction for SAEP [18]; in fact, Boneh's exact security bounds are meaningless as soon as k_0 is of the order of $\log(q_\mathcal{D})$.

3 A Primer on Verified Security

Verified security [4, 6] is an emerging approach to cryptographic proofs. While adhering to the principles and the methods of provable security, verified security takes the view that cryptographic proofs should be treated in a manner similar to high-integrity software, so that confidence in the design of a cryptographic system is no lower than confidence in the software systems that use it. Thus, verified security mandates that security proofs are built and validated using state-of-the-art technology in programming languages and verification.

EasyCrypt [4] is a recent realization of the verified security paradigm. As its predecessor CertiCrypt [6], it adopts a code-centric view of cryptography. Under this view, security assumptions and goals are formalized using probabilistic programs, also called *games*. Each game is a probabilistic imperative program composed of a main command and a collection of concrete procedures and adversaries. Moreover, the statements of the language include deterministic and probabilistic assignments, conditional statements and loops, as given by the following grammar:

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

where $\mathcal{V}$ is a set of variable identifiers, $\mathcal{P}$ a set of procedure names with a distinguished class of abstract procedures used to model adversaries, $\mathcal{E}$ is a set of expressions, and $\mathcal{DE}$ is a set of distribution expressions. The latter are expressions that evaluate to distributions from which values can be sampled; for the purpose of this paper, we only need to consider uniform distributions over bitstrings.

Programs in EasyCrypt are given a denotational semantics, that maps initial memories to sub-distributions over final memories, where a memory is a (well-typed) mapping from variables to values. We let $\Pr[c, m : A]$ denote the probability of an event A in the sub-distribution induced by executing the program c on some initial memory m, which we omit when it is not relevant. For additional details on the semantics, we refer the reader to [6].

As envisioned by Halevi [29] and Bellare and Rogaway [12], this code-centric view of cryptographic proofs leads to statements that are amenable to verification using programming language techniques. EasyCrypt captures common reasoning patterns in cryptographic proofs by means of a probabilistic relational Hoare Logic (pRHL). Judgments in pRHL are of the form

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

where c_1 and c_2 are probabilistic programs, and Ψ and Φ, respectively called the pre-condition and the post-condition, are relations over program states. We represent these relations as first-order formulae defined by the grammar:

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \to \Phi \mid \forall x.\ \Phi \mid \exists x.\ \Phi$$

where e stands for a Boolean expression over logical variables and program variables tagged with either $\langle 1 \rangle$ or $\langle 2 \rangle$ to denote their interpretation in the left or right-hand side program, respectively. We write $e\langle i \rangle$

for the expression e in which all program variables are tagged with $\langle i \rangle$. A relational formula is interpreted as a relation on program memories. For example, the formula $x\langle 1\rangle + 1 \leq y\langle 2\rangle$ is interpreted as the relation

$$R = \{(m_1, m_2) \mid m_1(x) + 1 \leq m_2(y)\}$$

There are two complementary means to establish the validity of a pRHL judgment. Firstly, the user can apply interactively atomic rules and semantics-preserving program transformations. Secondly, the user can invoke an automated procedure that given a logical judgment involving loop-free closed programs, computes a set of sufficient conditions for its validity, known as verification conditions. In the presence of loops or adversarial code, EasyCrypt requires the user to provide the necessary annotations. The outstanding feature of this procedure, and the key to its effectiveness, is that verification conditions are expressed as first-order formulae, without any mention of probability, and thus can be discharged automatically using off-the-shelf SMT solvers and theorem provers.

As security properties are typically expressed in terms of probability of events, and not as pRHL judgments, EasyCrypt provides mechanisms to derive from a valid judgment

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

inequalities of the form

$$\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]\,(+\Pr[c_2, m_2 : F])$$

for events A, B and F that are suitably related to the post-condition Φ. The mechanisms are described more precisely by the next two lemmas.

Lemma 2 (Probability Lemma). *Let c_1 and c_2 be two games and A and B be events such that*

$$\models c_1 \sim c_2 : \Psi \Rightarrow A\langle 1\rangle \rightarrow B\langle 2\rangle$$

For every pair of memories m_1, m_2 such that $m_1\ \Psi\ m_2$, we have

$$\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]$$

Lemma 3 (Shoup's Fundamental Lemma). *Let c_1 and c_2 be two games and $A, B,$ and F be events such that*

$$\models c_1 \sim c_2 : \Psi \Rightarrow (F\langle 1\rangle \leftrightarrow F\langle 2\rangle) \land (\neg F\langle 1\rangle \rightarrow A\langle 1\rangle \rightarrow B\langle 2\rangle)$$

Then, for every pair of memories m_1, m_2 such that $m_1\ \Psi\ m_2$, we have

$$\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B] + \Pr[c_2, m_2 : F]$$

Moreover, EasyCrypt includes support for applying probability laws (e.g. the union bound) and computing the probability of events. The proof of ZAEP relies on two main rules. The first one states that an adversary has probability $\frac{1}{2}$ of guessing a bit b independent from its view; independence is captured by proving that sampling the bit b after the adversary returns its guess does not change the semantics of the game. The second rule allows to upper bound the probability that a uniformly sampled value belongs to a list of bounded length. For instance, if L is a list of values in A of length at most q and x is a value sampled independently and uniformly over A, the probability that x belongs to L is upper bounded by $q/|A|$.

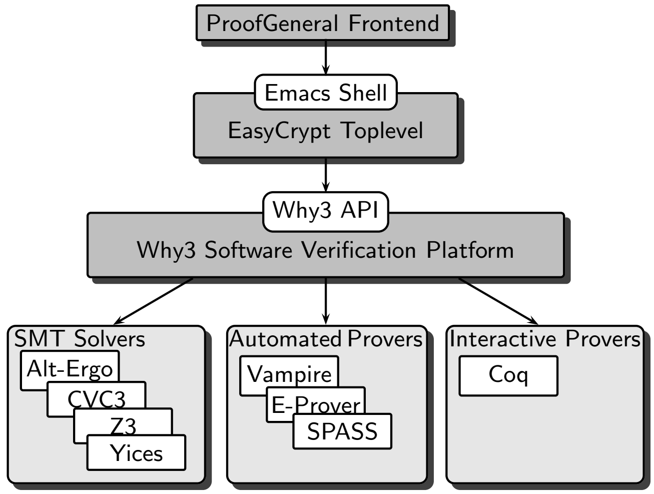


Figure 2: Overview of workflow in EasyCrypt

3.1 User Perspective

Building a cryptographic proof in EasyCrypt is a process that involves the following tasks:

- Defining a logical context, including declarations of types, constants and operators, axioms and derived lemmas. Declarations allow users to extend the core language, while axioms allow to give the extension a meaning. Derived lemmas are intermediary results proved from axioms, and are used to drive SMT solvers and automated provers.

- Defining games, including the initial experiment encoding the security property to be proved, intermediate games, and a number of final games, which either correspond to a security assumption or allow to directly compute a bound on the probability of some event.

- Proving logical judgments that establish equivalences between games. This may be done fully automatically, with the help of hints from the user in the form of relational invariants, or interactively using basic tactics and automated strategies. In order to benefit from existing technology and target multiple verification tools, verification conditions are generated in the intermediate language of the Why3 Software Verification Platform [17] and then translated to individual provers to check their validity.

- Deriving inequalities between probabilities of events in games, either by using previously proven logical judgments or by direct computation.

Although the above tasks can be carried out strictly in the order described, one can conveniently interleave them as in informal game-based proofs. To ease this process, EasyCrypt provides an interactive user-interface as an instance of ProofGeneral, a generic Emacs-based frontend for proof-assistants. Figure 2 gives an overview of the workflow in the framework.

Game CCA :
$\boldsymbol{L}_G \leftarrow$ nil; $\boldsymbol{c}^*_{\mathbf{def}} \leftarrow$ false; $\boldsymbol{q} \leftarrow 0$;
$(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}()$;
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk)$;
$b \stackrel{\$}{\leftarrow} \{0,1\}$;
$\boldsymbol{c}^* \leftarrow \mathcal{E}_{pk}(m_b)$;
$\boldsymbol{c}^*_{\mathbf{def}} \leftarrow$ true;
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma)$;
return $(b = b')$

Oracle $G(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
  $\boldsymbol{L}_G[x] \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;
return $\boldsymbol{L}_G[x]$

Oracle $\mathcal{D}(c)$:
if $\boldsymbol{q} < q_{\mathcal{D}} \wedge \neg(\boldsymbol{c}^*_{\mathbf{def}} \wedge c = \boldsymbol{c}^*)$ then
  $\boldsymbol{q} \leftarrow \boldsymbol{q} + 1$;
  $(r, s) \leftarrow f^{-1}_{\boldsymbol{sk}}(c)$;
  $g \leftarrow G(r)$;
  return $g \oplus s$
else return $\perp$

Game OW :
$(pk, sk) \leftarrow \mathcal{KG}()$;
$z \stackrel{\$}{\leftarrow} \{0,1\}^{k+\ell}$;
$(x, y) \leftarrow \mathcal{I}(pk, f_{pk}(z))$;
return $(f_{pk}(x, y) = f_{pk}(z))$

Adversary $\mathcal{I}(pk, z)$:
$\boldsymbol{L}_G, \boldsymbol{L}_{\mathcal{D}} \leftarrow$ nil; $\boldsymbol{c}^*_{\mathbf{def}} \leftarrow$ false; $\boldsymbol{q} \leftarrow 0$;
$\boldsymbol{c}^* \leftarrow z$; $\boldsymbol{pk} \leftarrow pk$;
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk)$;
$\boldsymbol{c}^*_{\mathbf{def}} \leftarrow$ true;
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma)$;
$r \leftarrow$ find $r \in \mathsf{dom}(\boldsymbol{L}_G)$. $\mathsf{sie}_{\boldsymbol{pk}}(\boldsymbol{c}^*, r) \neq \perp$;
if $r \neq \perp$ then return $(r, \mathsf{sie}_{\boldsymbol{pk}}(\boldsymbol{c}^*, r))$;
else
  $c \leftarrow$ find $c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}})$. $\mathsf{cie}_{\boldsymbol{pk}}(\boldsymbol{c}^*, c) \neq \perp$;
  if $c \neq \perp$ then
    $(r, s, t) \leftarrow \mathsf{cie}_{\boldsymbol{pk}}(\boldsymbol{c}^*, c)$; return (r, s)
  else return $\perp$

Oracle $G(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
  $c \leftarrow$ find $c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}})$. $\mathsf{sie}_{\boldsymbol{pk}}(c, x) \neq \perp$;
  if $c \neq \perp$ then
    $\boldsymbol{L}_G[x] \leftarrow \boldsymbol{L}_{\mathcal{D}}[c] \oplus \mathsf{sie}_{\boldsymbol{pk}}(c, x)$;
  else
    $\boldsymbol{L}_G[x] \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;
return $\boldsymbol{L}_G[x]$

Oracle $\mathcal{D}(c)$:
if $\boldsymbol{q} < q_{\mathcal{D}} \wedge \neg(\boldsymbol{c}^*_{\mathbf{def}} \wedge c = \boldsymbol{c}^*)$ then
  $\boldsymbol{q} \leftarrow \boldsymbol{q} + 1$;
  $r \leftarrow$ find $r \in \mathsf{dom}(\boldsymbol{L}_G)$. $\mathsf{sie}_{\boldsymbol{pk}}(c, r) \neq \perp$;
  if $r \neq \perp$ then return $\boldsymbol{L}_G[r] \oplus \mathsf{sie}_{\boldsymbol{pk}}(c, r)$
  else
    if $c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}})$ then return $\boldsymbol{L}_{\mathcal{D}}[c]$
    else
      $c' \leftarrow$ find $c' \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}})$. $\mathsf{cie}_{\boldsymbol{pk}}(c, c') \neq \perp$;
      if $c' \neq \perp$ then
        $(r, s, t) \leftarrow \mathsf{cie}_{\boldsymbol{pk}}(c, c')$;
        return $\boldsymbol{L}_{\mathcal{D}}[c'] \oplus s \oplus t$;
      else
        if $\boldsymbol{c}^*_{\mathbf{def}} \wedge \mathsf{cie}_{\boldsymbol{pk}}(c, \boldsymbol{c}^*) \neq \perp$ then
          $(r, s, t) \leftarrow \mathsf{cie}_{\boldsymbol{pk}}(c, \boldsymbol{c}^*)$;
          $\boldsymbol{L}_G[r] \stackrel{\$}{\leftarrow} \{0,1\}^\ell$; return $\boldsymbol{L}_G[r] \oplus s$;
        else
          $\boldsymbol{L}_{\mathcal{D}}[c] \stackrel{\$}{\leftarrow} \{0,1\}^\ell$; return $\boldsymbol{L}_{\mathcal{D}}[c]$
else return $\perp$

Figure 3: Initial IND-CCA game and reduction to the problem of inverting the underlying permutation

4 Security Proof

We overview the proof of Theorem 1 in EasyCrypt. The proof is organized as a sequence of games starting from game CCA, that encodes an adaptive chosen-ciphertext attack against ZAEP for an arbitrary adversary $\mathcal{A}$, and ending in game OW, that encodes the reduction to the one-wayness of the underlying trapdoor permutation. These two games are shown in Figure 3; the rest of the games are shown in Figures 4 and 5. Games are shown alongside the oracles made available to adversary $\mathcal{A}$ and global variables are typeset in boldface.

