Computer-aided cryptography: some tools and applications

Gilles Barthe[1], François Dupressoir[1], Benjamin Grégoire[2], Benedikt Schmidt[1], and Pierre-Yves Strub[1]

[1] IMDEA Software Institute, Madrid, Spain
[2] INRIA Sophia-Antipolis Méditerranée, France

1 Introduction

The goal of modern cryptography is to design efficient constructions that simultaneously achieve some desired functionality and provable security against resource-bounded adversaries. Over the years, the realm of cryptography has expanded from basic functionalities such as encryption, authentication and key agreement, to elaborate functionalities such as zero-knowledge protocols, secure multi-party computation, and more recently verifiable computation. In many cases, these elaborate functionalities can only be achieved through cryptographic systems, in which several elementary constructions interact. As a consequence of the evolution towards more complex functionalities, cryptographic proofs have become significantly more involved, and more difficult to check. Several cryptographers have therefore advocated the use of tool-supported frameworks for building and verifying proofs; the most vivid recommendation for using computer support is elaborated in a farseeing article in which Halevi (2005) describes a potential approach for realizing this vision.

Besides increasing confidence in cryptographic proofs, tool-supported frameworks have the potential to address another prominent difficulty with provable security: because cryptographic proofs are very complex, it is common practice to reason about algorithmic descriptions of the cryptographic constructions, rather than about implementations. As a consequence, implementations of well-known and provably secure constructions are vulnerable to attacks, and regularly fail to provide their intended security guarantees. This uncomfortable gap between provable security and cryptographic engineering may be due to i. the mismatch between the powerful but abstract adversary models considered in proofs and "real-world" adversaries that may glean information about the secret data not only from the input and output to computations, but also from a host of side-channels (timing, power consumption or electromagnetic radiations...), and may even be able to interfere with the computation itself; ii. the fact that the object on which the security proof is performed is not the object that is implemented in practice, either due to a developer's (possibly malicious) mistake or even to an unjustified refinement when turning abstract algorithms into standard documents and recommendations. These points are the focus of the recent "real world" security approach to provable security, developed, most notably by Degabriele, Paterson, and Watson (2011). However, we believe that tool support is essential for accomodating the additional complexity introduced by dealing with implementation-level descriptions of cryptographic constructions and complex adversary models. In particular, even though security proofs themselves are difficult to automate, some automation is useful to help deal with the low-level implementation details.

2 Tools

Since 2005, we have been actively working on developing foundations and proving tool support for building and verifying the security of cryptographic constructions. To date, we have constructed several tools, ranging from general frameworks that can be applied to many classes of constructions to specialized frameworks which target a single class of constructions. We review both kinds of tools below, and provide for each of them a brief account of the rationale behind their design and of their applications so far. We also discuss the status of proofs in these tools.

2.1 CertiCrypt

CertiCrypt (Barthe et al., 2009) is a machine-checked framework built on top of the Coq proof assistant (The Coq development team, 2004). It supports the game-based code-based approach to cryptography (Shoup, 2004; Bellare and Rogaway, 2004), in which security notions and assumptions are formalized as probabilistic programs, also called games, and proofs are organized as sequences or trees of games. The proof is then performed by bounding the total distance (defined as the upper bound on the probability that a bounded adversary can induce distinguishable input-output behaviours) between the initial game, which expresses the security of the construction under study, and some subset of the leaf games, that represent computational hardness assumptions. From the perspective of formalization, the advantages of the game-based code-based approach are two-fold. First, it offers a rigorous formalism based on programming languages, a field with a long history of formal verification and extensive tool support. Second, organizing proofs as sequences of games is essential to tame their complexity, and opens the possibility to identify high-level principles whose application could potentially be automated. Indeed, CertiCrypt supports the code-centric view adopted in the game-based code-based approach by providing a deep embedding of an extensible probabilistic imperative language. It provides a denotational semantics of the language, based on the ALEA[3] library by Audebaud and Paulin-Mohring (2009), as well as an instrumented semantics that is used for modelling the computational complexity of programs and for defining the class of probabilistic polynomial-time program. In addition, CertiCrypt supports common forms of reasoning in cryptographic proofs through a rich set of verification methods for probabilistic programs, including a probabilistic relational Hoare

[3] `https://www.lri.fr/~paulin/ALEA/`

logic (pRHL), certified program transformations, and techniques widely used in cryptographic proofs such as eager/lazy sampling and failure events. Verification methods and logical proof rules are implemented in Coq, and proven correct with respect to the program semantics.

