Verified Security of Redundancy-Free Encryption from Rabin and RSA

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

May 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 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 **3**

2 Redundancy-Free Encryption **4**
2.1 A Novel Redundancy-Free Scheme 5
2.2 Adaptive Security of ZAEP 5

3 A Primer on Verified Security **7**
3.1 User Perspective 9

4 Security Proof **10**

5 Instantiations **14**
5.1 Short Exponent RSA 14
5.2 Rabin Function 15
5.3 Practical Considerations 16
5.4 Comparison to 3-Round OAEP 16

6 Related Work **17**

7 Conclusion **18**

A EasyCrypt Input File **22**

1 Introduction

More than three decades after its inception by Rivest, Shamir and Adleman, the RSA algorithm [38] 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) [37], 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 [9] 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 [24, 25, 34] that convert encryption schemes that are just semantically secure under chosen-plaintext attacks [27] 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 [35] 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 [10], 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 [9, 26]. 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 [36]. This line of work was further developed in a series of papers, including [18, 31], 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 [17] 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 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 [19], 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.

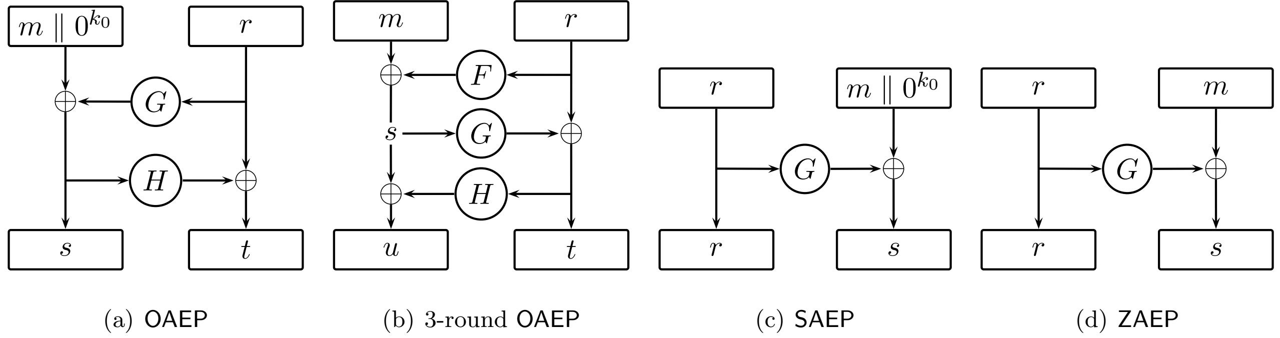
Our result is remarkable in two respects. First, ZAEP is surprisingly simple in comparison to the previous redundancy-free 3-round variant of OAEP that was shown to achieve IND-CCA security. Second, it constitutes the first application of verified security to a novel cryptographic construction. Specifically, we formally verify the security reduction (and the exact probability bound) of ZAEP using the EasyCrypt framework [3], 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 [28], 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 [9] 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. [26] 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 [36]. Although this latter reduction is still not tight, the redundancy resulting from padding the input message can be removed without breaking the proof.

Boneh [17] showed that by exploiting Coppersmith algorithm [19], 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 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



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

Figure 1: Asymmetric Encryption Paddings

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 [17] 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 [8]. 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:*

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

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 $\Pi = (\mathcal{KG}, \mathcal{E}, \mathcal{D})$, $\mathbf{Adv}_{\Pi}^{\mathsf{CCA}}(\mathcal{A})$, is defined as follows:*

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

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 $\boldsymbol{c}^*$. 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, $c_1, c_2 \in \{0,1\}^\ell$ and $r \in \{0,1\}^k$, 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*

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

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 [17]; 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 [3, 5] 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 [3] is a recent realization of the verified security paradigm. As its predecessor CertiCrypt [5], 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 where 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 [5].

As envisioned by Halevi [28] and Bellare and Rogaway [11], 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:

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

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

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

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*

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

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

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

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

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

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

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

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 $q/|A|$.

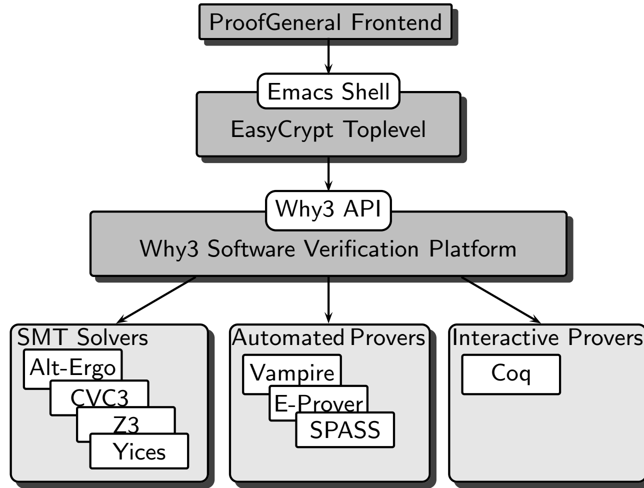


Figure 2: Overview of workflow in EasyCrypt

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 [16] 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 :
  L_G ← nil; c*_def ← false; q ← 0;
  (pk, sk) ← KG();
  (m_0, m_1, σ) ← A_1(pk);
  b ←$ {0,1};
  c* ← E_pk(m_b);
  c*_def ← true;
  b' ← A_2(c*, σ);
  return (b = b')

Oracle G(x) :
  if x ∉ dom(L_G) then
    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, s) ← f^{-1}_sk(c);
    g ← G(r);
    return g ⊕ s
  else return ⊥
```

```
Game OW :
  (pk, sk) ← KG();
  z ←$ {0,1}^{k+ℓ};
  (x, y) ← I(pk, f_pk(z));
  return (f_pk(x, y) = f_pk(z))

Adversary I(pk, z) :
  L_G, L_D ← nil; c*_def ← false; q ← 0;
  c* ← z; pk ← pk;
  (m_0, m_1, σ) ← A_1(pk);
  c*_def ← true;
  b' ← A_2(c*, σ);
  r ← find r ∈ dom(L_G). sie_pk(c*, r) ≠ ⊥;
  if r ≠ ⊥ then return (r, sie_pk(c*, r));
  else
    c ← find c ∈ dom(L_D). cie_pk(c*, c) ≠ ⊥;
    if c ≠ ⊥ then
      (r, s, t) ← cie_pk(c*, c); return (r, s)
    else return ⊥

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 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.

We begin 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:

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

Game $\boxed{\mathsf{G}_1}$ $\boxed{\mathsf{G}_2}$:
$\boldsymbol{L}_G \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$
$\mathbf{bad} \leftarrow \mathsf{false};\ \boldsymbol{r}^* \stackrel{\$}{\leftarrow} \{0,1\}^k;$
$(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}();$
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);\ b \stackrel{\$}{\leftarrow} \{0,1\};$
if $\boldsymbol{r}^* \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
  $\boldsymbol{g}^* \stackrel{\$}{\leftarrow} \{0,1\}^\ell;$ $\boxed{\boldsymbol{L}_G[\boldsymbol{r}^*] \leftarrow \boldsymbol{g}^*;}$
else
  $\mathbf{bad} \leftarrow \mathsf{true};$
  $\boxed{\boldsymbol{g}^* \leftarrow \boldsymbol{L}_G[\boldsymbol{r}^*];}$ $\boxed{\boldsymbol{g}^* \stackrel{\$}{\leftarrow} \{0,1\}^\ell;}$
$\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{g}^* \oplus m_b);\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$
return $(b = b')$

Oracle $G(x)$:
if $x = \boldsymbol{r}^*$ then $\mathbf{bad} \leftarrow \mathsf{true};$
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 G_3 :
$\boldsymbol{L}_G \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$
$\mathbf{bad} \leftarrow \mathsf{false};\ \boldsymbol{r}^* \stackrel{\$}{\leftarrow} \{0,1\}^k;$
$(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}();$
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);$
if $\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G)$ then $\mathbf{bad} \leftarrow \mathsf{true};$
$\boldsymbol{s}^* \stackrel{\$}{\leftarrow} \{0,1\}^\ell;$
$\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*);\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$
$b \stackrel{\$}{\leftarrow} \{0,1\};$
return $(b = b')$

Oracle $G(x)$:
if $x = \boldsymbol{r}^*$ then $\mathbf{bad} \leftarrow \mathsf{true};$
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 $\perp$

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.