Formalizing the security proof of ZAEP in EasyCrypt required providing an appropriate axiomatization of the underlying trapdoor permutation and the SIE and CIE properties. We extended the expression language with the following operators corresponding to the permutation f, its inverse, and algorithms sie and cie:

op `f` : $(\texttt{pkey}, \{0,1\}^k \times \{0,1\}^\ell) \to \{0,1\}^k \times \{0,1\}^\ell$
op `finv` : $(\texttt{skey}, \{0,1\}^k \times \{0,1\}^\ell) \to \{0,1\}^k \times \{0,1\}^\ell$
op `sie` : $(\texttt{pkey}, \{0,1\}^k \times \{0,1\}^\ell, \{0,1\}^k) \to \{0,1\}^\ell$ `option`
op `cie` : $(\texttt{pkey}, \{0,1\}^k \times \{0,1\}^\ell, \{0,1\}^k \times \{0,1\}^\ell) \to (\{0,1\}^k \times \{0,1\}^\ell \times \{0,1\}^\ell)$ `option`

We gave these operators a meaning by introducing their specifications as axioms; for instance, the operator

Game $\boxed{\mathsf{G}_1}$ $\boxed{\mathsf{G}_2}$:	**Oracle** $G(x)$:	**Oracle** $\mathcal{D}(c)$:
$\boldsymbol{L}_G \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$ $\mathbf{bad} \leftarrow \mathsf{false};\ \boldsymbol{r}^* \overset{\$}{\leftarrow} \{0,1\}^k;$ $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}();$ $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);\ b \overset{\$}{\leftarrow} \{0,1\};$ $\mathsf{if}\ \boldsymbol{r}^* \notin \mathsf{dom}(\boldsymbol{L}_G)\ \mathsf{then}$ $\quad \boldsymbol{g}^* \overset{\$}{\leftarrow} \{0,1\}^\ell;\ \boxed{\boldsymbol{L}_G[\boldsymbol{r}^*] \leftarrow \boldsymbol{g}^*;}$ else $\quad \mathbf{bad} \leftarrow \mathsf{true};$ $\quad \boxed{\boldsymbol{g}^* \leftarrow \boldsymbol{L}_G[\boldsymbol{r}^*];}\ \boxed{\boldsymbol{g}^* \overset{\$}{\leftarrow} \{0,1\}^\ell;}$ $\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{g}^* \oplus m_b);\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$ $b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$ $\mathsf{return}\ (b = b')$	$\mathsf{if}\ x = \boldsymbol{r}^*\ \mathsf{then}\ \mathbf{bad} \leftarrow \mathsf{true};$ $\mathsf{if}\ x \notin \mathsf{dom}(\boldsymbol{L}_G)\ \mathsf{then}$ $\quad \boldsymbol{L}_G[x] \overset{\$}{\leftarrow} \{0,1\}^\ell;$ $\mathsf{return}\ \boldsymbol{L}_G[x]$	$\mathsf{if}\ \boldsymbol{q} < q_{\mathcal{D}} \wedge \neg(\boldsymbol{c}^*_{\mathbf{def}} \wedge c = \boldsymbol{c}^*)\ \mathsf{then}$ $\quad \boldsymbol{q} \leftarrow \boldsymbol{q} + 1;$ $\quad (r, s) \leftarrow f^{-1}_{\boldsymbol{sk}}(c);$ $\quad g \leftarrow G(r);$ $\quad \mathsf{return}\ g \oplus s$ $\mathsf{else\ return}\ \perp$

Game G_3 :	**Oracle** $G(x)$:	**Oracle** $\mathcal{D}(c)$:
$\boldsymbol{L}_G \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$ $\mathbf{bad} \leftarrow \mathsf{false};\ \boldsymbol{r}^* \overset{\$}{\leftarrow} \{0,1\}^k;$ $(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}();$ $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);$ $\mathsf{if}\ \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G)\ \mathsf{then}\ \mathbf{bad} \leftarrow \mathsf{true};$ $\boldsymbol{s}^* \overset{\$}{\leftarrow} \{0,1\}^\ell;$ $\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*);\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$ $b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$ $b \overset{\$}{\leftarrow} \{0,1\};$ $\mathsf{return}\ (b = b')$	$\mathsf{if}\ x = \boldsymbol{r}^*\ \mathsf{then}\ \mathbf{bad} \leftarrow \mathsf{true};$ $\mathsf{if}\ x \notin \mathsf{dom}(\boldsymbol{L}_G)\ \mathsf{then}$ $\quad \boldsymbol{L}_G[x] \overset{\$}{\leftarrow} \{0,1\}^\ell;$ $\mathsf{return}\ \boldsymbol{L}_G[x]$	$\mathsf{if}\ \boldsymbol{q} < q_{\mathcal{D}} \wedge \neg(\boldsymbol{c}^*_{\mathbf{def}} \wedge c = \boldsymbol{c}^*)\ \mathsf{then}$ $\quad \boldsymbol{q} \leftarrow \boldsymbol{q} + 1;$ $\quad (r, s) \leftarrow f^{-1}_{\boldsymbol{sk}}(c);$ $\quad g \leftarrow G(r);$ $\quad \mathsf{return}\ g \oplus s$ $\mathsf{else\ return}\ \perp$

Figure 4: Sequence of games in the proof of ZAEP. Fragments of code displayed inside a box appear only in the game whose name is surrounded by the matching box.

`sie` is specified as follows:

$$\begin{array}{l}\textbf{axiom}\ \mathtt{sie\_spec}: \\ \quad \forall\ (\mathtt{pk}:\mathtt{pkey}, \mathtt{sk}:\mathtt{skey}),\ \mathtt{key\_pair}(\mathtt{pk},\mathtt{sk}) \Longrightarrow \\ \qquad \forall\ (\mathtt{c}:\{0,1\}^k \times \{0,1\}^\ell, \mathtt{r}:\{0,1\}^k, \mathtt{s}:\{0,1\}^\ell),\ \mathtt{sie}(\mathtt{pk},\mathtt{c},\mathtt{r}) = \mathtt{Some}(\mathtt{s}) \Longleftrightarrow \mathtt{c} = \mathtt{f}(\mathtt{pk},(\mathtt{r},\mathtt{s}))\end{array}$$

Verification conditions generated during the proof are first-order formulae over a mixture of theories: e.g. finite maps, integer arithmetic, exclusive-or, and the above axiomatization of the SIE and CIE solvers. All verification conditions are discharged automatically using the CVC3 and Alt-Ergo SMT solvers.

The proof itself begins by transforming the initial CCA game into game G_1, where we inline the encryption of the challenge ciphertext and eagerly sample the random value $\boldsymbol{r}^*$ used. We also introduce a Boolean flag **bad** that is set to true whenever $\boldsymbol{r}^*$ would be appear as a query to G in the CCA experiment. All these changes are semantics-preserving w.r.t. to the event $b = b'$ and thus we have

$$\Pr\left[\mathsf{CCA} : b = b'\right] = \Pr\left[\mathsf{G}_1 : b = b'\right]$$

Game G_2 behaves identically to game G_1 except that the value of $G(\boldsymbol{r}^*)$ used to mask the plaintext of the challenge ciphertext is always chosen at random, regardless of whether it has been queried by the adversary during the first stage of the experiment. Subsequent queries to $G(\boldsymbol{r}^*)$ are also answered with a fresh random value. This only makes a difference if the flag **bad** is set, and applying Lemma 3, we obtain:

$$\left|\Pr\left[\mathsf{G}_1 : b = b'\right] - \Pr\left[\mathsf{G}_2 : b = b'\right]\right| \le \Pr\left[\mathsf{G}_2 : \mathbf{bad}\right]$$

```
Game G_4 :
  L_G, L'_G ← nil;  c*_def ← false;  q ← 0;
  r* ←$ {0,1}^k;
  s* ←$ {0,1}^ℓ;
  c* ← f_pk(r*, s*);
  (pk, sk) ← KG();
  (m_0, m_1, σ) ← A_1(pk);
  c*_def ← true;
  b' ← A_2(c*, σ);
  return true
```

```
Oracle G(x) :
  if x ∉ dom(L_G) then
    if x ∉ dom(L'_G) then
      L_G[x] ←$ {0,1}^ℓ;
    else
      L_G[x] ← L'_G[x];
  return L_G[x]
```

```
Oracle D(c) :
  if q < q_D ∧ ¬(c*_def ∧ c = c*) then
    q ← q + 1;
    r ← find r ∈ dom(L_G). sie_pk(c, r) ≠ ⊥;
    if r ≠ ⊥ then return L_G[r] ⊕ sie_pk(c, r)
    else
      r ← find r ∈ dom(L'_G). sie_pk(c, r) ≠ ⊥;
      if r ≠ ⊥ then return L'_G[r] ⊕ sie_pk(c, r)
      else
        if c*_def ∧ cie_pk(c, c*) ≠ ⊥ then
          (r, s, t) ← cie_pk(c, c*);
          L_G[r] ←$ {0,1}^ℓ;  return L_G[r] ⊕ s
        else
          (r, s) ← f_sk^{-1}(c);
          m ←$ {0,1}^ℓ;  L'_G[r] ← m ⊕ s;
          return m;
  else return ⊥
```

```
Game G_5 :
  L_G, L_D ← nil;  c*_def ← false;  q ← 0;
  r* ←$ {0,1}^k;
  s* ←$ {0,1}^ℓ;
  c* ← f_pk(r*, s*);
  (pk, sk) ← KG();
  (m_0, m_1, σ) ← A_1(pk);
  c*_def ← true;
  b' ← A_2(c*, σ);
  return true
```

```
Oracle G(x) :
  if x ∉ dom(L_G) then
    c ← find c ∈ dom(L_D). sie_pk(c, x) ≠ ⊥;
    if c ≠ ⊥ then
      L_G[x] ← L_D[c] ⊕ sie_pk(c, x);
    else
      L_G[x] ←$ {0,1}^ℓ;
  return L_G[x]
```

```
Oracle D(c) :
  if q < q_D ∧ ¬(c*_def ∧ c = c*) then
    q ← q + 1;
    r ← find r ∈ dom(L_G). sie_pk(c, r) ≠ ⊥;
    if r ≠ ⊥ then return L_G[r] ⊕ sie_pk(c, r)
    else
      if c ∈ dom(L_D) then return L_D[c]
      else
        c' ← find c' ∈ dom(L_D). cie_pk(c, c') ≠ ⊥;
        if c' ≠ ⊥ then
          (r, s, t) ← cie_pk(c, c');
          return L_D[c'] ⊕ s ⊕ t;
        else
          if c*_def ∧ cie_pk(c, c*) ≠ ⊥ then
            (r, s, t) ← cie_pk(c, c*);
            L_G[r] ←$ {0,1}^ℓ;  return L_G[r] ⊕ s;
          else
            L_D[c] ←$ {0,1}^ℓ;  return L_D[c]
  else return ⊥
```

Figure 5: Sequence of games in the proof of ZAEP.

In game G_3 we remove the dependency of the adversary's output on the hidden bit b by applying a semantics-preserving transformation known as *optimistic sampling*. Instead of of sampling $\boldsymbol{g}^*$ at random and computing the challenge ciphertext $\boldsymbol{c}^*$ as $f_{pk}(\boldsymbol{r}^*, \boldsymbol{g}^* \oplus m_b)$, we sample directly a value $\boldsymbol{s}^*$ at random and compute $\boldsymbol{c}^*$ as $f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*)$, defining $\boldsymbol{g}^*$ as $\boldsymbol{s}^* \oplus m_b$. Once this is done, and since $\boldsymbol{g}^*$ is no longer used elsewhere in the game, we can drop its definition as dead-code and postpone sampling b to the end of the game, making it trivially independent of b'. We have

$$\Pr\left[\mathsf{G}_2 : b = b'\right] = \Pr\left[\mathsf{G}_3 : b = b'\right] = \frac{1}{2}$$
$$\Pr\left[\mathsf{G}_2 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_3 : \mathbf{bad}\right]$$

In game G_4, instead of always using f^{-1} to compute the pre-image (r, s) of an input c in the decryption oracle, we use the sie and cie algorithms to compute it when possible from previous queries made by the adversary. We can do this in two cases:

1. when r appeared before in a query to oracle G, using algorithm sie to obtain the second input s;

2. when $r = \boldsymbol{r}^*$, using algorithm cie to compute s from $\boldsymbol{c}^*$.