The main rationale behind the development of CertiCrypt was to provide strong trust guarantees on cryptographic proofs, rigorously formalizing game-based code-based assumptions and security notions and increasing trust in the proofs relating them. CertiCrypt was developed between 2005 and 2011, and was used to prove the security of several prominent cryptographic constructions, including the Full Domain Hash signature, the OAEP padding scheme, the Boneh-Frankling identity-based encryption scheme, zero-knowledge protocols, and hash functions into elliptic curves. An extension of CertiCrypt was used to reason about differential privacy, a notion that formalizes strong privacy guarantees in the context of privacy-preserving data mining. However, even such small examples reached the practical limits of the tool, due to the lack of automation, and additional burdens due to the formalization of arithmetic in $\mathbb{R}$.

2.2 EasyCrypt Prototype

The early EasyCrypt prototype (Barthe et al., 2011) is a tool-assisted framework for reasoning about the security of cryptographic constructions. As reported by Barthe et al. (2011), two key goals of the design of EasyCrypt were to improve automation (when compared to CertiCrypt) and to reuse existing program verification technology, in particular SMT solvers, leveraging their recent and future improvements. Thus, the main components of the initial prototype were a verification condition generator for CertiCrypt's probabilistic relational Hoare logic (pRHL) and a back-end interface to multiple theorem provers and SMT solvers, via the Why3 platform (Bobot et al., 2013).

These components enabled the semi-automated verification of pRHL judgments by interactively generating for each judgment a set of verification conditions that were sent to SMT solvers. Moreover, the initial prototype implemented a rudimentary algorithm for inferring loop invariants and adversary specifications, and allowed the user to provide specifications for loops and adversaries to palliate the incompleteness of the inference algorithms. In order to reduce the Trusted Computing Base and increase trust in EasyCrypt proofs, the original implementation of the verification condition generator produced an independently verifiable CertiCrypt proof of the validity of pRHL judgments, assuming the validity of the formulae discharged by the SMT solvers, hoping to later rely on proof-producing SMT solvers to obtain completely certified proofs in CertiCrypt.

The development of EasyCrypt was initiated in 2009, and the initial prototype was used to prove the security of several constructions, including the Cramer-Shoup encryption scheme, the Merkle-Damgård iterative hash function design, and of the ZAEP encryption scheme. As was done for CertiCrypt, an extension of the EasyCrypt prototype was developed to reason about differential privacy and its variant against computationally bounded adversaries, and was used to verify a smart-metering protocol. Even though this early prototype greatly simplified the writing of fully formal security proofs for primitives, it was still ill-suited to dealing with the complex layered cryptographic systems in which cryptographers are interested. In particular, without any abstraction mechanism, parallel or successive reductions could not be proved in isolation and combined in abstract ways, which led to important modularity and scalability issues.

2.3 EasyCrypt 1.0

Starting from 2012, a complete reimplementation of EasyCrypt (`https://www.easycrypt.info`) was therefore initiated, with the goal to overcome these limitations. In addition to dealing with the scalability issues mentioned above, the goals of the reimplementation were three-fold: first, consolidate the prototype into a robust platform that can be maintained and extended with reasonable effort; second, provide a versatile platform that supports automated proofs but also allows users to perform complex interactive proofs that interleave program verification and formalization of mathematics, which are intimately intertwined when formalizing cryptographic proofs; third, develop and implement the necessary foundations required to apply standard cryptographic reasoning principles that were not supported by the EasyCrypt prototype. To achieve these goals, the current version of EasyCrypt implements a probabilistic Hoare logic pHL for bounding the probability of post-conditions, and embeds both pRHL and pHL into an ambient logic that can for instance be used to perform hybrid arguments involving equivalences on parameterized programs coupled with inductive arguments on the parameters. In addition, it implements a module system and a theory mechanism that support compositional proofs through quantification over programs (as modules) and over types and values (through theories); using the module system and the new logics, we have been able to formalize cryptographic proofs that were out of reach of the initial prototype, including proofs of security for secure function evaluation, verifiable computation and authenticated key exchange protocols.