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

Game G_4 :
$\boldsymbol{L}_G, \boldsymbol{L}'_G \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$
$\boldsymbol{r}^* \xleftarrow{\$} \{0,1\}^k;$
$\boldsymbol{s}^* \xleftarrow{\$} \{0,1\}^\ell;$
$\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*);$
$(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}();$
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);$
$\boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$
return true

Oracle $G(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
  if $x \notin \mathsf{dom}(\boldsymbol{L}'_G)$ then
    $\boldsymbol{L}_G[x] \xleftarrow{\$} \{0,1\}^\ell;$
  else
    $\boldsymbol{L}_G[x] \leftarrow \boldsymbol{L}'_G[x];$
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
  $\boldsymbol{q} \leftarrow \boldsymbol{q} + 1;$
  $r \leftarrow \mathsf{find}\ r \in \mathsf{dom}(\boldsymbol{L}_G).\ \mathsf{sie}_{\boldsymbol{pk}}(c, r) \neq \perp;$
  if $r \neq \perp$ then return $\boldsymbol{L}_G[r] \oplus \mathsf{sie}_{\boldsymbol{pk}}(c, r)$
  else
    $r \leftarrow \mathsf{find}\ r \in \mathsf{dom}(\boldsymbol{L}'_G).\ \mathsf{sie}_{\boldsymbol{pk}}(c, r) \neq \perp;$
    if $r \neq \perp$ then return $\boldsymbol{L}'_G[r] \oplus \mathsf{sie}_{\boldsymbol{pk}}(c, r)$
    else
      if $\boldsymbol{c}^*_{\mathbf{def}} \wedge \mathsf{cie}_{\boldsymbol{pk}}(c, \boldsymbol{c}^*) \neq \perp$ then
        $(r, s, t) \leftarrow \mathsf{cie}_{\boldsymbol{pk}}(c, \boldsymbol{c}^*);$
        $\boldsymbol{L}_G[r] \xleftarrow{\$} \{0,1\}^\ell;$ return $\boldsymbol{L}_G[r] \oplus s$
      else
        $(r, s) \leftarrow f^{-1}_{\boldsymbol{sk}}(c);$
        $m \xleftarrow{\$} \{0,1\}^\ell;\ \boldsymbol{L}'_G[r] \leftarrow m \oplus s;$
        return m;
else return $\perp$

Game G_5 :
$\boldsymbol{L}_G, \boldsymbol{L}_{\mathcal{D}} \leftarrow \mathsf{nil};\ \boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{false};\ \boldsymbol{q} \leftarrow 0;$
$\boldsymbol{r}^* \xleftarrow{\$} \{0,1\}^k;$
$\boldsymbol{s}^* \xleftarrow{\$} \{0,1\}^\ell;$
$\boldsymbol{c}^* \leftarrow f_{pk}(\boldsymbol{r}^*, \boldsymbol{s}^*);$
$(pk, sk) \leftarrow \mathcal{KG}();$
$(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk);$
$\boldsymbol{c}^*_{\mathbf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{c}^*, \sigma);$
return true

Oracle $G(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L}_G)$ then
  $c \leftarrow \mathsf{find}\ c \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{sie}_{\boldsymbol{pk}}(c, x) \neq \perp;$
  if $c \neq \perp$ then
    $\boldsymbol{L}_G[x] \leftarrow \boldsymbol{L}_{\mathcal{D}}[c] \oplus \mathsf{sie}_{\boldsymbol{pk}}(c, x);$
  else
    $\boldsymbol{L}_G[x] \xleftarrow{\$} \{0,1\}^\ell;$
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
  $\boldsymbol{q} \leftarrow \boldsymbol{q} + 1;$
  $r \leftarrow \mathsf{find}\ r \in \mathsf{dom}(\boldsymbol{L}_G).\ \mathsf{sie}_{\boldsymbol{pk}}(c, r) \neq \perp;$
  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
      $c' \leftarrow \mathsf{find}\ c' \in \mathsf{dom}(\boldsymbol{L}_{\mathcal{D}}).\ \mathsf{cie}_{\boldsymbol{pk}}(c, c') \neq \perp;$
      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
          $(r, s, t) \leftarrow \mathsf{cie}_{\boldsymbol{pk}}(c, \boldsymbol{c}^*);$
          $\boldsymbol{L}_G[r] \xleftarrow{\$} \{0,1\}^\ell;$ return $\boldsymbol{L}_G[r] \oplus s;$
        else
          $\boldsymbol{L}_{\mathcal{D}}[c] \xleftarrow{\$} \{0,1\}^\ell;$ return $\boldsymbol{L}_{\mathcal{D}}[c]$
else return $\perp$

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

characterize the event **bad** of G_3 in terms of the variables of G_4:

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

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:

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

We have hence that

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

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:

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

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

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

From which we obtain

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

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 $\boldsymbol{r}^* \in \mathsf{dom}(\boldsymbol{L}_G)$:

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

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:

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

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

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

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 3-round OAEP. Our proofs are inspired by [17] and rely on Coppersmith algorithm to find small integer roots of polynomials [19]:

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 $\ell \leq n$, 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 $\mathsf{RSA}[N, e]$ and a value r, 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 $0 \le \ell \le n-1$ 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:

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

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 $t_{\mathsf{sie}} \le t_{C(N,2)}$.

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 Rabin-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 RSA-ZAEP, all the required properties to achieve IND-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 Comparison to 3-Round OAEP

We compare our security result of Theorem 1 to the security result for 3-round OAEP (see Fig. 1(b)), the only other redundancy-free scheme based on the integer factoring assumption. The original result about the IND-CCA security of 3-round RSA-OAEP [35] relies on an intermediate reduction to the partial-domain one-wayness of RSA. Phan and Pointcheval [36] 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 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 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.

In comparison, 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 dominant factor $q_{\mathcal{D}}q_{\mathsf{G}}\ t_{\mathsf{sie}}$ in the time bound is linear on q_{G} and allows 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 [9]. However, their work only dealt with a weak notion of plaintext-awareness that provides a weaker, non-adaptive, notion of chosen-ciphertext security [32] rather than the adaptive notion of IND-CCA security considered in this paper. Subsequently, Bellare et al. [6] 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 [29], Bellare and Palacio [7], Dent [22], and Birket and Dent [12]. 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 [35, 36]; earlier work by Desai [23] achieves a similar goal, but in the setting of symmetric encryption. Libert and Quisquater [31] build a redundancy-free identity-based encryption scheme that achieves adaptive IND-CCA security. More recently, Boyen [18] proposes a compact redundancy-free encryption scheme based on the Gap-Diffie-Hellman problem [33]. 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, and can thus be used to achieve cryptographically meaningful guarantees. The computational soundness result of Abadi and Rogaway has been extended in many directions; we refer the reader to [20] 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 [30] 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 [2] 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 [28] 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 [13] 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 [15] and several protocols; we refer to [14] for a more detailed account of the examples proved with CryptoVerif. The work we report in this paper uses EasyCrypt [3], 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 [4]. 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. [21], 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 [4]. Thus, our work provides further evidence that, as stated in [3], "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 (four) of operations: the (redundant-free and IND-CPA) Bellare and Rogaway encryption scheme [8], 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.

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] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] M. Bellare and A. Palacio. Towards plaintext-aware public-key encryption without random oracles. In P. J. Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 48–62. Springer, Dec. 2004.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.

[9] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In A. D. Santis, editor, *EURO-CRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, May 1994.

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

[10] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In U. M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 399–416. Springer, May 1996.

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

[12] J. Birkett and A. W. Dent. Relations among notions of plaintext awareness. In R. Cramer, editor, *PKC 2008*, volume 4939 of *LNCS*, pages 47–64. Springer, Mar. 2008.

[13] 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.

[14] 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.

[15] 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.

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

[17] D. Boneh. Simplified OAEP for the RSA and Rabin functions. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 275–291. Springer, Aug. 2001.

[18] X. Boyen. Miniature CCA2 PK encryption: Tight security without redundancy. In K. Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 485–501. Springer, Dec. 2007.

[19] D. Coppersmith. Finding a small root of a univariate modular equation. In U. M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 155–165. Springer, May 1996.

[20] 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.

[21] 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.

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

[23] A. Desai. New paradigms for constructing symmetric encryption schemes secure against chosen-ciphertext attack. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 394–412. Springer, Aug. 2000.

[24] 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*, volume 1560 of *LNCS*, pages 53–68. Springer, Mar. 1999.

[25] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Aug. 1999.

[26] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 260–274. Springer, Aug. 2001.

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

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

[29] J. Herzog, M. Liskov, and S. Micali. Plaintext awareness via key registration. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 548–564. Springer, Aug. 2003.

[30] 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.

[31] B. Libert and J.-J. Quisquater. Identity based encryption without redundancy. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 285–300. Springer, June 2005.

[32] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd ACM STOC*. ACM Press, May 1990.

[33] 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*, volume 1992 of *LNCS*, pages 104–118. Springer, Feb. 2001.

[34] T. Okamoto and D. Pointcheval. REACT: Rapid Enhanced-security Asymmetric Cryptosystem Transform. In D. Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 159–175. Springer, Apr. 2001.

[35] D. H. Phan and D. Pointcheval. Chosen-ciphertext security without redundancy. In C.-S. Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 1–18. Springer, Nov. / Dec. 2003.