When neither of these two cases occur, we use f^{-1} and the secret key to invert c and obtain (r, s). Rather than sampling a fresh value for $G(r)$, we apply once more the optimistic sampling transformation to sample a response m at random and define $G(r)$ as $m \oplus s$. We store values of $G(r)$ computed in this fashion in a different map $\boldsymbol{L}'_G$. We prove the following relational invariant between G_3 and G_4, which allows to characterize the event **bad** of G_3 in terms of the variables of G_4:

$$\mathbf{bad}\langle 1\rangle \iff (\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}'_G))\langle 2\rangle$$

To prove this, we have to first show that the simulation of the decryption oracle using algorithms cie and sie in G_4 is consistent with the view of the adversary in G_3. We do this by establishing that the following is a relational invariant between the implementations of $\mathcal{D}$ in G_3 and G_4:

$$\begin{array}{l}
(\boldsymbol{r}^*, \boldsymbol{s}^*, \boldsymbol{c}^*_{\mathbf{def}}, \boldsymbol{q})\langle 1\rangle = (\boldsymbol{r}^*, \boldsymbol{s}^*, \boldsymbol{c}^*_{\mathbf{def}}, \boldsymbol{q})\langle 2\rangle \ \land \\
(\boldsymbol{c}^* = f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*))\langle 2\rangle \ \land \\
\mathbf{bad}\langle 1\rangle \iff (\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}'_G))\langle 2\rangle \ \land \\
(\forall x \in \mathsf{dom}(\boldsymbol{L}_G\langle 2\rangle).x \in \mathsf{dom}(\boldsymbol{L}_G\langle 1\rangle) \land \boldsymbol{L}_G\langle 1\rangle[x] = \boldsymbol{L}_G\langle 2\rangle[x]) \ \land \\
(\forall x \in \mathsf{dom}(\boldsymbol{L}_G\langle 1\rangle).x \notin \mathsf{dom}(\boldsymbol{L}_G\langle 2\rangle) \to \boldsymbol{L}_G\langle 1\rangle[x] = \boldsymbol{L}'_G\langle 2\rangle[x]) \ \land \\
(\forall x.\ x \in \mathsf{dom}(\boldsymbol{L}_G\langle 1\rangle) \leftrightarrow (x \in \mathsf{dom}(\boldsymbol{L}_G) \lor x \in \mathsf{dom}(\boldsymbol{L}'_G))\langle 2\rangle
\end{array}$$

We have hence that

$$\Pr\left[\mathsf{G}_3 : \mathbf{bad}\right] = \Pr\left[\mathsf{G}_4 : \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}'_G)\right]$$

In game G_5 we finally eliminate every reference to f^{-1} from the decryption oracle. We do this by replacing the map $\boldsymbol{L}'_G$ with a map $\boldsymbol{L}_{\mathcal{D}}$ in where we store ciphertexts that implicitly define values of $G(r)$. We reformulate the simulation of the decryption oracle using this map instead of $\boldsymbol{L}'_G$, by proving the following invariant between the implementations of $\mathcal{D}$ in G_4 and G_5:

$$\begin{array}{l}
(\boldsymbol{L}_G, \boldsymbol{c}^*, \boldsymbol{c}^*_{\mathbf{def}}, \boldsymbol{q})\langle 1\rangle = (\boldsymbol{L}_G, \boldsymbol{c}^*, \boldsymbol{c}^*_{\mathbf{def}}, \boldsymbol{q})\langle 2\rangle \ \land \\
(\forall c.\ (\forall r \in \mathsf{dom}(\boldsymbol{L}'_G).\ \mathsf{sie}_{\boldsymbol{pk}}(c, r) = \bot)\langle 1\rangle \leftrightarrow (\forall c' \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{cie}_{\boldsymbol{pk}}(c, c') = \bot \land c \notin \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}))\langle 2\rangle \ \land \\
(\forall r.\ r \notin \mathsf{dom}(\boldsymbol{L}'_G\langle 1\rangle) \leftrightarrow (\forall c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{sie}_{\boldsymbol{pk}}(c, r) = \bot)\langle 2\rangle \land \\
(\forall c.\ \mathsf{let}\ (r, s) = f^{-1}_{\boldsymbol{sk}}(c)\ \mathsf{in} c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}))\langle 2\rangle \to r \in \mathsf{dom}(\boldsymbol{L}'_G\langle 1\rangle) \land \boldsymbol{L}'_G\langle 1\rangle[r] = s \oplus \boldsymbol{L}_{\mathcal{D}}\langle 2\rangle[c])
\end{array}$$

We then prove the following relational invariant between G_4 and G_5:

$$\begin{array}{l}
(\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}'_G))\langle 1\rangle \to \\
(\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \exists c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{cie}_{pk}(c, \boldsymbol{c}^*) \neq \bot)\langle 2\rangle
\end{array}$$

From which we obtain

$$\begin{array}{l}
\Pr\left[\mathsf{G}_4 : \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}'_G)\right] \leq \\
\Pr\left[\mathsf{G}_5 : \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \exists c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{cie}_{pk}(c, \boldsymbol{c}^*) \neq \bot\right]
\end{array}$$

We can finally write an inverter $\mathcal{I}$ against the one-wayness of the underlying trapdoor permutation that uses the map $\boldsymbol{L}_{\mathcal{D}}$ in the previous game to perfectly simulate the decryption oracle for the IND-CCA adversary $\mathcal{A}$. However, the inverter $\mathcal{I}$ only succeeds if $\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G)$:

$$\begin{array}{l}
\Pr\left[\mathsf{G}_5 : \boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G) \lor \exists c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{cie}_{pk}(c, \boldsymbol{c}^*) \neq \bot\right] \leq \\
\Pr\left[\mathsf{OW} : f_{pk}(x, y) = f_{pk}(z)\right] + \Pr\left[\mathsf{OW} : \boldsymbol{c}^* \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}})\right]
\end{array}$$

We bound the second term on the right-hand side of the above inequality by $q_{\mathcal{D}}/2^n$ using a short sequence of games that we omit. Putting all the above results together, we conclude:

$$\left|\Pr\left[\mathsf{CCA} : b = b'\right] - \frac{1}{2}\right| \le \Pr\left[\mathsf{OW} : f_{pk}(x, y) = f_{pk}(z)\right] + \frac{q_{\mathcal{D}}}{2^n}$$

The execution time of $t_{\mathcal{I}}$ can be bound by inspecting the formulation of the inverter $\mathcal{I}$ in game OW:

- Each simulated query to G requires at most $q_{\mathcal{D}}$ evaluations of algorithm sie;
- Each simulated query to $\mathcal{D}$ requires at most q_{G} evaluations of algorithm sie and at most $q_{\mathcal{D}}$ evaluations of algorithm cie;
- When the simulation finishes, the inverter $\mathcal{I}$ requires at most q_{G} evaluations of algorithm sie and at most $q_{\mathcal{D}} + 1$ evaluations of algorithm cie to find the inverse of its challenge.

Thus

$$t_{\mathcal{I}} \le t_{\mathcal{A}} + 2q_{\mathsf{G}}q_{\mathcal{D}}\ t_{\mathsf{sie}} + q_{\mathcal{D}}^2\ t_{\mathsf{cie}} + q_{\mathsf{G}}\ t_{\mathsf{sie}} + (q_{\mathcal{D}} + 1)\ t_{\mathsf{cie}}$$

The last two terms are negligible w.r.t. the rest and can be safely ignored.

5 Instantiations

In this section, we show that both the Rabin function and RSA with small exponent satisfy the properties required for the security reduction of ZAEP. Moreover, we provide a practical evaluation of both instantiations of ZAEP and a comparison to other redundancy-free encryption schemes. Our proofs are inspired by [18] and rely on Coppersmith algorithm to find small integer roots of polynomials [20]:

Theorem 4 (Coppersmith method)**.** *Let $p(X)$ be a monic integer polynomial of degree d and N a positive integer of unknown factorization. In time polynomial in $\log(N)$ and d, using Coppersmith algorithm one can find all integer solutions x_0 to $p(x_0) = 0 \bmod N$ with $|x_0| < N^{1/d}$.*

We denote by $t_{C(N,d)}$ an upper bound on the running time of the above method for finding all roots modulo N of a polynomial of degree d.

5.1 Short Exponent RSA

For an n-bit RSA modulus $N = pq$, the function

$$\mathsf{RSA}[N, e] : x \mapsto x^e \bmod N$$

is a well-known trapdoor one-way permutation on $\mathbb{Z}_N^*$ for any exponent e coprime to $\varphi(N)$. For any non-negative $\ell \le n$, an element $x \in \mathbb{Z}_N^*$ can be uniquely represented as $r \times 2^\ell + s$, where $s \in \{0,1\}^\ell$ and $r \in \{0,1\}^{n-\ell}$. We can thus express the RSA function as a function of two arguments:

$$\mathsf{RSA}[N, e] : (r, s) \mapsto (r \times 2^\ell + s)^e \bmod N$$

We denote by RSA-ZAEP the encryption scheme resulting from instantiating ZAEP with this function.

Second-Input Extractability Given an output c of $\mathsf{RSA}[N, e]$ and a tentative value r, the Second-Input Extraction problem boils down to solving $p(X) = 0 \bmod N$ for $p(X) = c - (r \times 2^\ell + X)^e \bmod N$ with the additional constraint $|X| < 2^\ell$. The Coppersmith method finds the root s (the second input to the function when r is the correct first input) when $2^\ell < N^{1/e}$, or equivalently, when $\ell < n/e$. We thus have an efficient sie algorithm that executes within time $t_{\mathsf{sie}} \leq t_{C(N,e)}$.

Common-Input Extractability Given two different outputs c_1 and c_2 of $\mathsf{RSA}[N, e]$, the Common-Input Extraction problem for $\mathsf{RSA}[N, e]$ consists in finding r, s_1 and s_2 such that $c_1 = (r \times 2^\ell + s_1)^e \bmod N$ and $c_2 = (r \times 2^\ell + s_2)^e \bmod N$, if they exist. Let us consider the two polynomials

$$p_1(X, \Delta) = c_1 - X^e \bmod N$$
$$p_2(X, \Delta) = c_2 - (X + \Delta)^e \bmod N$$

These polynomials should be equal to zero for the correct values $x = r \times 2^\ell + s_1 \bmod N$ for X and $\delta = s_2 - s_1 \bmod N$ for Δ. Therefore, the resultant polynomial $R(\Delta)$ of p_1 and p_2 in X, which is the determinant of the $2e \times 2e$ Sylvester Matrix associated to the polynomials p_1 and p_2 in the variable X, and thus with coefficients that are polynomials in Δ (of degree 0 for the coefficients of p_1, but of degree up to e for the coefficients of p_2), is a polynomial with $\delta = s_2 - s_1$ as a root. Due to the specific form of the matrix, $R(\Delta)$ is of degree at most e^2 modulo N, and the Coppersmith method finds the root δ provided $2^\ell < N^{1/e^2}$ or equivalently, when $\ell < n/e^2$. Once this root is known, we can focus on the monic polynomials $p_1(X) = c_1 - X^e \bmod N$ and $p_2(X) = c_2 - (X + \delta)^e \bmod N$, for which x is a common (and unique) root. These two polynomials are distinct, but are both divisible by $X - x$, which can be found by computing their GCD. We thus have an efficient cie algorithm that executes within time t_{cie} bounded by the running time of Coppersmith method for finding δ, $t_{\mathcal{C}(N,e^2)}$, plus the time needed to compute the GCD of two polynomials of degree e, which we denote $t_{\mathsf{GCD}(e)}$.

5.2 Rabin Function

The Rabin function is unfortunately not a permutation. However, for particular moduli we can limit its domain and co-domain to convert it into a bijection. More precisely, if p and q are Blum integers, then -1 a non-quadratic residue modulo p and q, and hence is a false square modulo $N = pq$. Put otherwise, $J_N(-1) = +1$ where $J_N(\cdot)$ denotes the Jacobi symbol modulo N. In addition, any square x in $\mathbb{Z}_N^*$ admits four square roots in $\mathbb{Z}_N^*$, derived from the two pairs of square roots of x in $\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$ using the Chinese Remainder Theorem. As a consequence, one and only one is also a quadratic residue modulo N, which we denote α. Then, α and $-\alpha$ are the two square roots of x with Jacobi symbol $+1$. We will ignore the other two square roots of x that have Jacobi symbol -1. Let $\mathcal{J}_N$ denote the subgroup of the multiplicative subgroup of $\mathbb{Z}_N$ whose elements have Jacobi symbol $+1$ (membership can be efficiently decided). We additionally restrict $\mathcal{J}_N$ to the elements smaller than $N/2$, and we denote this subset $\mathcal{J}_N^<$. We now consider the function

$$\mathsf{SQ}[N] : \mathcal{J}_N^< \times \{0, 1\} \to \mathcal{J}_N$$
$$\mathsf{SQ}[N] : (x, b) \mapsto (-1)^b x^2 \bmod N$$

The inverse function takes an element $y \in \mathcal{J}_N$, which may be a true quadratic residue or a false one. In the former case, one extracts the unique square root α that is also a quadratic residue and sets x to be the smallest value in $\{\alpha, N - \alpha\}$ that is less than $N/2$; the inverse of y is $(x, 0)$. In the latter case, one does as before to compute x, but from $-y$, which is a true quadratic residue; the inverse of y is $(x, 1)$. The function $\mathsf{SQ}[N]$ thus defined is a bijection from $\mathcal{J}_N^< \times \{0,1\}$ onto $\mathcal{J}_N$.