Example: security of a stateful random generator. As an example of an EasyCrypt proof, we now discuss a proof of security for a simple stateful random generator based on a pseudo-random function (PRF). We start by defining the assumption and security notion, and give a high-level overview of the proof. This simple proof does not fully exercise the module system. A more complex, albeit slightly more contrived, example can be found in the EasyCrypt tutorial (Barthe et al., 2014a).

We consider a set of *seeds* $\mathcal{S}$ and a set of *outputs* $\mathcal{O}$, equipped with unspecified but proper distributions $d_{\mathcal{S}}$ and $d_{\mathcal{O}}$. We denote sampling a variable x in $d_{\mathcal{S}}$ (resp. $d_{\mathcal{O}}$) with $x \xleftarrow{\$} \mathcal{S}$ (resp. $x \xleftarrow{\$} \mathcal{O}$). We assume a family of functions F from $\mathbb{N}$ to $\mathcal{O}$ indexed by $\mathcal{S}$. We construct a stateful random generator by using F in counter mode. The EasyCrypt code for these declarations and definitions is shown in Listing 1.1. The SRG construction is defined as a *module*, which defines a memory space containing global variables (here a variable s of type

$\mathcal{S}$ and a variable c in $\mathbb{N}$) and two procedures: i. a procedure init that, when called, simply samples the seed s in $d_{\mathcal{S}}$ and initializes the counter c to 0; and ii. a procedure next that, when queried, computes its output by applying F_{s} to c before incrementing c.

```
type S, O.
op F: S → ℕ → O.

module SRG = {
  var s:S
  var c:ℕ

  proc init(): unit = {
    s ←$ S;
    c = 0;
  }

  proc next(): O = {
    var r = F s c;
    c = c + 1;
    return r;
  }
}.
```

Listing 1.1. A Stateful Random Generator

Our objective is to show that SRG is a secure pseudo-random generator (PRG), under the assumption that F is a secure pseudo-random function (PRF). We first express these two notions formally, starting with the assumption on F.

Secure PRF. We say that F is a secure PRF if it is *computationally indistinguishable*, when used with a seed s sampled in $d_{\mathcal{S}}$, from the lazily sampled random function displayed in Listing 1.2.[4] We use the type (α,β) *map* of *finite maps* from α to β, using map0 to denote the empty map, dom m to denote the set of elements where m is defined, and standard notations for map updates and reads.

```
module RF = {
  var m:(ℕ,O) map

  proc init(): unit = { m = map0; }

  proc f(x:ℕ): O = {
    if (x ∉ dom m) m[x] ←$ O;
    return m[x];
  }
}.
```

Listing 1.2. Random Function

[4] This is a generalization of the standard cryptographic notion.

Computational indistinguishability is defined using a security experiment, parameterized by a construction and a distinguisher. *Module types* specifying the set of procedures expected to be implemented by a module are used to define parameterized modules. Module types themselves can be parameterized, allowing us to define, for example, the type of PRF distinguishers as modules that must implement a boolean procedure that may make oracle queries to a procedure f that take an argument in $\mathbb{N}$ and return some output in $\mathcal{O}$. We wrap the function family F into a module whose init procedure samples the seed that is used as index to F to answer f queries.

```
module PRFr = {
  var s:𝒮
  proc init(): unit = { s ←$ 𝒮; }
  proc f(x:ℕ): 𝒪 = { return F s x; }
}.

module type PRF = {
  proc init(): unit
  proc f(x:ℕ): 𝒪
}.

module type Distinguisher^PRF (P:PRF) = {
  proc distinguish(): bool { P.f }
}.

module IND^PRF (P:PRF,D:Distinguisher^PRF) = {
  proc main(): bool = {
    P.init();
    return D(P).distinguish();
  }
}.
```

Listing 1.3. Pseudo-Random Functions

We define the PRF advantage of a PRF distinguisher D against F as follows (writing $\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{M}}(\cdot)$ for $\mathsf{IND}^{\mathrm{PRF}}(\mathsf{M}, \cdot)$).

$$\mathsf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathsf{D}) = \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFr}}(\mathsf{D}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{RF}}(\mathsf{D}) : \mathsf{res}\right]$$

Intuitively, F is a secure PRF whenever, for all "efficient" PRF distinguisher D, $\mathsf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathsf{D})$ is "small". We do not formalize what it means for an algorithm to be efficient or for an advantage to be small, but simply related advantages of various adversaries against various constructions. In practice, further work is needed to argue for security, and the complexity of reductions, in particular, often needs to be analyzed.