[36] D. H. Phan and D. Pointcheval. OAEP 3-round: A generic and secure asymmetric encryption padding. In P. J. Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 63–77. Springer, Dec. 2004.

[37] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In J. Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 433–444. Springer, Aug. 1992.

[38] 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

```
100 cnst k : int.
101 cnst l : int.
102 cnst qD : int.
103
104 cnst zero_k : bitstring{k}.
105 cnst zero_l : bitstring{l}.
106
107 type pkey.
108 type skey.
109 type plaintext  = bitstring{l}.
110 type ciphertext = bitstring{k} * bitstring{l}.
111
112 axiom k_pos : 0 ≤ k.
113
114 axiom l_pos : 0 ≤ l.
115
116 axiom qD_pos : 0 ≤ qD.
117
118 pop KG : () → pkey * skey.
119
120 op key_pair : (pkey, skey) → bool.
121
122 spec KG() : k1 = KG() ~ k2 = KG() : true ⟹ k1 = k2 ∧ key_pair(fst(k1), snd(k1)).
123
124 op f    : (pkey, bitstring{k} * bitstring{l}) → bitstring{k} * bitstring{l}.
125 op finv : (skey, bitstring{k} * bitstring{l}) → bitstring{k} * bitstring{l}.
126
127 axiom finv_l :
128   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
129   ∀ (xy:bitstring{k} * bitstring{l}), finv(sk, f(pk, xy)) = xy.
130
131 axiom finv_r :
132   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
133   ∀ (xy:bitstring{k} * bitstring{l}), f(pk, finv(sk, xy)) = xy.
134
135 (* Second-Input Extractor *)
136 op sie : (pkey, bitstring{k} * bitstring{l}, bitstring{k}) → bitstring{l} option.
137
138 axiom sie_spec :
139   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
140   ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, s:bitstring{l}),
141     sie(pk, y, r) = Some(s) ⟺ y = f(pk, (r, s)).
142
143 op find_sie_fst :
144   (pkey, bitstring{k} * bitstring{l}, (bitstring{k}, bitstring{l}) map) →
145   bitstring{k} option.
146
147 axiom find_sie_fst_correct :
148   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
149   ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
150     in_dom(fst(finv(sk, y)), L) ⇒
151     find_sie_fst(pk, y, L) = Some(fst(finv(sk, y))).
152
```

```
153 axiom find_sie_fst_complete :
154   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
155   ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
156     ¬in_dom(fst(finv(sk, y)), L) ⇒
157     find_sie_fst(pk, y, L) = None.
158
159 op find_sie_snd :
160   (pkey, bitstring{k}, (bitstring{k} * bitstring{l}, bitstring{l}) map) →
161   (bitstring{k} * bitstring{l}) option.
162
163 axiom find_sie_snd_correct :
164   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
165   ∀ (y:bitstring{k} * bitstring{l},
166           L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
167     find_sie_snd(pk, fst(finv(sk, y)), L) = None ⇒
168     ¬in_dom(y, L).
169
170 axiom find_sie_snd_complete :
171   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
172   ∀ (r:bitstring{k}, L:(bitstring{k} * bitstring{l}, bitstring{l}) map,
173           y:bitstring{k} * bitstring{l}),
174     find_sie_snd(pk, r, L) = Some(y) ⇒
175     in_dom(y, L) ∧ r = fst(finv(sk, y)).
176
177 (* Common-Input Extractor *)
178 op cie : (pkey, bitstring{k} * bitstring{l}, bitstring{k} * bitstring{l}) →
179          (bitstring{k} * bitstring{l} * bitstring{l}) option.
180
181 axiom cie_spec :
182   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
183   ∀ (y,z:bitstring{k} * bitstring{l}, r:bitstring{k}, s,t:bitstring{l}),
184     cie(pk, y, z) = Some((r, s, t)) ⟺
185     y = f(pk, (r, s)) ∧ z = f(pk, (r, t)) ∧ y <> z.
186
187 op find_cie :
188   (pkey, bitstring{k} * bitstring{l},
189    (bitstring{k} * bitstring{l},bitstring{l}) map) →
190   (bitstring{k} * bitstring{l}) option.
191
192 axiom find_cie_correct :
193   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
194   ∀ (y,z:bitstring{k} * bitstring{l},
195           L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
196     find_cie(pk, y, L) = Some(z) ⇒
197     in_dom(z, L) ∧ fst(finv(sk, z)) = fst(finv(sk, y)) ∧ y <> z.
198
199 axiom find_cie_complete :
200   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
201   ∀ (y:ciphertext, L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
202     find_cie(pk, y, L) = None ⇒
203     ∀ (y':ciphertext),
204       in_dom(y', L) ⇒ fst(finv(sk, y')) <> fst(finv(sk, y)) ∨ y = y'.
205
206 (** Derived lemmas, proved either here or in Coq (lemmas.v) *)
207
```

```
208 prover alt-ergo, cvc3.
209
210 lemma find_cie_correct' :
211   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
212   ∀ (y,z:bitstring{k} * bitstring{l},
213          L:(bitstring{k} * bitstring{l}, bitstring{l}) map),
214     find_cie(pk, y, L) = Some(z) ⇒
215     cie(pk, y, z) = Some((fst(finv(sk, y)), snd(finv(sk, y)), snd(finv(sk, z)))).
216
217 lemma sie_find_sie_fst :
218   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
219   ∀ (y:bitstring{k} * bitstring{l}, L:(bitstring{k}, bitstring{l}) map),
220     find_sie_fst(pk, y, L) <> None ⇒
221     sie(pk, y, proj(find_sie_fst(pk, y, L))) = Some(snd(finv(sk, y))).
222
223 axiom cie_find_cie :
224   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
225   ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext,bitstring{l}) map),
226     find_cie(pk, y, L) <> None ⇒
227     cie(pk, y, proj(find_cie(pk, y, L))) =
228     Some((fst(finv(sk, y)), snd(finv(sk, y)),
229           snd(finv(sk, proj(find_cie(pk, y,L)))))).
230
231 lemma find_sie_fst_upd :
232   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
233   ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, g:bitstring{l},
234          L:(bitstring{k}, bitstring{l}) map),
235     find_sie_fst(pk, y, L[r <- g]) = None ⟺
236     find_sie_fst(pk, y, L) = None ∧ fst(finv(sk, y)) <> r.
237
238 lemma find_sie_snd_cie :
239   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
240   ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext, bitstring{l}) map),
241     find_sie_snd(pk, fst(finv(sk, y)), L) <> None ⇒
242     find_cie(pk, y, L) <> None ⇒
243     let r,s,t = proj(cie(pk, y, proj(find_cie(pk, y, L)))) in y = f(pk, (r, s)).
244
245 axiom find_cie_find_sie_snd :
246   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
247   ∀ (y:bitstring{k} * bitstring{l}, L:(ciphertext, bitstring{l}) map),
248     find_cie(pk, y, L) = None ⇒
249     ¬in_dom(y, L) ⇒
250     find_sie_snd(pk, fst(finv(sk, y)), L) = None.
251
252 axiom find_sie_snd_upd :
253   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
254   ∀ (y:bitstring{k} * bitstring{l}, r:bitstring{k}, m:bitstring{l},
255          L:(ciphertext, bitstring{l}) map),
256     find_sie_snd(pk, r, L[y <- m]) = None ⟺
257     find_sie_snd(pk, r, L) = None ∧ fst(finv(sk, y)) <> r.
258
259 axiom find_cie_upd :
260   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
261   ∀ (y,y':ciphertext, m:bitstring{l}, L:(ciphertext, bitstring{l}) map),