One-wayness Let us assume that an algorithm $\mathcal{A}$ can invert $\mathsf{SQ}[N]$ with non-negligible probability. Then one can first choose a random $z \in \mathbb{Z}_N^* \backslash \mathcal{J}_N$ (instead of $\mathcal{J}_N^<$) and a random bit b, and submit $y = (-1)^b \times z^2 \bmod N$ to $\mathcal{A}$. This element y is uniformly distributed in $\mathcal{J}_N$, and thus with non-negligible probability $\mathcal{A}$ outputs $(x, b') \in \mathcal{J}_N^< \times \{0,1\}$ such that $y = (-1)^{b'} \times x^2 = (-1)^b \times z^2 \bmod N$. Since -1 is a false quadratic residue, necessarily $b' = b$ and $x^2 = z^2 \bmod N$, with $x \in \mathcal{J}_N$ and $z \notin \mathcal{J}_N$. The GCD of $x - z$ and N is either p or q, from which N can be factored. This function is thus one-way under the integer factoring problem.

As above, in order to be used with ZAEP, we have to consider the function $\mathsf{SQ}[N]$ as a function of two bitstrings. Given an input $(x, b) \in \mathcal{J}_N^< \times \{0,1\}$, for any $0 \leq \ell \leq n-1$ we can uniquely write $x \in \mathbb{Z}_N^*$ as $x = r \times 2^\ell + s$, with $s \in \{0,1\}^\ell$ and $r \in \{0,1\}^{n-1-\ell}$. We consider thus the function:

$$\begin{aligned} \mathsf{SQ}[N] &: \{0,1\}^{n-\ell} \times \{0,1\}^\ell \to \{0,1\}^n \\ \mathsf{SQ}[N] &: (b\|r, s) \mapsto (-1)^b \times (r \times 2^\ell + s)^2 \bmod N \end{aligned}$$

Second-Input Extractability Given an output c of $\mathsf{SQ}[N]$ and a pair of values b, r, the Second-Input Extraction problem consists in solving the equation $p(X) = 0 \bmod N$ for $p(X) = c - (-1)^b \times (r \times 2^\ell + X)^2 \bmod N$ with the additional constraint $|X| < 2^\ell$. The above Coppersmith method finds the root s (the second input to $\mathsf{SQ}[N]$ used to compute c if $b\|r$ is the correct first input) provided $2^\ell < N^{1/2}$, or equivalently when $\ell < n/2$. We thus have an efficient sie algorithm that runs within time $t_{\mathsf{sie}} \leq t_{C(N,2)}$.

Common-Input Extractability The Common-Input Extraction problem can be solved as in the case of RSA, provided $\ell < n/4$. We thus have an efficient cie algorithm whose running time t_{cie} is bounded by $t_{C(N,4)} + t_{\mathsf{GCD}(2)}$.

We denote by $\mathsf{Rabin\text{-}ZAEP}$ the encryption scheme resulting from instantiating ZAEP with the function $\mathsf{SQ}[N]$. Since this function operates only on elements in $\mathcal{J}_N^<$, the encryption algorithm may have to iterate:

Key Generation The algorithm $\mathcal{KG}$ generates two Blum integers p and q of length $n/2$, and outputs (pk, sk), where $pk = N = pq$ and $sk = (p, q)$;

Encryption Given a public key N and a message $m \in \{0,1\}^\ell$, the encryption algorithm iteratively samples a random value $r \in \{0,1\}^{k-1}$ and a bit b and sets $s = m \oplus G(b\|r)$, stopping when $x = r \times 2^\ell + s \in \mathcal{J}_N^<$. This requires on average one iteration only. The ciphertext c is computed as

$$\mathsf{SQ}[N](b\|r, s) = (-1)^b \times (r \times 2^\ell + s)^2 \bmod N;$$

Decryption Given a secret key (p, q) and a ciphertext c, $\mathcal{D}$ first inverts $\mathsf{SQ}[N]$ using the prime factors (p, q) of N and gets (x, b). It then parses x as $r \times 2^\ell + s \bmod N$ and outputs $m = s \oplus G(b\|r)$.

5.3 Practical Considerations

For $\mathsf{RSA\text{-}ZAEP}$, all the required properties to achieve $\mathsf{IND\text{-}CCA}$-security hold as long as $e < \sqrt{n/\ell}$. For a practical message size ℓ, e has to be small (e.g. $e = 3$). But for a small exponent e, both sie and cie algorithms are efficient operations on small polynomials, and thus the reduction is efficient: from an adversary that achieves an IND-CCA advantage ε within time t, one can invert RSA with small exponent with success probability essentially ε, within time close to t. As a consequence, one can use classical RSA moduli: for $e = 3$, a 1024-bit modulus allows to encrypt 112-bit messages, whereas a 1536-bit modulus allows to securely encrypt messages of up to 170-bits.

For Rabin-ZAEP, encryption is reasonably efficient (an evaluation of $\mathcal{J}(\cdot)$ on average plus one modular square). The IND-CCA-security of the scheme can be reduced to the integer factoring problem in the random oracle model, with an efficient reduction (even better than for RSA exponent 3). As a consequence, for $n = 1024$, one can securely encrypt messages of up to 256-bits. This suffices, for instance, to encrypt AES keys of all standard sizes.

5.4 Other Redundancy-Free Schemes

We compare our security result of Theorem 1 to the security results for 3-round OAEP (see Fig. 1(b)) and the 4-round scheme of Abe et al. [2].

The original result about the IND-CCA security of 3-round RSA-OAEP [36] relies on an intermediate reduction to the partial-domain one-wayness of RSA. Phan and Pointcheval [37] improved on this result by showing a direct reduction to the (full-domain) one-wayness of RSA, which avoids the additional cost of reducing partial-domain one-wayness to one-wayness. They show that given an adversary $\mathcal{A}$ against the IND-CCA-security of 3-round OAEP that executes within time $t_{\mathcal{A}}$ and makes at most q_{G} queries to its 3 hash oracles and $q_{\mathcal{D}}$ queries to its decryption oracle, it is possible to construct an inverter $\mathcal{I}$ for RSA that executes within time $t_{\mathcal{I}}$, such that

$$t_{\mathcal{I}} \leq t_{\mathcal{A}} + t_{\mathsf{RSA}} \times ((q_{\mathcal{D}} + 1)q_{\mathsf{G}}^2 + q_{\mathcal{D}}^2)$$
$$\mathbf{Succ}_f^{\mathsf{OW}}(\mathcal{I}) \geq \mathbf{Adv}_{\mathsf{OAEP3R}}^{\mathsf{CCA}}(\mathcal{A}) - \frac{5q_{\mathcal{D}}q_{\mathsf{G}} + q_{\mathcal{D}}^2 + q_{\mathcal{D}} + q_{\mathsf{G}}}{2^k}$$

The probability loss in the above reduction can be made negligibly small with an appropriate choice of k, the length of the random value used during encryption. However, even while t_{RSA} is small, the $q_{\mathcal{D}}q_{\mathsf{G}}^2$ factor in the time bound makes the reduction for 3-round OAEP inefficient, because $q_{\mathsf{G}} \gg q_{\mathcal{D}}$ can be large. This quadratic contribution in the number of hash queries also appears in the OAEP security bound and is the major reason for requiring larger moduli.

The 4-round scheme of Abe et al. [2] improves on the efficiency of 3-round OAEP at the cost of one extra Feistel round. Given an adversary $\mathcal{A}$ against the IND-CCA-security of the scheme that executes within time $t_{\mathcal{A}}$ and makes at most q_{G} hash oracle queries and $q_{\mathcal{D}}$ decryption queries, it is possible to construct an inverter $\mathcal{I}$ for the underlying permutation, say RSA, that executes within time $t_{\mathcal{I}}$, such that

$$t_{\mathcal{I}} \leq t_{\mathcal{A}} + t_{\mathsf{RSA}} \times q_{\mathsf{G}}^2$$
$$\mathbf{Succ}_f^{\mathsf{OW}}(\mathcal{I}) \geq \mathbf{Adv}_{\mathsf{OAEP4R}}^{\mathsf{CCA}}(\mathcal{A}) - \frac{4q_{\mathsf{G}}}{2^k} - \frac{2q_{\mathcal{D}}^2}{2^{2k}} - \frac{2q_{\mathsf{G}}(q_{\mathcal{D}} + 1)}{2^{3k}}$$

In contrast to 3-round OAEP, the leading term in the probability loss is $O((q_{\mathsf{G}} + q_{\mathcal{D}})/2^k)$ because $q_{\mathsf{G}}, q_{\mathcal{D}}$ must be bounded by 2^k to achieve semantic security. This allows to use smaller moduli and to get an optimal ciphertext overhead for sufficiently large messages.

In comparison to the above schemes, we show the following bounds for ZAEP in Theorem 1:

$$t_{\mathcal{I}} \leq t_{\mathcal{A}} + 2q_{\mathsf{G}}q_{\mathcal{D}}\ t_{\mathsf{sie}} + q_{\mathcal{D}}^2\ t_{\mathsf{cie}}$$
$$\mathbf{Succ}_f^{\mathsf{OW}}(\mathcal{I}) \geq \mathbf{Adv}_{\mathsf{ZAEP}}^{\mathsf{CCA}}(\mathcal{A}) - \frac{q_{\mathcal{D}}}{2^n}$$

The probability loss in our reduction is negligible and the leading term in the time bound is linear in q_{G}, allowing the use of standard RSA moduli.

6 Related Work

Plaintext-awareness and Non-Redundancy Plaintext awareness is an intuitive concept, that has proved difficult to formalize. The concept was introduced by Bellare and Rogaway for proving security of OAEP [10]. However, their work only dealt with a weak notion of plaintext-awareness that provides a weaker, non-adaptive, notion of chosen-ciphertext security [33] rather than the adaptive notion of IND-CCA security considered in this paper. Subsequently, Bellare et al. [7] enhanced the plaintext-awareness notion to guarantee IND-CCA security. In an effort to accommodate it to the standard model, the definition was further refined by Herzog, Liskov and Micali [30], Bellare and Palacio [8], Dent [23], and Birket and Dent [13]. As noted in the introduction, plaintext-awareness is an appealing concept: it is satisfied by most IND-CCA encryption schemes, and the common way to transform an IND-CPA scheme into an IND-CCA scheme is to introduce redundancy that ensures plaintext-awareness. In fact, it has been observed that existing schemes, such as OAEP, cease to guarantee IND-CCA security—but still retain IND-CPA security—whenever the redundancy is omitted. Nevertheless, several works have shown that redundancy and plaintext-awareness are not required to achieve chosen-ciphertext security. The initial results in this direction are due to Phan and Pointcheval [36, 37]; earlier work by Desai [24] achieves a similar goal, but in the setting of symmetric encryption. Libert and Quisquater [32] build a redundancy-free identity-based encryption scheme that achieves adaptive IND-CCA security. More recently, Boyen [19] proposes a compact redundancy-free encryption scheme based on the Gap-Diffie-Hellman problem [34]. Whereas Boyen's scheme is definitely optimal from the point of view of bandwidth, with a 160-bit overhead only, it is not really efficient because many costly full exponentiations must be computed for encryption and decryption.

Formal proofs of cryptographic schemes The application of formal methods to cryptography has a long and rich history. However, much of the the work in this area has focused on the formal verification of cryptographic protocols in the symbolic model, which assumes that the underlying primitives are perfectly secure. A seminal article by Abadi and Rogaway [1] shows, for the case of encryption, that symbolic methods are indeed sound for the computational model (under strong but achievable assumptions on primitives), and can thus be used to prove cryptographically meaningful guarantees. The computational soundness result of Abadi and Rogaway has been extended in many directions; we refer the reader to [21] for a survey on computational soundness.

In contrast, the application of formal proofs to cryptographic schemes is more recent, and less developed. To our best knowledge, Impagliazzo and Kapron [31] were the first to propose a formal logic to reason about indistinguishability. Using this logic, they prove that next-bit unpredictability implies pseudo-randomness. However, the logic cannot handle adaptive adversaries with oracle access. Computational Indistinguishability Logic [3] is a more recent logic that overcomes these limitations. Both of these works provide logical foundations for reasoning about cryptographic systems, but lack tool support.