Secure PRG. We say that SRG is a secure PRG if it is computationally indistinguishable from the true random generator that samples its successive outputs in $d_{\mathcal{O}}$ (Listing 1.4).

```
module RG = {
  proc init(): unit = { }
  proc next(): O = {
    var r <$- O;
    return r;
  }
}.
```

Listing 1.4. Random Generator

We use similar module types and modules to those used to define PRF security and say that SRG is a secure PRG if, for all "efficient" PRG distinguisher D, the following quantity is "small".

$$\mathsf{Adv}^{\text{PRG}}_{\mathsf{SRG}}(\mathsf{D}) = \Pr\left[\mathsf{IND}^{\text{PRG}}_{\mathsf{SRG}}(\mathsf{D}) : \mathsf{res}\right] - \Pr\left[\mathsf{IND}^{\text{PRG}}_{\mathsf{RG}}(\mathsf{D}) : \mathsf{res}\right]$$

```
module type PRG = {
  proc init(): unit
  proc next(): O
}.

module type Distinguisher^PRG (G:PRG) = {
  proc distinguish(): bool { G.next }
}.

module IND^PRG (G:PRG,D:Distinguisher^PRG) = {
  proc main(): bool = {
    G.init();
    return D(G).distinguish();
  }
}.
```

Listing 1.5. Pseudo-Random Generators

Security of SRG. We prove the security of the SRG construction by constructing, from any PRG distinguisher D, a PRF distinguisher D′ whose advantage bounds that of D. This involves simulating PRG oracles using only the PRF oracles and public information, so the PRG distinguisher can be run, and using its result to break the PRF security. In this case, the reduction is a simple reinterpretation of the SRG construction as part of an adversary against the underlying PRF and the security proof is a simple proof of equivalence. The adversary is defined as follows, first defining a module PRGp that simulates the SRG algorithm with only oracle-access to the PRF and then using the PRG security experiment as distinguisher. Note that the initialization of the PRF module's state is, crucially, left to the PRF experiment, allowing us to prove that, given a PRG distinguisher D, the module $\mathsf{D}^{\text{PRF}}_{\mathsf{D}}$ is a valid PRF distinguisher.

```
module D^PRF (D:Distinguisher^PRG,P:PRF) = {
  module PRGp = {
    proc init() = { SRG.c = 0; }
    proc next() = {
      var r = P.f(SRG.c);
      SRG.c = SRG.c + 1;
      return r;
    }
  }

  proc distinguish = IND^PRG_PRGp(D).main
}.
```

Listing 1.6. Reduction

Indeed, $\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}$ implements a boolean procedure distinguish that may make oracle queries only to the f procedure of its parameter P. This observation allows us to consider the modules $\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{P}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}})$ (for $\mathsf{P} <: \mathsf{PRF}$, and in particular for $\mathsf{P} \in \{\mathsf{PRFr}, \mathsf{RF}\}$), since they are well-typed, and we can prove the following equalities.

$$\Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{SRG}}(\mathsf{D}) : \mathsf{res}\right] = \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{PRFr}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] \tag{1}$$

$$\Pr\left[\mathsf{IND}^{\mathrm{PRG}}_{\mathsf{RG}}(\mathsf{D}) : \mathsf{res}\right] = \Pr\left[\mathsf{IND}^{\mathrm{PRF}}_{\mathsf{RF}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}) : \mathsf{res}\right] \tag{2}$$

Equality (1) is an easy program equivalence: by inlining $\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}}$.distinguish in the PRF security experiment, we see that the initialization code is the same in both programs. The PRG adversary is called on the left with the next oracle from module SRG, whereas it is called on the right using the next oracle from the PRGp simulation. We use EasyCrypt's adversary rule to reduce the equivalence of these two adversary calls to the observational equivalence of the oracles they query (with respect to some observation on the states). In this case, we use the fact that the value of SRG.s on the left (SRG.s{1}) is equal to the value of PRFr.k on the right (PRGr.k{2}), and that the counters are equal. This can be proved easily by inlining P.f.