262     find_cie(pk, y, L[y' <- m]) = None ⟺
```

```
263     find_cie(pk, y, L) = None ∧
264     (fst(finv(sk, y)) <> fst(finv(sk, y')) ∨ y = y').
265
266 axiom cie_spec' :
267   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
268   ∀ (y,z:bitstring{k} * bitstring{l}),
269     cie(pk, y, z) <> None ⇒ fst(finv(sk, y)) = fst(finv(sk, z)).
270
271 axiom find_cie_empty :
272   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
273   ∀ (y:bitstring{k} * bitstring{l}), find_cie(pk, y, empty_map) = None.
274
275 axiom find_sie_snd_empty :
276   ∀ (pk:pkey, sk:skey), key_pair(pk, sk) ⇒
277   ∀(r:bitstring{k}), find_sie_snd(pk, r, empty_map) = None.
278
279 lemma xor_2 : ∀ (x,y:bitstring{l}), x ⊕ (y ⊕ x) = y.
280
281 pred eq_except(M1, M2 : ('a, 'b) map, a : 'a) =
282   ∀ (w: 'a), w <> a ⇒ M1[w] = M2[w] ∧ (in_dom(w,M1) ⟺ in_dom(w,M2)).
283
284 lemma eqe_update_diff :
285   ∀(M1, M2 : ('a, 'b) map, a, a' : 'a, b : 'b),
286     eq_except(M1, M2, a) ⇒
287     eq_except(M1[a' <- b], M2[a' <- b], a).
288
289 lemma eqe_update_same_L :
290   ∀(M1, M2 : ('a, 'b) map, a : 'a, b : 'b),
291     eq_except(M1, M2, a) ⇒ eq_except(M1[a <- b], M2, a).
292
293 lemma eqe_update_same_R :
294   ∀(M1, M2 : ('a, 'b) map, a : 'a, b : 'b),
295     eq_except(M1, M2, a) ⇒ eq_except(M1, M2[a <- b], a).
296
297 type state.
298
299 adversary A1() : plaintext * plaintext * state
300   { bitstring{k} → bitstring{l}; ciphertext → plaintext }.
301
302 adversary A2(st:state, c:ciphertext) : bool
303   { bitstring{k} → bitstring{l}; ciphertext → plaintext }.
304
305 (*
306 ** Game CCA:
307 ** This is the standard CCA experiment
308 *)
309 game CCA = {
310  var pk    : pkey
311  var sk    : skey
312  var LG    : (bitstring{k}, bitstring{l}) map
313  var cstar : ciphertext
314  var cdef  : bool
315  var q     : int
316
317  fun G(x:bitstring{k}) : bitstring{l} = {
```

```
318     var g : bitstring{l} = {0,1}^l;
319     if (¬in_dom(x, LG)) {
320       LG[x] = g;
321     }
322     return LG[x];
323   }
324 
325   fun Enc(m:plaintext) : ciphertext = {
326     var g : bitstring{l};
327     var r : bitstring{k} = {0,1}^k;
328     g = G(r);
329     return f(pk, (r, g ⊕ m));
330   }
331 
332   fun Dec(c:ciphertext) : plaintext = {
333     var r : bitstring{k};
334     var g, s, m : bitstring{l};
335     if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
336       q = q + 1;
337       (r, s) = finv(sk, c);
338       g = G(r);
339       m = g ⊕ s;
340     }
341     else {
342       m = zero_l;
343     }
344     return m;
345   }
346 
347   abs A1 = A1 {G, Dec}
348   abs A2 = A2 {G, Dec}
349 
350   fun Main() : bool = {
351     var m0, m1 : plaintext;
352     var b, b' : bool;
353     var st : state;
354     (pk, sk) = KG();
355     LG = empty_map;
356     cdef = false;
357     q = 0;
358     (m0, m1, st) = A1();
359     b = {0,1};
360     cstar = Enc(b ? m0 : m1);
361     cdef = true;
362     b' = A2(st, cstar);
363     return (b = b');
364   }
365 }.
366 
367 
368 (*
369 ** Game G1:
370 ** - Introduce bad
371 ** - Hoist sampling of rstar
372 ** - Inline Enc(mb), G(rstar) in Main and remove Enc procedure
```

```
373 *)
374 game G1 = CCA
375  var rstar : bitstring{k}
376  var gstar : bitstring{l}
377  var bad   : bool
378
379  where G = {
380    var g : bitstring{l} = {0,1}^l;
381    if(x = rstar) { bad = true; }
382    if (¬in_dom(x, LG)) {
383      LG[x] = g;
384    }
385    return LG[x];
386  }
387
388  and Main = {
389    var m0, m1 : plaintext;
390    var b, b' : bool;
391    var st : state;
392    (pk, sk) = KG();
393    rstar = {0,1}^k;
394    bad = false;
395    LG = empty_map;
396    cdef = false;
397    q = 0;
398    (m0, m1, st) = A1();
399    b = {0,1};
400    if (¬in_dom(rstar, LG)) {
401      gstar = {0,1}^l;
402      LG[rstar] = gstar;
403    }
404    else {
405      bad = true;
406      gstar = LG[rstar];
407    }
408    cstar = f(pk, (rstar, gstar ⊕ (b ? m0 : m1)));
409    cdef = true;
410    b' = A2(st, cstar);
411    return (b = b');
412  }.
413
414 prover alt-ergo.
415 unset all.
416
417 equiv CCA_G1 : CCA.Main ~ G1.Main : true ⟹ ={res}.
418 proof.
419  inline⟨1⟩ Enc, G; derandomize.
420  call (={pk,sk,LG,cstar,cdef,q}); wp.
421  auto (={pk,sk,LG,cdef,q} ∧ ¬cdef⟨2⟩).
422  swap⟨1⟩ 2 1; trivial.
423 save.
424
425 claim Pr_CCA_G1 : CCA.Main[res] = G1.Main[res] using CCA_G1.
426
427
```

```
428 (*
429 ** Game G2:
430 ** Replace inlined G(rstar) by a random sampling in Main
431 *)
432 game G2 = G1
433  where Main = {
434    var m0, m1 : plaintext;
435    var b, b' : bool;
436    var st : state;
437    (pk, sk) = KG();
438    rstar = {0,1}^k;
439    bad = false;
440    LG = empty_map;
441    cdef = false;
442    q = 0;
443    (m0, m1, st) = A1();
444    b = {0,1};
445    gstar = {0,1}^l;
446    if (in_dom(rstar, LG)) { bad = true; }
447    cstar = f(pk, (rstar, gstar ⊕ (b ? m0 : m1)));
448    cdef = true;
449    b' = A2(st, cstar);
450    return (b = b');
451  }.
452
453 set eqe_update_diff, eqe_update_same_L, eqe_update_same_R.
454
455 equiv G1_G2 : G1.Main ~ G2.Main : true ⟹ ={bad} ∧ (¬bad⟨1⟩ ⇒ ={res}).
456 proof.
457  call upto (bad) with
458   (={pk,sk,cstar,rstar,gstar,cdef,q} ∧
459    (bad⟨1⟩ ⟺ in_dom(rstar⟨2⟩, LG⟨2⟩)) ∧ eq_except(LG⟨1⟩, LG⟨2⟩, rstar⟨1⟩)).
460  derandomize; wp.
461  call upto (bad) with
462   (={pk,sk,LG,rstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧ (bad⟨1⟩ ⟺ in_dom(rstar⟨2⟩, LG⟨2⟩))).
463  trivial.
464 save.
465
466 unset eqe_update_diff, eqe_update_same_L, eqe_update_same_R.
467
468 claim Pr_G1_G2 : | G1.Main[res] - G2.Main[res] | ≤ G2.Main[bad]
469 using G1_G2.
470
471
472 (*
473 ** Game G3:
474 ** Use optimistic sampling to sample sstar instead of gstar, where
475 **
476 ** G2: gstar = {0,1}^l; sstar = gstar ⊕ mb; cstar = f(rstar, sstar)
477 ** G3: sstar = {0,1}^k; gstar = sstar ⊕ mb; cstar = f(rstar, sstar)
478 **
479 ** Remove dependency of b' from b by eliminating gstar as dead-code
480 ** and postponing sampling b
481 *)
482 game G3 = G2
```

```
483  var sstar : bitstring{l}
484
485  where Main = {
486    var m0, m1 : plaintext;
487    var b, b' : bool;
488    var st : state;
489    (pk, sk) = KG();
490    rstar = {0,1}^k;
491    bad = false;
492    LG = empty_map;
493    cdef = false;
494    q = 0;
495    (m0, m1, st) = A1();
496    if (in_dom(rstar, LG)) { bad = true; }
497    sstar = {0,1}^l;
498    cstar = f(pk, (rstar, sstar));
499    cdef = true;
500    b' = A2(st, cstar);
501    b = {0,1};
502    return (b = b');
503  }.
504
505 set xor_l_cancel, xor_l_zero_r, xor_l_assoc.
506
507 equiv G2_G3 : G2.Main ~ G3.Main : true ==> ={bad,res}.
508 proof.
509  swap⟨2⟩ 13 -5.
510  call (={pk,sk,LG,rstar,cstar,cdef,q,bad}); wp.
511  rnd (sstar ⊕ (b ? m0 : m1)⟨2⟩); wp; rnd.
512  call (={pk,sk,LG,rstar,cdef,q,bad} ∧ ¬cdef⟨1⟩).
513  derandomize; trivial.
514 save.
515
516 unset xor_l_cancel, xor_l_zero_r, xor_l_assoc.