In an inspiring article, Halevi [29] advocates that cryptographic proofs should be computer-assisted, and outlines the design of an automated tool to support cryptographic proofs that follow the code-based game-playing approach. CryptoVerif [14] is among the first tools to have provided support for computer-aided cryptographic proofs. It allows users to conduct, automatically or interactively, game-based concrete security proofs of primitives or protocols. Games in CryptoVerif are modeled as processes in the applied

π-calculus, and transitions are proved using a variety of methods, including process-algebraic (for instance bisimulations) or purpose-built (for instance failure events) tools. To date, CryptoVerif has been applied to prove the security of the Full-Domain Hash signature scheme [16] and several protocols; we refer to [15] for a more detailed account of the examples proved with CryptoVerif. The work we report in this paper uses EasyCrypt [4], a more recent tool that takes a programming language approach to cryptographic proofs. EasyCrypt and its predecessor CertiCrypt have been used to verify a number of emblematic cryptographic schemes, including OAEP [5]. As CryptoVerif, EasyCrypt and CertiCrypt aim to provide general frameworks that capture common reasoning patterns in cryptography. An alternative is to develop specialized logics, that are able to prove a particular property for a given class of schemes. A relevant example is the Hoare logic of Courant et al. [22], which allows to prove automatically that an encryption scheme based on trapdoor one-way functions, random oracles, concatenation and exclusive-or is IND-CPA or IND-CCA secure. Their logic (or a suitable extension) uses a syntactic form of plaintext-awareness to conclude that an encryption scheme is IND-CCA secure; hence it cannot be applied to conclude IND-CCA security of ZAEP.

7 Conclusion

ZAEP is a surprisingly simple and efficient padding scheme that achieves adaptive chosen-ciphertext security without introducing any redundancy. Using the EasyCrypt tool, we have built a machine-checked proof that ZAEP yields IND-CCA security with a rather efficient reduction, whenever it is instantiated with trapdoor permutations satisfying two intuitive algebraic properties that hold for the Rabin function and small exponent RSA. The proof is significant beyond its intrinsic interest, as the first application of verified security to a novel construction. Pleasingly, starting from a high-level intuition, we were able to build with reasonable effort in less than a week and directly in EasyCrypt, the sequence of games for proving IND-CCA security. The time needed to complete the proof stands in sharp contrast with the six man-monthes that were reported needed to reproduce the proof of OAEP in CertiCrypt [5]. Thus, our work provides further evidence that, as stated in [4], "EasyCrypt makes a significant step towards the adoption of computer-aided proofs by working cryptographers".

The ZAEP proof opens exciting perspectives for future work. On the one hand, it suggests that automation can be significantly improved through user-defined and built-in strategies that automatically generate a sequence of games. More speculatively, we are currently investigating whether strategies could provide an effective means to automate IND-CPA and IND-CCA proofs for encryption schemes obtained with methods of program synthesis. In a parallel thread of work,[1] we have implemented a synthesis tool that generates encryption schemes based on trapdoor one-way permutations, random oracles, concatenation and exclusive-or. In order to limit the set of candidate schemes to examine, we have constrained the generation mechanism by Dolev-Yao filters that eliminate obviously insecure schemes. Thus, the synthesis algorithm generates a list of candidates that is exhaustive up to a given number of operations. Noticeably, there are only two candidates with a minimal number of operations (four): the (redundant-free and IND-CPA) Bellare and Rogaway encryption scheme [9], which is known since 1993, and ZAEP, which has not been studied before. The case of ZAEP makes us hopeful that automated synthesis of cryptographic schemes may lead to surprising discoveries.

[1] Joint work with Juan Manuel Crespo, Yassine Lakhnech, and César Kunz.

References

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

[2] M. Abe, E. Kiltz, and T. Okamoto. Chosen ciphertext security with optimal ciphertext overhead. In J. Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 355–371. Springer, Dec. 2008.

[3] G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *17th ACM conference on Computer and Communications Security, CCS 2010*, pages 375–386, New York, 2010. ACM.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Heidelberg, 2011. Springer.

[5] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196, Heidelberg, 2011. Springer.

[6] 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, 2009. ACM.

[7] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, Aug. 1998.

[8] M. Bellare and A. Palacio. Towards plaintext-aware public-key encryption without random oracles. In P. J. Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 48–62. Springer, Dec. 2004.

[9] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73. ACM Press, Nov. 1993.

[10] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In A. D. Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, May 1994.

[11] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In U. M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, May 1996.

[12] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, May / June 2006.

[13] J. Birkett and A. W. Dent. Relations among notions of plaintext awareness. In R. Cramer, editor, *PKC 2008: 11th International Conference on Theory and Practice of Public Key Cryptography*, volume 4939 of *Lecture Notes in Computer Science*, pages 47–64. Springer, Mar. 2008.

[14] 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.

[15] B. Blanchet. Security protocol verification: Symbolic and computational models. In P. Degano and J. D. Guttman, editors, *POST*, volume 7215 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2012.

[16] 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, Heidelberg, 2006. Springer.

[17] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. The Why3 platform. Version 0.71. Online – http://why3.lri.fr, 2010.

[18] D. Boneh. Simplified OAEP for the RSA and Rabin functions. In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 275–291. Springer, Aug. 2001.

[19] X. Boyen. Miniature CCA2 PK encryption: Tight security without redundancy. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 485–501. Springer, Dec. 2007.

[20] D. Coppersmith. Finding a small root of a univariate modular equation. In U. M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer, May 1996.

[21] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011.

[22] 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.

[23] A. W. Dent. The Cramer-Shoup encryption scheme is plaintext aware in the standard model. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 289–307. Springer, May / June 2006.

[24] A. Desai. New paradigms for constructing symmetric encryption schemes secure against chosen-ciphertext attack. In M. Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 394–412. Springer, Aug. 2000.

[25] E. Fujisaki and T. Okamoto. How to enhance the security of public-key encryption at minimum cost. In H. Imai and Y. Zheng, editors, *PKC'99: 2nd International Workshop on Theory and Practice in Public Key Cryptography*, volume 1560 of *Lecture Notes in Computer Science*, pages 53–68. Springer, Mar. 1999.

[26] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, Aug. 1999.

[27] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer, Aug. 2001.

[28] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

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

[30] J. Herzog, M. Liskov, and S. Micali. Plaintext awareness via key registration. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 548–564. Springer, Aug. 2003.

[31] R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. In *44th Annual IEEE symposium on Foundations of Computer Science, FOCS 2003*, pages 372–383. IEEE Computer Society, 2003.

[32] B. Libert and J.-J. Quisquater. Identity based encryption without redundancy. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *ACNS 05: 3rd International Conference on Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 285–300. Springer, June 2005.

[33] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd Annual ACM Symposium on Theory of Computing*. ACM Press, May 1990.

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

[35] T. Okamoto and D. Pointcheval. REACT: Rapid Enhanced-security Asymmetric Cryptosystem Transform. In D. Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 159–175. Springer, Apr. 2001.

[36] D. H. Phan and D. Pointcheval. Chosen-ciphertext security without redundancy. In C.-S. Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Nov. / Dec. 2003.

[37] D. H. Phan and D. Pointcheval. OAEP 3-round: A generic and secure asymmetric encryption padding. In P. J. Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 63–77. Springer, Dec. 2004.

[38] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer, Aug. 1992.

[39] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.