The invariant used to prove Equality (2) is more involved. Indeed, the program on the left always returns freshly sampled randomness whereas the program on the right only returns freshly sampled randomness on fresh queries to the PRF. However, it is easy to see that the very structure of the $\mathsf{D}^{\mathrm{PRF}}$ construction imposes that all queries made to the PRF oracle are indeed fresh. We therefore use the following invariant, easily discharged by inlining and the SMT solvers, which allows us to prove that the results of each query to the next oracle are equally distributed.

$$\forall \mathsf{x}, \mathsf{x} \in \mathsf{dom}\ \mathsf{PRFi.m}\{2\} \Leftrightarrow 0 \le \mathsf{x} \le \mathsf{SRG.c}\{1\}$$

We then conclude the proof by rewriting in the advantage definitions, establishing the following equality for all PRG distinguisher D.

$$\mathsf{Adv}^{\mathrm{PRG}}_{\mathsf{SRG}}(\mathsf{D}) = \mathsf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathsf{D}^{\mathrm{PRF}}_{\mathsf{D}})$$

Comparison with the EasyCrypt prototype. EasyCrypt 1.0, learning from the prototype's shortcomings, focused on a "mostly-interactive" proof engine in which program logic judgments in pRHL and pHL are valid logical formulas and stateful programs become valid logical entities, that can be quantified and manipulated in abstract ways. In addition, the implementation of a theory mechanism allows us to develop libraries of *data structures* and *security notions* that can be instantiated at will by the end user, instead of having to re-formalize them in the context of each particular proof. For example, the security notions for PRF and PRG security described in the example above can be made into abstract theories, and instantiated with concrete types and distributions as needed. The SRG construction itself is described on abstract sets $\mathcal{S}$ and $\mathcal{O}$ and an abstract function family F that can be instantiated, for example, with bitstrings of appropriate lengths and the AES block cipher. This additional abstraction mechanism allows us to think modularly about proofs of implementations, but also about proofs of complex constructions: an proof obtained in an abstract setting can be fully refined (obtaining a proof for some implementation code), or simply instantiated with the types, operators and distributions used in another abstract construction (obtaining a generic proof of composition).

In addition, highly modular proofs allow us to identify widely-used high-level principles in cryptographic proofs. In turn, developing specialized libraries for these high-level principles allows us to support their application with minimal use of EasyCrypt's interactive core, simply proving straightforward program equivalences when refactoring cryptographic constructions to enable the application of the desired high-level principle.

2.4 ZooCrypt

The ZooCrypt framework (Barthe et al., 2013) provides tools for automatically analyzing and synthesizing padding-based encryption schemes. The class of padding-based encryption schemes consists of public-key encryption schemes built from one-way trapdoor permutations and random oracles. In practice, these primitives are often instantiated with the RSA function and hash functions.

Even though these building blocks are relatively simple and well understood, it is surprisingly difficult to find constructions that are simple, minimize ciphertext expansion and support tight reductions to the security of the employed one-way function. For example, Bellare and Rogaway (1994) proved security against chosen-ciphertext attacks (IND-CCA) for the OAEP scheme shown in Listing 1.7 under the one-way assumption.

$\mathsf{r} \xleftarrow{\$} \{0,1\}^k;\ \mathsf{s} = \mathsf{H}(\mathsf{r}) \oplus (\mathsf{m} \mid 0^l);\ \mathsf{t} = \mathsf{H}(\mathsf{s}) \oplus \mathsf{r};\ \textbf{return}\ \mathsf{f}(\mathsf{s}|\mathsf{t})$

Listing 1.7. OAEP encryption of message m

Later on, Shoup (2001) proved that it is impossible to reduce the security of OAEP to one-wayness and the proof must therefore be flawed. To regain confidence in the widely used OAEP scheme, Shoup (2001) and Fujisaki et al. (2001) developed new proofs for OAEP under stronger assumptions. Additionally, many schemes have been proposed that improve on various aspects of OAEP, for example by providing security under the weaker one-wayness assumption.

The goal of the ZooCrypt framework is to demonstrate that fully automated game-based proofs and computer-aided design are feasible in the domain of padding-based encryption schemes. ZooCrypt consists of two components: an analyzer that can decide efficiently whether an instance construction is secure, and a synthesizer that implements a smart generation algorithm for candidate instances. The analyzer combines efficient search procedures to prove the security of an instance using a custom proof system, and attack finding procedures based on symbolic models of cryptography. The custom proof system consists of a small number of high level proof rules that formalize the game hops used in such proofs. Using ZooCrypt, we have built a database that contains more than one million padding-based encryption schemes. To build this database, our tool has not only found many new schemes, it has also rediscovered most schemes from the literature (including proofs).