517
518 claim Pr_G2_G3  : G2.Main[res] = G3.Main[res] using G2_G3.
519
520 claim Pr_G2_G3' : G2.Main[bad] = G3.Main[bad] using G2_G3.
521
522 claim Pr_G3     : G3.Main[res] = 1 / 2 compute.
523
524
525 (*
526 ** Game G4:
527 ** Introduce LG' to store implicitly-defined values of G(r)
528 ** Inline calls to G in Dec
529 ** Apply optimistic-sampling to sample m rather than LG'[r] in Dec
530 *)
531 game G4 = G3
532  var LG' : (bitstring{k}, bitstring{l}) map
533
534  where G = {
535    var g : bitstring{l} = {0,1}^l;
536    if (¬in_dom(x, LG)) {
537      if (¬in_dom(x, LG')) {
```

```
538         LG[x] = g;
539       }
540       else {
541         LG[x] = LG'[x];
542       }
543     }
544     return LG[x];
545   }
546
547   and Dec = {
548     var r' : bitstring{k} option;
549     var c' : (bitstring{k} * bitstring{l}) option;
550     var r  : bitstring{k};
551     var g, s, t, m : bitstring{l};
552     if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
553       q = q + 1;
554       r' = find_sie_fst(pk, c, LG);
555       if (r' <> None) {
556         r = proj(r');
557         s = proj(sie(pk, c, r)); (* c = f(r, s) *)
558         g = LG[r];
559         m = g ⊕ s;
560       }
561       else {
562         r' = find_sie_fst(pk, c, LG');
563         if (r' <> None) {
564           r = proj(r');
565           s = proj(sie(pk, c, r)); (* c = f(r, s) *)
566           g = LG'[r];
567           m = g ⊕ s;
568         }
569         else {
570           if (cdef ∧ cie(pk, c, cstar) <> None) {
571             (r, s, t) = proj(cie(pk, c, cstar));
572             (* c = f(r, s) ∧ cstar = f(r, t) *)
573             g = {0,1}^l;
574             LG[r] = g;
575             m = g ⊕ s;
576          }
577          else {
578            (r, s) = finv(sk, c);
579            m = {0,1}^l;
580            LG'[r] = m ⊕ s;
581           }
582         }
583       }
584     }
585     else {
586       m = zero_l;
587     }
588     return m;
589   }
590
591   and Main = {
592     var m0, m1 : plaintext;
```

```
593     var b' : bool;
594     var st : state;
595     (pk, sk) = KG();
596     rstar = {0,1}^k;
597     sstar = {0,1}^l;
598     cstar = f(pk, (rstar, sstar));
599     LG = empty_map;
600     LG' = empty_map;
601     cdef = false;
602     q = 0;
603     (m0, m1, st) = A1();
604     cdef = true;
605     b' = A2(st, cstar);
606     return true;
607   }.
608
609 set find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst,
610   xor_l_cancel, xor_l_zero_r, xor_l_assoc,
611   cie_spec', cie_spec, finv_l, finv_r.
612
613 equiv G3_G4_Dec : G3.Dec ~ G4.Dec :
614  (={pk,sk,rstar,sstar,cstar,cdef,q} ∧
615   (key_pair(pk, sk) ∧ cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
616  (bad⟨1⟩ ⟺
617   (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
618  (∀ (x:bitstring{k}),
619     in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
620  (∀ (x:bitstring{k}),
621     ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
622     in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
623  (∀ (x:bitstring{k}),
624     ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
625 proof.
626  if; [ | trivial].
627  inline G.
628  case⟨2⟩: find_sie_fst(pk,c,LG) <> None.
629  condf⟨1⟩ at 6; [ | trivial].
630  condt⟨2⟩ last; [ | trivial ].
631  derandomize; wp; trivial.
632  condf⟨2⟩ last; [ | trivial].
633  case⟨2⟩: find_sie_fst(pk,c,LG') <> None.
634  condt⟨2⟩ last; [ | trivial].
635  condf⟨1⟩ at 6; [ | trivial].
636  derandomize; wp; trivial.
637  condf⟨2⟩ last; [ | trivial].
638  condt⟨1⟩ at 6; [ | trivial].
639  case⟨1⟩: cdef ∧ cie(pk,c, cstar) <> None.
640  condt⟨1⟩ at 5; [ | trivial].
641  condt⟨2⟩ last; [ | trivial].
642  derandomize; trivial.
643  derandomize; wp. rnd (m_0 ⊕ snd(finv(sk,c)⟨2⟩)); trivial.
644  unset xor_l_assoc.
645  trivial.
646 save.
647
```

```
648 equiv G3_G4_Dec : G3.Dec ∼ G4.Dec :
649  (={pk,sk,rstar,sstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
650   (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
651  (bad⟨1⟩ ⟺
652   (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
653  (∀ (x:bitstring{k}),
654    in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
655  (∀ (x:bitstring{k}),
656    ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
657    in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
658  (∀ (x:bitstring{k}),
659    ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
660 proof.
661  if; [ | trivial].
662  inline G.
663  case⟨2⟩: find_sie_fst(pk,c,LG) <> None.
664  condf⟨1⟩ at 6; [ | trivial].
665  condt⟨2⟩ last; [ | trivial].
666  derandomize; wp; trivial.
667  condf⟨2⟩ last; [ | trivial].
668  case⟨2⟩: find_sie_fst(pk,c,LG') <> None.
669  condt⟨2⟩ last; [ | trivial].
670  condf⟨1⟩ at 6; [ | trivial].
671  derandomize; wp; trivial.
672  condf⟨2⟩ last; [ | trivial].
673  condt⟨1⟩ at 6; [ | trivial].
674  case⟨1⟩: cdef ∧ cie(pk,c,cstar) <> None.
675  condt⟨1⟩ at 5; [ | trivial].
676  condt⟨2⟩ last; [ | trivial].
677  derandomize; trivial.
678  set xor_l_assoc.
679  derandomize; wp; rnd (m_0 ⊕ snd(finv(sk,c)⟨2⟩)); trivial.
680  unset xor_l_assoc.
681  trivial.
682 save.
683
684 unset find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst,
685   xor_l_cancel, xor_l_zero_r, xor_l_assoc, cie_spec', cie_spec.
686
687 equiv G3_G4 : G3.Main ∼ G4.Main : true ⟹
688  (bad⟨1⟩ ⟺
689   (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩).
690  app 1 1 ={pk,sk} ∧ key_pair(pk,sk)⟨1⟩.
691  derandomize; wp; apply: KG(); trivial.
692  swap⟨1⟩ -11; swap⟨1⟩ 9 -6.
693  call
694   (={pk,sk,rstar,sstar,cstar,cdef,q} ∧
695    (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
696   (bad⟨1⟩ ⟺
697     (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
698   (∀ (x:bitstring{k}),
699     in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
700   (∀ (x:bitstring{k}),
701     ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
702     in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
```

```
703   (∀ (x:bitstring{k}),
704     ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
705  wp.
706  call
707   (={pk,sk,rstar,sstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
708    (key_pair(pk, sk)⟨1⟩) ∧ (cstar = f(pk, (rstar,sstar)))⟨2⟩ ∧
709   (bad⟨1⟩ ⟺
710     (in_dom(fst(finv(sk, cstar)), LG) ∨ in_dom(fst(finv(sk, cstar)), LG'))⟨2⟩) ∧
711   (∀ (x:bitstring{k}),
712     in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ∧ LG⟨1⟩[x] = LG⟨2⟩[x]) ∧
713   (∀ (x:bitstring{k}),
714     ¬in_dom(x, LG⟨2⟩) ⇒ in_dom(x, LG⟨1⟩) ⇒
715     in_dom(x, LG'⟨2⟩) ∧ LG⟨1⟩[x] = LG'⟨2⟩[x]) ∧
716   (∀ (x:bitstring{k}),
717     ¬in_dom(x, LG⟨1⟩) ⇒ ¬in_dom(x, LG⟨2⟩) ∧ ¬in_dom(x, LG'⟨2⟩))).
718  trivial.
719 save.
720
721 unset finv_l, finv_r.
722
723 claim Pr_G3_G4 :
724   G3.Main[bad] =
725   G4.Main[in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG')]