A EasyCrypt Input File

```
cnst k : int.
cnst l : int.
cnst qD : int.

cnst zero_k : bitstring{k}.
cnst zero_l : bitstring{l}.

type pkey.
type skey.
type plaintext  = bitstring{l}.
type ciphertext = bitstring{k} * bitstring{l}.

axiom k_pos : 0 ≤ k.

axiom l_pos : 0 ≤ l.

axiom qD_pos : 0 ≤ qD.

pop KG : () → pkey * skey.

op key_pair : (pkey, skey) → bool.

spec KG() : k1 = KG() ~ k2 = KG() : true ⟹ k1 = k2 ∧ key_pair(fst(k1), snd(k1)).

op f    : (pkey, bitstring{k} * bitstring{l}) → bitstring{k} * bitstring{l}.
op finv : (skey, bitstring{k} * bitstring{l}) → bitstring{k} * bitstring{l}.

axiom finv_l :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (xy:bitstring{k} * bitstring{l}), finv(sk, f(pk, xy)) = xy.

axiom finv_r :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (xy:bitstring{k} * bitstring{l}), f(pk, finv(sk, xy)) = xy.

(* Second-Input Extractor *)
op sie : (pkey, bitstring{k} * bitstring{l}, bitstring{k}) → bitstring{l} option.

axiom sie_spec :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, s:bitstring{l}),
    sie(pk, y, r) = Some(s) ⟺ y = f(pk, (r, s)).

op find_sie_fst :
  (pkey, bitstring{k} * bitstring{l}, (bitstring{k}, bitstring{l}) map) →
  bitstring{k} option.

axiom find_sie_fst_correct :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
```

```
  ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
    in_dom(fst(finv(sk, y)), L) ⇒
    find_sie_fst(pk, y, L) = Some(fst(finv(sk, y))).

axiom find_sie_fst_complete :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
    ¬in_dom(fst(finv(sk, y)), L) ⇒
    find_sie_fst(pk, y, L) = None.

op find_sie_snd :
  (pkey, bitstring{k}, (bitstring{k} * bitstring{l}, bitstring{l}) map) →
  (bitstring{k} * bitstring{l}) option.

axiom find_sie_snd_correct :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l},
        L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
    find_sie_snd(pk, fst(finv(sk, y)), L) = None ⇒
    ¬in_dom(y, L).

axiom find_sie_snd_complete :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (r:bitstring{k}, L:(bitstring{k} * bitstring{l}, bitstring{l}) map,
        y:bitstring{k} * bitstring{l}),
    find_sie_snd(pk, r, L) = Some(y) ⇒
    in_dom(y, L) ∧ r = fst(finv(sk, y)).

(* Common-Input Extractor *)
op cie : (pkey, bitstring{k} * bitstring{l}, bitstring{k} * bitstring{l}) →
        (bitstring{k} * bitstring{l} * bitstring{l}) option.

axiom cie_spec :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y,z:bitstring{k} * bitstring{l}, r:bitstring{k}, s,t:bitstring{l}),
    cie(pk, y, z) = Some((r, s, t)) ⟺
    y = f(pk, (r, s)) ∧ z = f(pk, (r, t)) ∧ y <> z.

op find_cie :
  (pkey, bitstring{k} * bitstring{l},
   (bitstring{k} * bitstring{l},bitstring{l}) map) →
  (bitstring{k} * bitstring{l}) option.

axiom find_cie_correct :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y,z:bitstring{k} * bitstring{l},
        L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
    find_cie(pk, y, L) = Some(z) ⇒
    in_dom(z, L) ∧ fst(finv(sk, z)) = fst(finv(sk, y)) ∧ y <> z.

axiom find_cie_complete :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:ciphertext, L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
    find_cie(pk, y, L) = None ⇒
    ∀ (y':ciphertext),
```

```
      in_dom(y', L) ⇒ fst(finv(sk, y')) <> fst(finv(sk, y)) ∨ y = y'.

(** Derived lemmas, proved either here or in Coq (lemmas.v) *)

prover alt-ergo, cvc3.

lemma find_cie_correct' :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y,z:bitstring{k} * bitstring{l},
         L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
    find_cie(pk, y, L) = Some(z) ⇒
    cie(pk, y, z) = Some((fst(finv(sk, y)), snd(finv(sk, y)), snd(finv(sk, z)))).

lemma sie_find_sie_fst :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
    find_sie_fst(pk, y, L) <> None ⇒
    sie(pk, y, proj(find_sie_fst(pk, y, L))) = Some(snd(finv(sk, y))).

axiom cie_find_cie :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext,bitstring{l}) map),
    find_cie(pk, y, L) <> None ⇒
    cie(pk, y, proj(find_cie(pk, y, L))) =
    Some((fst(finv(sk, y)), snd(finv(sk, y)),
         snd(finv(sk, proj(find_cie(pk, y,L)))))).

lemma find_sie_fst_upd :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, g:bitstring{l},
         L:(bitstring{k}, bitstring{l}) map),
    find_sie_fst(pk, y, L[r <- g]) = None ⟺
    find_sie_fst(pk, y, L) = None ∧ fst(finv(sk, y)) <> r.

lemma find_sie_snd_cie :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext, bitstring{l}) map),
    find_sie_snd(pk, fst(finv(sk, y)), L) <> None ⇒
    find_cie(pk, y, L) <> None ⇒
    let r,s,t = proj(cie(pk, y, proj(find_cie(pk, y, L)))) in y = f(pk, (r, s)).

axiom find_cie_find_sie_snd :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext, bitstring{l}) map),
    find_cie(pk, y, L) = None ⇒
    ¬in_dom(y, L) ⇒
    find_sie_snd(pk, fst(finv(sk, y)), L) = None.

axiom find_sie_snd_upd :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, m:bitstring{l},
         L:(ciphertext, bitstring{l}) map),
    find_sie_snd(pk, r, L[y <- m]) = None ⟺
    find_sie_snd(pk, r, L) = None ∧ fst(finv(sk, y)) <> r.
```

```
axiom find_cie_upd :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y,y':ciphertext, m:bitstring{l}, L:(ciphertext, bitstring{l}) map),
    find_cie(pk, y, L[y' <- m]) = None ⟺
    find_cie(pk, y, L) = None ∧
    (fst(finv(sk, y)) <> fst(finv(sk, y')) ∨ y = y').

axiom cie_spec' :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y,z:bitstring{k} * bitstring{l}),
    cie(pk, y, z) <> None ⇒ fst(finv(sk, y)) = fst(finv(sk, z)).

axiom find_cie_empty :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀ (y:bitstring{k} * bitstring{l}), find_cie(pk, y, empty_map) = None.

axiom find_sie_snd_empty :
  ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
  ∀(r:bitstring{k}), find_sie_snd(pk, r, empty_map) = None.

lemma xor_2 : ∀ (x,y:bitstring{l}), x ⊕ (y ⊕ x) = y.

pred eq_except(M1, M2 : ('a, 'b) map, a : 'a) =
  ∀ (w: 'a), w <> a ⇒ M1[w] = M2[w] ∧ (in_dom(w,M1) ⟺ in_dom(w,M2)).

lemma eqe_update_diff :
  ∀(M1, M2 : ('a, 'b) map, a, a' : 'a, b : 'b),
    eq_except(M1, M2, a) ⇒
    eq_except(M1[a' <- b], M2[a' <- b], a).

lemma eqe_update_same_L :
  ∀(M1, M2 : ('a, 'b) map, a : 'a, b : 'b),
    eq_except(M1, M2, a) ⇒ eq_except(M1[a <- b], M2, a).

lemma eqe_update_same_R :
  ∀(M1, M2 : ('a, 'b) map, a : 'a, b : 'b),
    eq_except(M1, M2, a) ⇒ eq_except(M1, M2[a <- b], a).

type state.

adversary A1() : plaintext * plaintext * state
  { bitstring{k} → bitstring{l}; ciphertext → plaintext }.

adversary A2(st:state, c:ciphertext) : bool
  { bitstring{k} → bitstring{l}; ciphertext → plaintext }.

(*
** Game CCA:
** This is the standard CCA experiment
*)
game CCA = {
 var pk    : pkey
 var sk    : skey
 var LG    : (bitstring{k}, bitstring{l}) map
 var cstar : ciphertext
```

```
  var cdef   : bool
  var q      : int

  fun G(x:bitstring{k}) : bitstring{l} = {
    var g : bitstring{l} = {0,1}^l;
    if (¬in_dom(x, LG)) {
      LG[x] = g;
    }
    return LG[x];
  }

  fun Enc(m:plaintext) : ciphertext = {
    var g : bitstring{l};
    var r : bitstring{k} = {0,1}^k;
    g = G(r);
    return f(pk, (r, g ⊕ m));
  }

  fun Dec(c:ciphertext) : plaintext = {
    var r : bitstring{k};
    var g, s, m : bitstring{l};
    if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
      q = q + 1;
      (r, s) = finv(sk, c);
      g = G(r);
      m = g ⊕ s;
    }
    else {
      m = zero_l;
    }
    return m;
  }

  abs A1 = A1 {G, Dec}
  abs A2 = A2 {G, Dec}

  fun Main() : bool = {
    var m0, m1 : plaintext;
    var b, b' : bool;
    var st : state;
    (pk, sk) = KG();
    LG = empty_map;
    cdef = false;
    q = 0;
    (m0, m1, st) = A1();
    b = {0,1};
    cstar = Enc(b ? m0 : m1);
    cdef = true;
    b' = A2(st, cstar);
    return (b = b');
  }
}.


(*
```

```
** Game G1:
** - Introduce bad
** - Hoist sampling of rstar
** - Inline Enc(mb), G(rstar) in Main and remove Enc procedure
*)
game G1 = CCA
 var rstar : bitstring{k}
 var gstar : bitstring{l}
 var bad   : bool

 where G = {
   var g : bitstring{l} = {0,1}^l;
   if(x = rstar) { bad = true; }
   if (¬in_dom(x, LG)) {
     LG[x] = g;
   }
   return LG[x];
 }

 and Main = {
   var m0, m1 : plaintext;
   var b, b' : bool;
   var st : state;
   (pk, sk) = KG();
   rstar = {0,1}^k;
   bad = false;
   LG = empty_map;
   cdef = false;
   q = 0;
   (m0, m1, st) = A1();
   b = {0,1};
   if (¬in_dom(rstar, LG)) {
     gstar = {0,1}^l;
     LG[rstar] = gstar;
   }
   else {
     bad = true;
     gstar = LG[rstar];
   }
   cstar = f(pk, (rstar, gstar ⊕ (b ? m0 : m1)));
   cdef = true;
   b' = A2(st, cstar);
   return (b = b');
 }.

prover alt-ergo.
unset all.

equiv CCA_G1 : CCA.Main ~ G1.Main : true ⟹ ={res}.
proof.
 inline⟨1⟩ Enc, G; derandomize.
 call (={pk,sk,LG,cstar,cdef,q}); wp.
 auto (={pk,sk,LG,cdef,q} ∧ ¬cdef⟨2⟩).
 swap⟨1⟩ 2 1; trivial.
save.
```

```
claim Pr_CCA_G1 : CCA.Main[res] = G1.Main[res] using CCA_G1.


(*
** Game G2:
** Replace inlined G(rstar) by a random sampling in Main
*)
game G2 = G1
 where Main = {
   var m0, m1 : plaintext;
   var b, b' : bool;
   var st : state;
   (pk, sk) = KG();
   rstar = {0,1}^k;
   bad = false;
   LG = empty_map;
   cdef = false;
   q = 0;
   (m0, m1, st) = A1();
   b = {0,1};
   gstar = {0,1}^l;
   if (in_dom(rstar, LG)) { bad = true; }
   cstar = f(pk, (rstar, gstar ⊕ (b ? m0 : m1)));
   cdef = true;
   b' = A2(st, cstar);
   return (b = b');
 }.

set eqe_update_diff, eqe_update_same_L, eqe_update_same_R.

equiv G1_G2 : G1.Main ∼ G2.Main : true ⟹ ={bad} ∧ (¬bad⟨1⟩ ⇒ ={res}).
proof.
 call upto (bad) with
  (={pk,sk,cstar,rstar,gstar,cdef,q} ∧
   (bad⟨1⟩ ⟺ in_dom(rstar⟨2⟩, LG⟨2⟩)) ∧ eq_except(LG⟨1⟩, LG⟨2⟩, rstar⟨1⟩)).
 derandomize; wp.
 call upto (bad) with
  (={pk,sk,LG,rstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧ (bad⟨1⟩ ⟺ in_dom(rstar⟨2⟩, LG⟨2⟩))).
 trivial.
save.

unset eqe_update_diff, eqe_update_same_L, eqe_update_same_R.

claim Pr_G1_G2 : | G1.Main[res] - G2.Main[res] | ≤ G2.Main[bad]
using G1_G2.


(*
** Game G3:
** Use optimistic sampling to sample sstar instead of gstar, where
**
** G2: gstar = {0,1}^l; sstar = gstar ⊕ mb; cstar = f(rstar, sstar)
** G3: sstar = {0,1}^k; gstar = sstar ⊕ mb; cstar = f(rstar, sstar)
**
```

```
** Remove dependency of b' from b by eliminating gstar as dead-code
** and postponing sampling b
*)
game G3 = G2
 var sstar : bitstring{l}

 where Main = {
   var m0, m1 : plaintext;
   var b, b' : bool;
   var st : state;
   (pk, sk) = KG();
   rstar = {0,1}^k;
   bad = false;
   LG = empty_map;
   cdef = false;
   q = 0;
   (m0, m1, st) = A1();
   if (in_dom(rstar, LG)) { bad = true; }
   sstar = {0,1}^l;
   cstar = f(pk, (rstar, sstar));
   cdef = true;
   b' = A2(st, cstar);
   b = {0,1};
   return (b = b');
 }.

set xor_l_cancel, xor_l_zero_r, xor_l_assoc.

equiv G2_G3 : G2.Main ∼ G3.Main : true ⟹ ={bad,res}.
proof.
 swap⟨2⟩ 13 -5.
 call (={pk,sk,LG,rstar,cstar,cdef,q,bad}); wp.
 rnd (sstar ⊕ (b ? m0 : m1)⟨2⟩); wp; rnd.
 call (={pk,sk,LG,rstar,cdef,q,bad} ∧ ¬cdef⟨1⟩).
 derandomize; trivial.
save.

unset xor_l_cancel, xor_l_zero_r, xor_l_assoc.

claim Pr_G2_G3  : G2.Main[res] = G3.Main[res] using G2_G3.

claim Pr_G2_G3' : G2.Main[bad] = G3.Main[bad] using G2_G3.

claim Pr_G3     : G3.Main[res] = 1 / 2 compute.


(*
** Game G4:
** Introduce LG' to store implicitly-defined values of G(r)
** Inline calls to G in Dec
** Apply optimistic-sampling to sample m rather than LG'[r] in Dec
*)
game G4 = G3
 var LG' : (bitstring{k}, bitstring{l}) map

```

```
where G = {
  var g : bitstring{l} = {0,1}^l;
  if (¬in_dom(x, LG)) {
    if (¬in_dom(x, LG')) {
      LG[x] = g;
    }
    else {
      LG[x] = LG'[x];
    }
  }
  return LG[x];
}

and Dec = {
  var r' : bitstring{k} option;
  var c' : (bitstring{k} * bitstring{l}) option;
  var r  : bitstring{k};
  var g, s, t, m : bitstring{l};
  if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
    q = q + 1;
    r' = find_sie_fst(pk, c, LG);
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, c, r)); (* c = f(r, s) *)
      g = LG[r];
      m = g ⊕ s;
    }
    else {
      r' = find_sie_fst(pk, c, LG');
      if (r' <> None) {
        r = proj(r');
        s = proj(sie(pk, c, r)); (* c = f(r, s) *)
        g = LG'[r];
        m = g ⊕ s;
      }
      else {
        if (cdef ∧ cie(pk, c, cstar) <> None) {
          (r, s, t) = proj(cie(pk, c, cstar));
          (* c = f(r, s) ∧ cstar = f(r, t) *)
          g = {0,1}^l;
          LG[r] = g;
          m = g ⊕ s;
        }
        else {
          (r, s) = finv(sk, c);
          m = {0,1}^l;
          LG'[r] = m ⊕ s;
        }
      }
    }
  }
  else {
    m = zero_l;
  }
  return m;
```

```
}

and Main = {
  var m0, m1 : plaintext;
  var b' : bool;
  var st : state;
  (pk, sk) = KG();
  rstar = {0,1}^k;
  sstar = {0,1}^l;
  cstar = f(pk, (rstar, sstar));
  LG = empty_map;
  LG' = empty_map;
  cdef = false;
  q = 0;
  (m0, m1, st) = A1();
  cdef = true;
  b' = A2(st, cstar);
  return true;
}.

set find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst,
 xor_l_cancel, xor_l_zero_r, xor_l_assoc,
 cie_spec', cie_spec, finv_l, finv_r.

equiv G3_G4_Dec : G3.Dec ~ G4.Dec :
 (={pk,sk,rstar,sstar,cstar,cdef,q} ∧
  (key_pair(pk, sk) ∧ cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
 (bad⟨1⟩ ⟺
  (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
 (∀ (x:bitstring{k}),
   in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
 (∀ (x:bitstring{k}),
   ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
   in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
 (∀ (x:bitstring{k}),
   ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
proof.
 if; [ | trivial].
 inline G.
 case⟨2⟩: find_sie_fst(pk,c,LG) <> None.
 condf⟨1⟩ at 6; [ | trivial].
 condt⟨2⟩ last; [ | trivial ].
 derandomize; wp; trivial.
 condf⟨2⟩ last; [ | trivial].
 case⟨2⟩: find_sie_fst(pk,c,LG') <> None.
 condt⟨2⟩ last; [ | trivial].
 condf⟨1⟩ at 6; [ | trivial].
 derandomize; wp; trivial.
 condf⟨2⟩ last; [ | trivial].
 condt⟨1⟩ at 6; [ | trivial].
 case⟨1⟩: cdef ∧ cie(pk,c, cstar) <> None.
 condt⟨1⟩ at 5; [ | trivial].
 condt⟨2⟩ last; [ | trivial].
 derandomize; trivial.
 derandomize; wp. rnd (m_0 ⊕ snd(finv(sk,c)⟨2⟩)); trivial.
```

```
 unset xor_l_assoc.
 trivial.
save.

equiv G3_G4_Dec : G3.Dec ~ G4.Dec :
 (={pk,sk,rstar,sstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
  (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
 (bad⟨1⟩ ⟺
  (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
 (∀ (x:bitstring{k}),
   in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
 (∀ (x:bitstring{k}),
   ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
   in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
 (∀ (x:bitstring{k}),
   ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
proof.
 if; [ | trivial].
 inline G.
 case⟨2⟩: find_sie_fst(pk,c,LG) <> None.
 condf⟨1⟩ at 6; [ | trivial].
 condt⟨2⟩ last; [ | trivial].
 derandomize; wp; trivial.
 condf⟨2⟩ last; [ | trivial].
 case⟨2⟩: find_sie_fst(pk,c,LG') <> None.
 condt⟨2⟩ last; [ | trivial].
 condf⟨1⟩ at 6; [ | trivial].
 derandomize; wp; trivial.
 condf⟨2⟩ last; [ | trivial].
 condt⟨1⟩ at 6; [ | trivial].
 case⟨1⟩: cdef ∧ cie(pk,c,cstar) <> None.
 condt⟨1⟩ at 5; [ | trivial].
 condt⟨2⟩ last; [ | trivial].
 derandomize; trivial.
 set xor_l_assoc.
 derandomize; wp; rnd (m_0 ⊕ snd(finv(sk,c)⟨2⟩)); trivial.
 unset xor_l_assoc.
 trivial.
save.

unset find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst,
  xor_l_cancel, xor_l_zero_r, xor_l_assoc, cie_spec', cie_spec.

equiv G3_G4 : G3.Main ~ G4.Main : true ⟹
 (bad⟨1⟩ ⟺
  (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩).
 app 1 1 ={pk,sk} ∧ key_pair(pk,sk)⟨1⟩.
 derandomize; wp; apply: KG(); trivial.
 swap⟨1⟩ -11; swap⟨1⟩ 9 -6.
 call
  (={pk,sk,rstar,sstar,cstar,cdef,q} ∧
   (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
  (bad⟨1⟩ ⟺
    (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
  (∀ (x:bitstring{k}),
```

```
    in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
  (∀ (x:bitstring{k}),
    ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
    in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
  (∀ (x:bitstring{k}),
    ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
 wp.
 call
  (={pk,sk,rstar,sstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
   (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
  (bad⟨1⟩ ⟺
    (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
  (∀ (x:bitstring{k}),
    in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
  (∀ (x:bitstring{k}),
    ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
    in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
  (∀ (x:bitstring{k}),
    ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
 trivial.
save.

unset finv_l, finv_r.

claim Pr_G3_G4 :
  G3.Main[bad] =
  G4.Main[in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG')]
using G3_G4.


(*
** Game G5:
** Introduce LD
** Ciphertexts that implicitly-define values of G(r) are stored in LD
** Remove finv from Dec
*)
game G5 = G4
 var LD : (bitstring{k} * bitstring{l}, bitstring{l}) map

 where G = {
   var c : ciphertext option;
   var g : bitstring{l} = {0,1}^l;
   if (¬in_dom(x, LG)) {
     c = find_sie_snd(pk, x, LD);
     if (c = None) {
       LG[x] = g;
     }
     else {
       LG[x] = LD[proj(c)] ⊕ proj(sie(pk, proj(c), x));
     }
   }
   return LG[x];
 }

 and Dec = {
```

```
  var r' : bitstring{k} option;
  var c' : (bitstring{k} * bitstring{l}) option;
  var r  : bitstring{k};
  var g, s, t, m : bitstring{l};
  if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
    q = q + 1;
    r' = find_sie_fst(pk, c, LG);
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, c, r)); (* c = f(r, s) *)
      g = LG[r];
      m = g ⊕ s;
    }
    else {
      if (in_dom(c, LD)) {
        m = LD[c];
      }
      else {
        c' = find_cie(pk, c, LD);
        if (c' <> None) {
          (r, s, t) = proj(cie(pk, c, proj(c')));
          (* c = f(r, s) ∧ c' = f(r, t) *)
          g = LD[proj(c')] ⊕ s;
          m = g ⊕ t;
        }
        else {
          if (cdef ∧ cie(pk, c, cstar) <> None) {
            (r, s, t) = proj(cie(pk, c, cstar));
            (* c = f(r, s) ∧ cstar = f(r, t) *)
            g = {0,1}^l;
            LG[r] = g;
            m = g ⊕ s;
          }
          else {
            m = {0,1}^l;
            LD[c] = m;
          }
        }
      }
    }
  }
  else {
    m = zero_l;
  }
  return m;
}

and Main = {
  var m0, m1 : plaintext;
  var b' : bool;
  var st : state;
  (pk, sk) = KG();
  rstar = {0,1}^k;
  sstar = {0,1}^l;
  cstar = f(pk, (rstar, sstar));
```

```
   bad = false;
   LG = empty_map;
   LD = empty_map;
   cdef = false;
   q = 0;
   (m0, m1, st) = A1();
   cdef = true;
   b' = A2(st, cstar);
   return true;
 }.

set find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst, xor_2.

equiv G4_G5_Dec : G4.Dec ~ G5.Dec :
 (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk, sk)⟨1⟩ ∧
  (∀ (x:ciphertext),
    find_sie_fst(pk⟨1⟩, x, LG'⟨1⟩) = None ⟺
    find_cie(pk⟨2⟩, x, LD⟨2⟩) = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
  (∀ (r:bitstring{k}),
    ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
  (∀ (x:ciphertext),
   let r,s = finv(sk⟨1⟩, x) in
     (in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x]))).
proof.
 if; [ | trivial].
 case⟨1⟩: find_sie_fst(pk, c, LG) <> None.
 condt last; trivial.
 condf last; [ | trivial | trivial].
 case⟨2⟩: in_dom(c, LD).
 condt⟨2⟩ last; [ | trivial].
 condt⟨1⟩ last; [ | trivial].
 trivial.
 condf⟨2⟩ last; [ | trivial].
 case⟨1⟩: find_sie_fst(pk, c, LG') <> None.
 condt⟨1⟩ last; [ | trivial].
 condt⟨2⟩ last; [ | trivial].
 trivial.
 app 0 0
  (={c,pk,sk,LG,cstar,cdef} ∧ key_pair(pk,sk)⟨1⟩ ∧
   (∀ (x:ciphertext),
      find_sie_fst(pk,x,LG')⟨1⟩ = None ⟺
      find_cie(pk,x,LD)⟨2⟩ = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
   (∀ (x: ciphertext),
     let r,s = finv(sk⟨1⟩, x) in
     in_dom(x,LD⟨2⟩) ⇒ in_dom(r,LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x]) ∧
   (cdef⟨2⟩ <> true ∨ c⟨2⟩ <> cstar⟨2⟩) ∧
   ¬in_dom(c,LD)⟨2⟩ ∧
   let c' = find_cie(pk, c, LD)⟨2⟩ in
     c' <> None ∧ in_dom(proj(c'), LD⟨2⟩)).
 set find_cie_correct.
 trivial.
 unset find_cie_correct.
 set sie_find_sie_fst, cie_find_cie, find_cie_correct',
     cie_spec, finv_l, finv_r, xor_l_cancel, xor_l_zero_r, xor_l_assoc.
 app 0 0 (
```

```
   ={c,pk,sk,LG,cstar,cdef} ∧ key_pair (pk⟨1⟩,sk⟨1⟩) ∧
   find_sie_fst(pk,c,LG')⟨1⟩ <> None ∧
   (∀ (x_0:ciphertext),
   let r,s = finv(sk⟨1⟩,x_0) in
     in_dom(x_0,LD⟨2⟩) ⇒ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x_0]) ∧
   let c' = find_cie(pk, c, LD)⟨2⟩ in
   let r', s, t = proj(cie(pk, c, proj(c')))⟨2⟩ in
   let r = proj(find_sie_fst(pk, c, LG'))⟨1⟩ in
     c' <> None ∧
     r = r' ∧
     c⟨1⟩ = f(pk, (r, s))⟨1⟩ ∧
     proj(c') = f(pk, (r, t))⟨2⟩ ∧
     in_dom(proj(c'), LD⟨2⟩) ∧
     r = fst(finv(sk⟨1⟩, proj(c')))).
  trivial.
  app 0 0 (
    let c' = find_cie(pk, c, LD)⟨2⟩ in
    let _, s, t = proj(cie(pk, c, proj(c')))⟨2⟩ in
    let r = proj(find_sie_fst(pk, c, LG'))⟨1⟩ in
      sie(pk,c,r)⟨1⟩ = Some(s) ∧
      LG'⟨1⟩[r] = t ⊕ LD⟨2⟩[proj(c')]).
  trivial.
  trivial.

  condf⟨1⟩ last; [ | trivial].
  condf⟨2⟩ last; [ | trivial].
  case⟨1⟩: cdef ∧ cie(pk, c, cstar) <> None.
  condt⟨1⟩ last; [ | trivial].
  condt⟨2⟩ last; [ | trivial].
  trivial.
  condf⟨1⟩ last; [ | trivial].
  condf⟨2⟩ last; [ | trivial].
  set find_sie_fst_upd, find_sie_snd_upd, find_cie_upd.
  trivial.
save.

timeout 5.

equiv G4_G5_G : G4.G ~ G5.G :
  (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk, sk)⟨1⟩ ∧
  (∀ (x:ciphertext),
    find_sie_fst(pk⟨1⟩, x, LG'⟨1⟩) = None ⇒
    find_cie(pk⟨2⟩, x, LD⟨2⟩) = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
  (∀ (r:bitstring{k}),
    ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
  (∀ (x:ciphertext),
    let r,s = finv(sk⟨1⟩, x) in
    in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x])).
proof.
  case⟨1⟩: ¬in_dom(x, LG).
  condt last; [ | trivial | trivial].
  case⟨1⟩: ¬in_dom(x, LG').
  condt⟨1⟩ last; [ | trivial].
  condt⟨2⟩ last; [ | trivial].
  trivial.
```

```
 condf⟨1⟩ last; [ | trivial].
 condf⟨2⟩ last; [ | trivial].
 unset xor_2.
 set find_sie_snd_complete, xor_l_comm.
 trivial.
 app 0 0 (
   ={x,pk,sk,LG} ∧ key_pair(pk,sk)⟨1⟩ ∧
   let c' = find_sie_snd(pk,x,LD)⟨2⟩ in
     c' <> None ∧ in_dom (x,LG')⟨1⟩ ∧
     LG'⟨1⟩[fst(finv(sk⟨1⟩,proj(c')))] =
     snd(finv(sk⟨1⟩,proj(c'))) ⊕ LD⟨2⟩[proj(c')]); trivial.
 condf last; trivial.
save.

set find_sie_snd_empty, find_cie_empty.

equiv G4_G5 : G4.Main ~ G5.Main : true ⟹
 (in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG'))⟨1⟩ ⇒
 (in_dom(fst(finv(sk,cstar)), LG) ∨
  find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None)⟨2⟩.
proof.
 app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