726 using G3_G4.
727
728
729 (*
730 ** Game G5:
731 ** Introduce LD
732 ** Ciphertexts that implicitly-define values of G(r) are stored in LD
733 ** Remove finv from Dec
734 *)
735 game G5 = G4
736  var LD : (bitstring{k} * bitstring{l}, bitstring{l}) map
737
738  where G = {
739    var c : ciphertext option;
740    var g : bitstring{l} = {0,1}^l;
741    if (¬in_dom(x, LG)) {
742      c = find_sie_snd(pk, x, LD);
743      if (c = None) {
744        LG[x] = g;
745      }
746      else {
747        LG[x] = LD[proj(c)] ⊕ proj(sie(pk, proj(c), x));
748      }
749    }
750    return LG[x];
751  }
752
753  and Dec = {
754    var r' : bitstring{k} option;
755    var c' : (bitstring{k} * bitstring{l}) option;
756    var r  : bitstring{k};
757    var g, s, t, m : bitstring{l};
```

```
758    if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
759      q = q + 1;
760      r' = find_sie_fst(pk, c, LG);
761      if (r' <> None) {
762        r = proj(r');
763        s = proj(sie(pk, c, r)); (* c = f(r, s) *)
764        g = LG[r];
765        m = g ⊕ s;
766      }
767      else {
768        if (in_dom(c, LD)) {
769          m = LD[c];
770        }
771        else {
772          c' = find_cie(pk, c, LD);
773          if (c' <> None) {
774            (r, s, t) = proj(cie(pk, c, proj(c')));
775            (* c = f(r, s) ∧ c' = f(r, t) *)
776            g = LD[proj(c')] ⊕ s;
777            m = g ⊕ t;
778          }
779          else {
780            if (cdef ∧ cie(pk, c, cstar) <> None) {
781              (r, s, t) = proj(cie(pk, c, cstar));
782              (* c = f(r, s) ∧ cstar = f(r, t) *)
783              g = {0,1}^l;
784              LG[r] = g;
785              m = g ⊕ s;
786            }
787            else {
788              m = {0,1}^l;
789              LD[c] = m;
790            }
791          }
792        }
793      }
794    }
795    else {
796      m = zero_l;
797    }
798    return m;
799  }
800
801  and Main = {
802    var m0, m1 : plaintext;
803    var b' : bool;
804    var st : state;
805    (pk, sk) = KG();
806    rstar = {0,1}^k;
807    sstar = {0,1}^l;
808    cstar = f(pk, (rstar, sstar));
809    bad = false;
810    LG = empty_map;
811    LD = empty_map;
812    cdef = false;
```

```
813    q = 0;
814    (m0, m1, st) = A1();
815    cdef = true;
816    b' = A2(st, cstar);
817    return true;
818  }.
819
820 set find_sie_fst_correct, find_sie_fst_complete, sie_find_sie_fst, xor_2.
821
822 equiv G4_G5_Dec : G4.Dec ~ G5.Dec :
823  (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk, sk)⟨1⟩ ∧
824   (∀ (x:ciphertext),
825     find_sie_fst(pk⟨1⟩, x, LG'⟨1⟩) = None ⟺
826     find_cie(pk⟨2⟩, x, LD⟨2⟩) = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
827   (∀ (r:bitstring{k}),
828     ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
829   (∀ (x:ciphertext),
830    let r,s = finv(sk⟨1⟩, x) in
831      (in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x]))).
832 proof.
833  if; [ | trivial].
834  case⟨1⟩: find_sie_fst(pk, c, LG) <> None.
835  condt last; trivial.
836  condf last; [ | trivial | trivial].
837  case⟨2⟩: in_dom(c, LD).
838  condt⟨2⟩ last; [ | trivial].
839  condt⟨1⟩ last; [ | trivial].
840  trivial.
841  condf⟨2⟩ last; [ | trivial].
842  case⟨1⟩: find_sie_fst(pk, c, LG') <> None.
843  condt⟨1⟩ last; [ | trivial].
844  condt⟨2⟩ last; [ | trivial].
845  trivial.
846  app 0 0
847   (={c,pk,sk,LG,cstar,cdef} ∧ key_pair(pk,sk)⟨1⟩ ∧
848    (∀ (x:ciphertext),
849       find_sie_fst(pk,x,LG')⟨1⟩ = None ⟺
850       find_cie(pk,x,LD)⟨2⟩ = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
851    (∀ (x: ciphertext),
852      let r,s = finv(sk⟨1⟩, x) in
853      in_dom(x,LD⟨2⟩) ⇒ in_dom(r,LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x]) ∧
854    (cdef⟨2⟩ <> true ∨ c⟨2⟩ <> cstar⟨2⟩) ∧
855    ¬in_dom(c,LD)⟨2⟩ ∧
856    let c' = find_cie(pk, c, LD)⟨2⟩ in
857      c' <> None ∧ in_dom(proj(c'), LD⟨2⟩)).
858  set find_cie_correct.
859  trivial.
860  unset find_cie_correct.
861  set sie_find_sie_fst, cie_find_cie, find_cie_correct',
862      cie_spec, finv_l, finv_r, xor_l_cancel, xor_l_zero_r, xor_l_assoc.
863  app 0 0 (
864   ={c,pk,sk,LG,cstar,cdef} ∧ key_pair (pk⟨1⟩,sk⟨1⟩) ∧
865   find_sie_fst(pk,c,LG')⟨1⟩ <> None ∧
866   (∀ (x_0:ciphertext),
867   let r,s = finv(sk⟨1⟩,x_0) in
```

```
868     in_dom(x_0,LD⟨2⟩) ⇒ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x_0]) ∧
869   let c' = find_cie(pk, c, LD)⟨2⟩ in
870   let r', s, t = proj(cie(pk, c, proj(c')))⟨2⟩ in
871   let r = proj(find_sie_fst(pk, c, LG'))⟨1⟩ in
872     c' <> None ∧
873     r = r' ∧
874     c⟨1⟩ = f(pk, (r, s))⟨1⟩ ∧
875     proj(c') = f(pk, (r, t))⟨2⟩ ∧
876     in_dom(proj(c'), LD⟨2⟩) ∧
877     r = fst(finv(sk⟨1⟩, proj(c'))))).
878  trivial.
879  app 0 0 (
880    let c' = find_cie(pk, c, LD)⟨2⟩ in
881    let _, s, t = proj(cie(pk, c, proj(c')))⟨2⟩ in
882    let r = proj(find_sie_fst(pk, c, LG'))⟨1⟩ in
883      sie(pk,c,r)⟨1⟩ = Some(s) ∧
884      LG'⟨1⟩[r] = t ⊕ LD⟨2⟩[proj(c')]).
885  trivial.
886  trivial.
887
888  condf⟨1⟩ last; [ | trivial].
889  condf⟨2⟩ last; [ | trivial].
890  case⟨1⟩: cdef ∧ cie(pk, c, cstar) <> None.
891  condt⟨1⟩ last; [ | trivial].
892  condt⟨2⟩ last; [ | trivial].
893  trivial.
894  condf⟨1⟩ last; [ | trivial].
895  condf⟨2⟩ last; [ | trivial].
896  set find_sie_fst_upd, find_sie_snd_upd, find_cie_upd.
897  trivial.
898 save.
899
900 timeout 5.
901
902 equiv G4_G5_G : G4.G ~ G5.G :
903   (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk, sk)⟨1⟩ ∧
904   (∀ (x:ciphertext),
905     find_sie_fst(pk⟨1⟩, x, LG'⟨1⟩) = None ⇒
906     find_cie(pk⟨2⟩, x, LD⟨2⟩) = None ∧ ¬in_dom(x, LD⟨2⟩)) ∧
907   (∀ (r:bitstring{k}),
908     ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
909   (∀ (x:ciphertext),
910     let r,s = finv(sk⟨1⟩, x) in
911     in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x])).
912 proof.
913  case⟨1⟩: ¬in_dom(x, LG).
914  condt last; [ | trivial | trivial].
915  case⟨1⟩: ¬in_dom(x, LG').
916  condt⟨1⟩ last; [ | trivial].
917  condt⟨2⟩ last; [ | trivial].
918  trivial.
919  condf⟨1⟩ last; [ | trivial].
920  condf⟨2⟩ last; [ | trivial].
921  unset xor_2.
922  set find_sie_snd_complete, xor_l_comm.
```

```
923  trivial.
924  app 0 0 (
925    ={x,pk,sk,LG} ∧ key_pair(pk,sk)⟨1⟩ ∧
926    let c' = find_sie_snd(pk,x,LD)⟨2⟩ in
927      c' <> None ∧ in_dom (x,LG')⟨1⟩ ∧
928      LG'⟨1⟩[fst(finv(sk⟨1⟩,proj(c')))] =
929      snd(finv(sk⟨1⟩,proj(c'))) ⊕ LD⟨2⟩[proj(c')]); trivial.
930  condf last; trivial.
931 save.
932
933 set find_sie_snd_empty, find_cie_empty.
934
935 equiv G4_G5 : G4.Main ~ G5.Main : true ⟹
936  (in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG'))⟨1⟩ ⇒
937  (in_dom(fst(finv(sk,cstar)), LG) ∨
938   find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None)⟨2⟩.
939 proof.
940  app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
941  derandomize; wp; apply: KG(); trivial.
942  auto
943   (={pk,sk,LG,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧
944   (∀ (x:ciphertext),
945     find_sie_fst(pk⟨1⟩,x,LG'⟨1⟩) = None ⟺
946     find_cie(pk⟨2⟩,x,LD⟨2⟩) = None ∧ ¬in_dom(x,LD⟨2⟩)) ∧
947   (∀ (r:bitstring{k}),
948     ¬in_dom(r,LG'⟨1⟩) ⟺ find_sie_snd(pk⟨2⟩, r, LD⟨2⟩) = None) ∧
949   (∀ (x:ciphertext),
950     let r,s = finv(sk⟨1⟩,x) in
951     in_dom(x, LD⟨2⟩) ⇒ in_dom(r, LG'⟨1⟩) ∧ LG'⟨1⟩[r] = s ⊕ LD⟨2⟩[x])).
952  trivial.
953 save.
954
955 claim Pr_G4_G5 :
956   G4.Main[in_dom(fst(finv(sk,cstar)), LG) ∨ in_dom(fst(finv(sk,cstar)), LG')] ≤
957   G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
958           find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None]
959 using G4_G5.
960
961
962 game OW = {
963  var pk    : pkey
964  var sk    : skey
965  var LG    : (bitstring{k}, bitstring{l}) map
966  var LD    : (bitstring{k} * bitstring{l}, bitstring{l}) map
967  var cstar : ciphertext
968  var cdef  : bool
969  var q     : int
970
971  fun G(x:bitstring{k}) : bitstring{l} = {
972    var c : ciphertext option;
973    var g : bitstring{l} = {0,1}^l;
974    if (¬in_dom(x, LG)) {
975      c = find_sie_snd(pk, x, LD); (* t_sie * qD *)
976      if (c = None) {
977        LG[x] = g;
```

```
978       }
979       else {
980         LG[x] = LD[proj(c)] ⊕ proj(sie(pk,proj(c), x));
981       }
982     }
983     return LG[x];
984   }
985
986   fun Dec(c:ciphertext) : plaintext = {
987     var r' : bitstring{k} option;
988     var c' : (bitstring{k} * bitstring{l}) option;
989     var r  : bitstring{k};
990     var g, s, t, m : bitstring{l};
991     if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
992       q = q + 1;
993       r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
994       if (r' <> None) {
995         r = proj(r');
996         s = proj(sie(pk, c, r));
997         g = LG[r];
998         m = g ⊕ s;
999       }
1000      else {
1001        if (in_dom(c, LD)) {
1002          m = LD[c];
1003        }
1004        else {
1005          c' = find_cie(pk, c, LD); (* t_cie * qD *)
1006          if (c' <> None) {
1007            (r, s, t) = proj(cie(pk, c, proj(c')));
1008            g = LD[proj(c')] ⊕ s;
1009            m = g ⊕ t;
1010          }
1011          else {
1012            if (cdef ∧ cie(pk, c, cstar) <> None) { (* t_cie *)
1013              (r, s, t) = proj(cie(pk, c, cstar));
1014              g = {0,1}^l;
1015              LG[r] = g;
1016              m = g ⊕ s;
1017            }
1018            else {
1019              m = {0,1}^l;
1020              LD[c] = m;
1021            }
1022          }
1023        }
1024      }
1025    }
1026    else {
1027      m = zero_l;
1028    }
1029    return m;
1030  }
1031
1032  abs A1 = A1 {G, Dec}
```

```
1033  abs A2 = A2 {G, Dec}
1034
1035  fun B(z:bitstring{k} * bitstring{l}) : bitstring{k} * bitstring{l} = {
1036    var m0, m1 : plaintext;
1037    var b' : bool;
1038    var r' : bitstring{k} option;
1039    var r : bitstring{k};
1040    var s, t : bitstring{l};
1041    var c : ciphertext option;
1042    var st : state;
1043    LG = empty_map;
1044    LD = empty_map;
1045    cstar = z;
1046    cdef = false;
1047    q = 0;
1048    (m0, m1, st) = A1();
1049    cdef = true;
1050    b' = A2(st, cstar);
1051    r' = find_sie_fst(pk, cstar, LG); (* t_sie * qG *)
1052    if (r' <> None) {
1053      r = proj(r');
1054      s = proj(sie(pk, cstar, r));
1055    }
1056    else
1057    {
1058      c = find_cie(pk, cstar, LD); (* t_cie * qD *)
1059      if (c <> None) {
1060        (r, s, t) = proj(cie(pk, cstar, proj(c)));
1061      }
1062      else {
1063        r = zero_k;
1064        s = zero_l;
1065      }
1066    }
1067    return (r, s);
1068  }
1069
1070  var xstar : bitstring{k}
1071  var ystar : bitstring{l}
1072
1073  fun Main() : bool = {
1074    var x : bitstring{k};
1075    var y : bitstring{l};
1076    (pk, sk) = KG();
1077    xstar = {0,1}^k;
1078    ystar = {0,1}^l;
1079    (x,y) = B(f(pk, (xstar,ystar)));
1080    return (f(pk, (x,y)) = f(pk, (xstar,ystar)));
1081  }
1082 }.
1083
1084 set find_cie_find_sie_snd, find_sie_snd_cie, find_cie_correct'.
1085
1086 equiv G5_OW : G5.Main ~ OW.Main : true ==>
1087  in_dom(fst(finv(sk⟨1⟩,cstar⟨1⟩)), LG⟨1⟩) ∨
```

```
1088  find_sie_snd(pk⟨1⟩,fst(finv(sk⟨1⟩,cstar⟨1⟩)), LD⟨1⟩) <> None ⇒
1089  ¬in_dom(cstar⟨2⟩, LD⟨2⟩) ⇒ res⟨2⟩.
1090 proof.
1091  app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
1092  derandomize; wp; apply: KG(); trivial.
1093  inline B; derandomize.
1094  app 15 13 (={pk,sk,LG,LD,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧
1095    (cstar = f(pk,(xstar,ystar)))⟨2⟩ ).
1096  auto (={pk,sk,LG,LD,cstar,cdef,q} ∧ key_pair(pk,sk)⟨1⟩); trivial.
1097  trivial.
1098 save.
1099
1100 unset find_cie_find_sie_snd, find_sie_snd_cie, find_cie_correct'.
1101
1102 claim Pr_G5_OW' :
1103   G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
1104           find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None] ≤
1105   OW.Main[res ∨ in_dom(cstar, LD)]
1106 using G5_OW.
1107
1108 claim Pr_OW :
1109   OW.Main[res ∨ in_dom(cstar,LD)] ≤ OW.Main[res] + OW.Main[in_dom(cstar,LD)]
1110 compute.
1111
1112 claim Pr_G5_OW :
1113   G5.Main[in_dom(fst(finv(sk,cstar)), LG) ∨
1114           find_sie_snd(pk,fst(finv(sk,cstar)), LD) <> None] ≤
1115   OW.Main[res] + OW.Main[in_dom(cstar, LD)].
1116
1117 claim CCA_OW :
1118   | CCA.Main[res] - 1 / 2 | ≤ OW.Main[res] + OW.Main[in_dom(cstar, LD)].
1119
1120 (*
1121 ** Follows a rather technical sequence of games to bound
1122 ** OW.Main[in_dom(cstar,LD)]
1123 *)
1124
1125 game OW1 = {
1126  var pk    : pkey
1127  var sk    : skey
1128  var LG    : (bitstring{k}, bitstring{l}) map
1129  var LD    : (bitstring{k} * bitstring{l}, bitstring{l}) map
1130  var LC    : ciphertext list
1131  var cstar : ciphertext
1132  var cdef  : bool
1133  var q     : int
1134
1135  fun G(x:bitstring{k}) : bitstring{l} = {
1136    var c : ciphertext option;
1137    var g : bitstring{l} = {0,1}^l;
1138    if (¬in_dom(x, LG)) {
1139      c = find_sie_snd(pk, x, LD); (* t_sie * qD *)
1140      if (c = None) {
1141        LG[x] = g;
1142      }
```

```
1143      else {
1144        LG[x] = LD[proj(c)] ⊕ proj(sie(pk, proj(c), x));
1145      }
1146    }
1147    return LG[x];
1148  }
1149
1150  fun Dec1(c:ciphertext) : plaintext = {
1151    var r' : bitstring{k} option;
1152    var c' : (bitstring{k} * bitstring{l}) option;
1153    var r  : bitstring{k};
1154    var g, s, t, m : bitstring{l};
1155    if (q < qD) {
1156      q = q + 1;
1157      r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
1158      if (r' <> None) {
1159        r = proj(r');
1160        s = proj(sie(pk, c, r));
1161        g = LG[r];
1162        m = g ⊕ s;
1163      }
1164      else {
1165        if (in_dom(c, LD)) {
1166          m = LD[c];
1167        }