 derandomize; wp; apply: KG(); trivial.
 auto
  (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧
  (∀ (x:ciphertext),
    find_sie_fst(pk⟨1⟩,x,LG'⟨1⟩) = None ⟺
    find_cie(pk⟨2⟩,x,LD⟨2⟩) = None ∧ ¬in_dom(x,LD⟨2⟩)) ∧
  (∀ (r:bitstring{k}),
    ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
  (∀ (x:ciphertext),
    let r,s = finv(sk⟨1⟩,x) in
    in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x])).
 trivial.
save.

claim Pr_G4_G5 :
  G4.Main[in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG')] ≤
  G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
          find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None]
using G4_G5.


game OW = {
 var pk    : pkey
 var sk    : skey
 var LG    : (bitstring{k}, bitstring{l}) map
 var LD    : (bitstring{k} * bitstring{l}, bitstring{l}) map
 var cstar : ciphertext
 var cdef  : bool
 var q     : int

 fun G(x:bitstring{k}) : bitstring{l} = {
   var c : ciphertext option;
   var g : bitstring{l} = {0,1}^l;
```

```
  if (¬in_dom(x, LG)) {
    c = find_sie_snd(pk, x, LD); (* t_sie * qD *)
    if (c = None) {
      LG[x] = g;
    }
    else {
      LG[x] = LD[proj(c)] ⊕ proj(sie(pk,proj(c), x));
    }
  }
  return LG[x];
}

fun Dec(c:ciphertext) : plaintext = {
  var r' : bitstring{k} option;
  var c' : (bitstring{k} * bitstring{l}) option;
  var r  : bitstring{k};
  var g, s, t, m : bitstring{l};
  if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
    q = q + 1;
    r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, c, r));
      g = LG[r];
      m = g ⊕ s;
    }
    else {
      if (in_dom(c, LD)) {
        m = LD[c];
      }
      else {
        c' = find_cie(pk, c, LD); (* t_cie * qD *)
        if (c' <> None) {
          (r, s, t) = proj(cie(pk, c, proj(c')));
          g = LD[proj(c')] ⊕ s;
          m = g ⊕ t;
        }
        else {
          if (cdef ∧ cie(pk, c, cstar) <> None) { (* t_cie *)
            (r, s, t) = proj(cie(pk, c, cstar));
            g = {0,1}^l;
            LG[r] = g;
            m = g ⊕ s;
          }
          else {
            m = {0,1}^l;
            LD[c] = m;
          }
        }
      }
    }
  }
  else {
    m = zero_l;
  }
```

```
    return m;
  }

  abs A1 = A1 {G, Dec}
  abs A2 = A2 {G, Dec}

  fun B(z:bitstring{k} * bitstring{l}) : bitstring{k} * bitstring{l} = {
    var m0, m1 : plaintext;
    var b' : bool;
    var r' : bitstring{k} option;
    var r : bitstring{k};
    var s, t : bitstring{l};
    var c : ciphertext option;
    var st : state;
    LG = empty_map;
    LD = empty_map;
    cstar = z;
    cdef = false;
    q = 0;
    (m0, m1, st) = A1();
    cdef = true;
    b' = A2(st, cstar);
    r' = find_sie_fst(pk, cstar, LG); (* t_sie * qG *)
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, cstar, r));
    }
    else
    {
      c = find_cie(pk, cstar, LD); (* t_cie * qD *)
      if (c <> None) {
        (r, s, t) = proj(cie(pk, cstar, proj(c)));
      }
      else {
        r = zero_k;
        s = zero_l;
      }
    }
    return (r, s);
  }

  var xstar : bitstring{k}
  var ystar : bitstring{l}

  fun Main() : bool = {
    var x : bitstring{k};
    var y : bitstring{l};
    (pk, sk) = KG();
    xstar = {0,1}^k;
    ystar = {0,1}^l;
    (x,y) = B(f(pk, (xstar,ystar)));
    return (f(pk, (x,y)) = f(pk, (xstar,ystar)));
  }
}.
```

```
set find_cie_find_sie_snd, find_sie_snd_cie, find_cie_correct'.

equiv G5_OW : G5.Main ~ OW.Main : true ⟹
 in_dom(fst(finv(sk⟨1⟩,cstar⟨1⟩)), LG⟨1⟩) ∨
 find_sie_snd(pk⟨1⟩,fst(finv(sk⟨1⟩,cstar⟨1⟩)), LD⟨1⟩) <> None ⇒
 ¬in_dom(cstar⟨2⟩, LD⟨2⟩) ⇒ res⟨2⟩.
proof.
 app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
 derandomize; wp; apply: KG(); trivial.
 inline B; derandomize.
 app 15 13 (={pk,sk,LG,LD,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧
   (cstar = f(pk,(xstar,ystar)))⟨2⟩ ).
 auto (={pk,sk,LG,LD,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩); trivial.
 trivial.
save.

unset find_cie_find_sie_snd, find_sie_snd_cie, find_cie_correct'.

claim Pr_G5_OW' :
  G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
          find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None] ≤
  OW.Main[res ∨ in_dom(cstar, LD)]
using G5_OW.

claim Pr_OW :
  OW.Main[res ∨ in_dom(cstar,LD)] ≤ OW.Main[res] + OW.Main[in_dom(cstar,LD)]
compute.

claim Pr_G5_OW :
  G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
          find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None] ≤
  OW.Main[res] + OW.Main[in_dom(cstar, LD)].

claim CCA_OW :
  | CCA.Main[res] - 1 / 2 | ≤ OW.Main[res] + OW.Main[in_dom(cstar, LD)].

(*
** Follows a rather technical sequence of games to bound
** OW.Main[in_dom(cstar,LD)]
*)

game OW1 = {
 var pk    : pkey
 var sk    : skey
 var LG    : (bitstring{k}, bitstring{l}) map
 var LD    : (bitstring{k} * bitstring{l}, bitstring{l}) map
 var LC    : ciphertext list
 var cstar : ciphertext
 var cdef  : bool
 var q     : int

 fun G(x:bitstring{k}) : bitstring{l} = {
   var c : ciphertext option;
   var g : bitstring{l} = {0,1}^l;
   if (¬in_dom(x, LG)) {
```

```
      c = find_sie_snd(pk, x, LD); (* t_sie * qD *)
      if (c = None) {
        LG[x] = g;
      }
      else {
        LG[x] = LD[proj(c)] ⊕ proj(sie(pk, proj(c), x));
      }
    }
    return LG[x];
}

fun Dec1(c:ciphertext) : plaintext = {
  var r' : bitstring{k} option;
  var c' : (bitstring{k} * bitstring{l}) option;
  var r  : bitstring{k};
  var g, s, t, m : bitstring{l};
  if (q < qD) {
    q = q + 1;
    r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, c, r));
      g = LG[r];
      m = g ⊕ s;
    }
    else {
      if (in_dom(c, LD)) {
        m = LD[c];
      }
      else {
        c' = find_cie(pk, c, LD); (* t_cie * qD *)
        if (c' <> None) {
          (r, s, t) = proj(cie(pk, c, proj(c')));
          g = LD[proj(c')] ⊕ s;
          m = g ⊕ t;
        }
        else {
          m = {0,1}^l;
          LD[c] = m;
          LC = c :: LC;
        }
      }
    }
  }
  else {
    m = zero_l;
  }
  return m;
}

fun Dec2(c:ciphertext) : plaintext = {
  var r' : bitstring{k} option;
  var c' : (bitstring{k} * bitstring{l}) option;
  var r  : bitstring{k};
  var g, s, t, m : bitstring{l};
```

```
  if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
    q = q + 1;
    r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
    if (r' <> None) {
      r = proj(r');
      s = proj(sie(pk, c, r));
      g = LG[r];
      m = g ⊕ s;
    }
    else {
      if (in_dom(c, LD)) {
        m = LD[c];
      }
      else {
        c' = find_cie(pk, c, LD); (* t_cie * qD *)
        if (c' <> None) {
          (r, s, t) = proj(cie(pk, c, proj(c')));
          g = LD[proj(c')] ⊕ s;
          m = g ⊕ t;
        }
        else {
          if (cdef ∧ cie(pk, c, cstar) <> None) { (* t_cie *)
            (r, s, t) = proj(cie(pk, c, cstar));
            g = {0,1}^l;
            LG[r] = g;
            m = g ⊕ s;
          }
          else {
            m = {0,1}^l;
            LD[c] = m;
          }
        }
      }
    }
  }
  else {
    m = zero_l;
  }
  return m;
}

abs A1 = A1 {G, Dec1}
abs A2 = A2 {G, Dec2}

var zstar : bitstring{k} * bitstring{l}

fun Main() : bool = {
  var m0, m1 : plaintext;
  var b' : bool;
  var st : state;
  (pk, sk) = KG();
  LG = empty_map;
  LD = empty_map;
  LC = [];
  cdef = false;
```

```
    q = 0;
    (m0, m1, st) = A1();
    zstar = ({0,1}^k, {0,1}^l);
    cstar = f(pk, zstar);
    cdef = true;
    b' = A2(st, cstar);
    return true;
  }
}.

equiv OW_OW1 : OW.Main ∼ OW1.Main :
 true ⟹ in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩).
proof.
 inline B; derandomize; wp.
 swap⟨2⟩ [11-12] -6.
 call (={pk,sk,LG,LD,cstar,cdef,q} ∧ cdef⟨1⟩ ∧
   (in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩))).
 wp.
 call (={pk,sk,LG,LD,cstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
   (in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩))).
 trivial.
save.

claim Pr_OW_OW1 : OW.Main[in_dom(cstar, LD)] ≤ OW1.Main[mem(cstar, LC)]
using OW_OW1.

op msb  : bitstring{k+l} → bitstring{k}.
op lsb  : bitstring{k+l} → bitstring{l}.
op [‖] : (bitstring{k}, bitstring{l}) → bitstring{k+l} as app_kl.

axiom app_inj : ∀ (z:bitstring{k+l}), (msb(z) ‖ lsb(z)) = z.

spec rnd_pair() :
  xy1 = ({0,1}^k, {0,1}^l) ∼ xy2 = {0,1}^(k+l) :
  true ⟹ xy1 = (msb(xy2), lsb(xy2)).

game OW2 = OW1
 var zstar' : bitstring{k+l}

 where Main = {
    var m0, m1 : plaintext;
    var b' : bool;
    var st : state;
    (pk, sk) = KG();
    LG = empty_map;
    LD = empty_map;
    LC = [];
    cdef = false;
    q = 0;
    (m0, m1, st) = A1();
    zstar' = {0,1}^(k+l);
    cstar = f(pk, (msb(zstar'), lsb(zstar')));
    cdef = true;
    b' = A2(st, cstar);
    return true;
```

```
 }.

equiv OW1_OW2 : OW1.Main ~ OW2.Main : true ⟹ ={cstar, LC}.
proof.
 app 7 7 (={pk,sk,st,LG,LD,LC,cdef,q}).
 call (={pk,sk,LG,LD,LC,cdef,q}).
 derandomize; trivial.
 app 2 2 (={pk,sk,st,LG,LD,LC,cdef,q,cstar}).
 wp; apply: rnd_pair(); trivial.
 auto (={pk,sk,LG,LD,LC,cdef,q,cstar}).
save.

claim Pr_OW1_OW2 : OW1.Main[mem(cstar, LC)] = OW2.Main[mem(cstar, LC)]
using OW1_OW2.


game OW3 = OW2
 var LZ : bitstring{k+l} list

 where Dec1 = {
   var r' : bitstring{k} option;
   var c' : (bitstring{k} * bitstring{l}) option;
   var r  : bitstring{k};
   var g, s, t, m : bitstring{l};
   if (q < qD) {
     q = q + 1;
     r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
     if (r' <> None) {
       r = proj(r');
       s = proj(sie(pk, c, r));
       g = LG[r];
       m = g ⊕ s;
     }
     else {
       if (in_dom(c, LD)) {
         m = LD[c];
       }
       else {
         c' = find_cie(pk, c, LD); (* t_cie * qD *)
         if (c' <> None) {
           (r, s, t) = proj(cie(pk, c, proj(c')));
           g = LD[proj(c')] ⊕ s;
           m = g ⊕ t;
         }
         else {
           m = {0,1}^l;
           LD[c] = m;
           LZ = (fst(finv(sk, c)) ‖ snd(finv(sk, c))) :: LZ;
         }
       }
     }
   }
   else {
     m = zero_l;
   }
```

```
   return m;
 }

 and Main = {
   var m0, m1 : plaintext;
   var b' : bool;
   var st : state;
   (pk, sk) = KG();
   LG = empty_map;
   LD = empty_map;
   LZ = [];
   cdef = false;
   q = 0;
   (m0, m1, st) = A1();
   zstar' = {0,1}^(k+l);
   cstar = f(pk, (msb(zstar'), lsb(zstar')));
   cdef = true;
   b' = A2(st, cstar);
   return true;
 }.

set qD_pos, k_pos, l_pos.

equiv OW2_OW3 : OW2.Main ∼ OW3.Main :
  true ⟹
  (length(LZ⟨2⟩) ≤ qD) ∧
  (mem(cstar⟨1⟩, LC⟨1⟩) ⇒ mem(msb(zstar'⟨2⟩) ∥ lsb(zstar'⟨2⟩), LZ⟨2⟩)).
proof.
 app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
 derandomize; wp; apply: KG(); trivial.
 call (={pk,sk,LG,LD,cstar,cdef,q,zstar'} ∧ key_pair(pk,sk)⟨1⟩ ∧ cdef⟨1⟩ ∧
  cstar⟨1⟩ = f(pk⟨1⟩,(msb(zstar'⟨1⟩),lsb (zstar'⟨1⟩))) ∧
  length(LZ⟨2⟩) ≤ q⟨2⟩ ∧ q⟨2⟩ ≤ qD ∧
  (mem(cstar⟨1⟩, LC⟨1⟩) ⇒ mem(zstar'⟨2⟩, LZ⟨2⟩))).
 wp; rnd.
 call (={pk,sk,LG,LD,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧ ¬cdef⟨1⟩ ∧
  length(LZ⟨2⟩) ≤ q⟨2⟩ ∧ q⟨2⟩ ≤ qD ∧
  (∀ (z:bitstring{k+l}),
    mem(f(pk⟨1⟩, (msb(z), lsb(z))), LC⟨1⟩) ⇒ mem(z, LZ⟨2⟩))).
 trivial.
save.

claim Pr_OW2_OW3 :
  OW2.Main[mem(cstar, LC)] ≤ OW3.Main[mem(zstar', LZ) ∧ length(LZ) ≤ qD]
using OW2_OW3.

claim Pr_OW3 :
  OW3.Main[mem(zstar', LZ) ∧ length(LZ) ≤ qD] ≤ qD / (2 ^ (k+l))
compute.

claim conclusion :
  | CCA.Main[res] - 1 / 2 | ≤ OW.Main[res] + qD / (2 ^ (k+l)).
```