1168        else {
1169          c' = find_cie(pk, c, LD); (* t_cie * qD *)
1170          if (c' <> None) {
1171            (r, s, t) = proj(cie(pk, c, proj(c')));
1172            g = LD[proj(c')] ⊕ s;
1173            m = g ⊕ t;
1174          }
1175          else {
1176            m = {0,1}^l;
1177            LD[c] = m;
1178            LC = c :: LC;
1179          }
1180        }
1181      }
1182    }
1183    else {
1184      m = zero_l;
1185    }
1186    return m;
1187  }
1188
1189  fun Dec2(c:ciphertext) : plaintext = {
1190    var r' : bitstring{k} option;
1191    var c' : (bitstring{k} * bitstring{l}) option;
1192    var r  : bitstring{k};
1193    var g, s, t, m : bitstring{l};
1194    if (q < qD ∧ (¬cdef ∨ c <> cstar)) {
1195      q = q + 1;
1196      r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
1197      if (r' <> None) {
```

```
1198        r = proj(r');
1199        s = proj(sie(pk, c, r));
1200        g = LG[r];
1201        m = g ⊕ s;
1202      }
1203      else {
1204        if (in_dom(c, LD)) {
1205          m = LD[c];
1206        }
1207        else {
1208          c' = find_cie(pk, c, LD); (* t_cie * qD *)
1209          if (c' <> None) {
1210            (r, s, t) = proj(cie(pk, c, proj(c')));
1211            g = LD[proj(c')] ⊕ s;
1212            m = g ⊕ t;
1213          }
1214          else {
1215            if (cdef ∧ cie(pk, c, cstar) <> None) { (* t_cie *)
1216              (r, s, t) = proj(cie(pk, c, cstar));
1217              g = {0,1}^l;
1218              LG[r] = g;
1219              m = g ⊕ s;
1220            }
1221            else {
1222              m = {0,1}^l;
1223              LD[c] = m;
1224            }
1225          }
1226        }
1227      }
1228    }
1229    else {
1230      m = zero_l;
1231    }
1232    return m;
1233  }
1234
1235  abs A1 = A1 {G, Dec1}
1236  abs A2 = A2 {G, Dec2}
1237
1238  var zstar : bitstring{k} * bitstring{l}
1239
1240  fun Main() : bool = {
1241    var m0, m1 : plaintext;
1242    var b' : bool;
1243    var st : state;
1244    (pk, sk) = KG();
1245    LG = empty_map;
1246    LD = empty_map;
1247    LC = [];
1248    cdef = false;
1249    q = 0;
1250    (m0, m1, st) = A1();
1251    zstar = ({0,1}^k, {0,1}^l);
1252    cstar = f(pk, zstar);
```

```
1253     cdef = true;
1254     b' = A2(st, cstar);
1255     return true;
1256  }
1257 }.
1258
1259 equiv OW_OW1 : OW.Main ~ OW1.Main :
1260  true ⟹ in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩).
1261 proof.
1262  inline B; derandomize; wp.
1263  swap⟨2⟩ [11-12] -6.
1264  call (={pk,sk,LG,LD,cstar,cdef,q} ∧ cdef⟨1⟩ ∧
1265    (in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩))).
1266  wp.
1267  call (={pk,sk,LG,LD,cstar,cdef,q} ∧ ¬cdef⟨1⟩ ∧
1268    (in_dom(cstar, LD)⟨1⟩ ⇒ mem(cstar⟨2⟩, LC⟨2⟩))).
1269  trivial.
1270 save.
1271
1272 claim Pr_OW_OW1 : OW.Main[in_dom(cstar, LD)] ≤ OW1.Main[mem(cstar, LC)]
1273 using OW_OW1.
1274
1275 op msb  : bitstring{k+l} → bitstring{k}.
1276 op lsb  : bitstring{k+l} → bitstring{l}.
1277 op [‖] : (bitstring{k}, bitstring{l}) → bitstring{k+l} as app_kl.
1278
1279 axiom app_inj : ∀ (z:bitstring{k+l}), (msb(z) ‖ lsb(z)) = z.
1280
1281 spec rnd_pair() :
1282   xy1 = ({0,1}^k, {0,1}^l) ~ xy2 = {0,1}^(k+l) :
1283   true ⟹ xy1 = (msb(xy2), lsb(xy2)).
1284
1285 game OW2 = OW1
1286  var zstar' : bitstring{k+l}
1287
1288  where Main = {
1289    var m0, m1 : plaintext;
1290    var b' : bool;
1291    var st : state;
1292    (pk, sk) = KG();
1293    LG = empty_map;
1294    LD = empty_map;
1295    LC = [];
1296    cdef = false;
1297    q = 0;
1298    (m0, m1, st) = A1();
1299    zstar' = {0,1}^(k+l);
1300    cstar = f(pk, (msb(zstar'), lsb(zstar')));
1301    cdef = true;
1302    b' = A2(st, cstar);
1303    return true;
1304  }.
1305
1306 equiv OW1_OW2 : OW1.Main ~ OW2.Main : true ⟹ ={cstar, LC}.
1307 proof.
```

```
1308  app 7 7 (={pk,sk,st,LG,LD,LC,cdef,q}).
1309  call (={pk,sk,LG,LD,LC,cdef,q}).
1310  derandomize; trivial.
1311  app 2 2 (={pk,sk,st,LG,LD,LC,cdef,q,cstar}).
1312  wp; apply: rnd_pair(); trivial.
1313  auto (={pk,sk,LG,LD,LC,cdef,q,cstar}).
1314 save.
1315
1316 claim Pr_OW1_OW2 : OW1.Main[mem(cstar, LC)] = OW2.Main[mem(cstar, LC)]
1317 using OW1_OW2.
1318
1319
1320 game OW3 = OW2
1321  var LZ : bitstring{k+l} list
1322
1323  where Dec1 = {
1324    var r' : bitstring{k} option;
1325    var c' : (bitstring{k} * bitstring{l}) option;
1326    var r  : bitstring{k};
1327    var g, s, t, m : bitstring{l};
1328    if (q < qD) {
1329      q = q + 1;
1330      r' = find_sie_fst(pk, c, LG); (* t_sie * qG *)
1331      if (r' <> None) {
1332        r = proj(r');
1333        s = proj(sie(pk, c, r));
1334        g = LG[r];
1335        m = g ⊕ s;
1336      }
1337      else {
1338        if (in_dom(c, LD)) {
1339          m = LD[c];
1340        }
1341        else {
1342          c' = find_cie(pk, c, LD); (* t_cie * qD *)
1343          if (c' <> None) {
1344            (r, s, t) = proj(cie(pk, c, proj(c')));
1345            g = LD[proj(c')] ⊕ s;
1346            m = g ⊕ t;
1347          }
1348          else {
1349            m = {0,1}^l;
1350            LD[c] = m;
1351            LZ = (fst(finv(sk, c)) ‖ snd(finv(sk, c))) :: LZ;
1352          }
1353        }
1354      }
1355    }
1356    else {
1357      m = zero_l;
1358    }
1359    return m;
1360  }
1361
1362  and Main = {
```

```
1363    var m0, m1 : plaintext;
1364    var b' : bool;
1365    var st : state;
1366    (pk, sk) = KG();
1367    LG = empty_map;
1368    LD = empty_map;
1369    LZ = [];
1370    cdef = false;
1371    q = 0;
1372    (m0, m1, st) = A1();
1373    zstar' = {0,1}^(k+l);
1374    cstar = f(pk, (msb(zstar'), lsb(zstar')));
1375    cdef = true;
1376    b' = A2(st, cstar);
1377    return true;
1378  }.
1379
1380 set qD_pos, k_pos, l_pos.
1381
1382 equiv OW2_OW3 : OW2.Main ~ OW3.Main :
1383   true ⟹
1384   (length(LZ⟨2⟩) ≤ qD) ∧
1385   (mem(cstar⟨1⟩, LC⟨1⟩) ⇒ mem(msb(zstar'⟨2⟩) ‖ lsb(zstar'⟨2⟩), LZ⟨2⟩)).
1386 proof.
1387  app 1 1 ={pk,sk} ∧ key_pair(pk⟨1⟩,sk⟨1⟩).
1388  derandomize; wp; apply: KG(); trivial.
1389  call (={pk,sk,LG,LD,cstar,cdef,q,zstar'} ∧ key_pair(pk,sk)⟨1⟩ ∧ cdef⟨1⟩ ∧
1390   cstar⟨1⟩ = f(pk⟨1⟩,(msb(zstar'⟨1⟩),lsb (zstar'⟨1⟩))) ∧
1391   length(LZ⟨2⟩) ≤ q⟨2⟩ ∧ q⟨2⟩ ≤ qD ∧
1392   (mem(cstar⟨1⟩, LC⟨1⟩) ⇒ mem(zstar'⟨2⟩, LZ⟨2⟩))).
1393  wp; rnd.
1394  call (={pk,sk,LG,LD,cdef,q} ∧ key_pair(pk,sk)⟨1⟩ ∧ ¬cdef⟨1⟩ ∧
1395   length(LZ⟨2⟩) ≤ q⟨2⟩ ∧ q⟨2⟩ ≤ qD ∧
1396   (∀ (z:bitstring{k+l}),
1397     mem(f(pk⟨1⟩, (msb(z), lsb(z))), LC⟨1⟩) ⇒ mem(z, LZ⟨2⟩))).
1398  trivial.
1399 save.
1400
1401 claim Pr_OW2_OW3 :
1402   OW2.Main[mem(cstar, LC)] ≤ OW3.Main[mem(zstar', LZ) ∧ length(LZ) ≤ qD]
1403 using OW2_OW3.
1404
1405 claim Pr_OW3 :
1406   OW3.Main[mem(zstar', LZ) ∧ length(LZ) ≤ qD] ≤ qD / (2 ^ (k+l))
1407 compute.
1408
1409 claim conclusion :
1410   | CCA.Main[res] - 1 / 2 | ≤ OW.Main[res] + qD / (2 ^ (k+l)).
```