2.5 Generic Group Analyzer

The GenericGroupAnalyzer (Barthe et al., 2014b) is a tool to analyze cryptographic assumptions in generic group models. Barring a breakthrough in complexity theory, the hardness of cryptographic assumptions, such as the discrete logarithm problem in certain cyclic groups, cannot be proved in general models of computation. To sidestep this problem, a commonly used approach is to prove lower bounds on the runtime of *generic algorithms* that do not exploit the concrete representation of group elements. This approach was initiated by Nechaev (1994) and Shoup (1997) and a proof in the generic group model can be considered as a minimal requirement for a newly proposed cryptographic assumption.

The GenericGroupAnalyzer supports three types of problems: i. non-parametric problems where the group setting is fixed and the adversary obtains a fixed set of group elements; ii. parametric problems where the group setting and the size of the adversary input is parameterized; iii. interactive problems where the adversary can adaptively query oracles to obtain new group elements.

In the non-parametric mode, our tool takes a group setting and a specification of the right and left adversary input and returns either an upper bound on the winning probability or an algebraic attack on the assumption. The assumption shown in Listing 1.8 formalizes the decisional Diffie-Hellman problem in a bilinear group of Type II. Namely, the adversary must distinguish the triple $(g_2^x, g_2^y, g_2^{x*y}) \in \mathbb{G}_2^3$ from the triple $(g_2^x, g_2^y, g_2^z) \in \mathbb{G}_2^3$ for randomly sampled $x, y, z \in \mathbb{F}_p$ given blackbox access to a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and an isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$. For this input, our tool returns the distinguishing test $e(\psi(w_1), w_2) = e(g_1, w_3)$, where w_i denotes the i-th element of the triple.

```
iso G1 → G2. map G1 ∗ G2 → GT.

input [ x, y ] in G2.
```

```
input_left [ x*y ] in G2.
input_right [ z ] in G2.
```

Listing 1.8. Decisional Diffie-Hellman problem in bilinear group of Type II

Listing 1.9 shows the Diffie-Hellman exponent problem, where the adversary is given $g_1^y, g_1, g_1^x, \ldots, g_1^{x^{n-1}}, g_1^{x^{n+1}}, \ldots, g_1^{x^{2n}} \in \mathbb{G}_1$ for random $x, y \in \mathbb{F}_p$ in a symmetric bilinear group. To win, the adversary must distinguish $g_2^{y*x^n}$ from a random element in the target group $\mathbb{G}_2$. Our tool confirms that the winning probability of the adversary is negligible.

```
setting symmetric. levels 2. problem_type decisional.

input [ y, forall i in [0, n − 1]: x^i, forall j in [n + 1, 2*n]: x^j ] in G1.

challenge y*x^n in G2.
```

Listing 1.9. Parametric n-Diffie-Hellman-exponent-problem

Listing 1.10 shows the interactive LRSW problem introduced by Lysyanskaya et al. (2000). Here, the adversary gets $g_1^x, g_1^y \in \mathbb{G}_1$ and can additionally query the oracle O with an element m_O to obtain the triple $(A, A^y, A^{x+m_O*x*y}) \in \mathbb{G}_1^3$ for a randomly sampled $A \in \mathbb{G}_1$. To win, the adversary must compute a tuple $(A', V, W) \in (\mathbb{G}_1 \setminus \{1\}) \times \mathbb{G}_1 \times \mathbb{G}_1$ that satisifies the same relation for an $m \in \mathbb{F}_p^*$ of his choice that has not been queried to O. Our tool confirms that the winning probability of the adversary is negligible for a polynomial number of queries.

```
input [x, y] in G1.

oracle O(mo:Fp) = sample a; return [a, y*a, a*(x + mo*x*y)] in G1.

win (A':G1, V:G1, W:G1, m:Fp) =
  V = A'*y ∧ W = A'*(x + m*x*y) ∧ A' ≠ 0 ∧ mo ≠ m ∧ m ≠ 0.
```

Listing 1.10. Interactive LRSW problem

Our tool relies on a generalization of the master theorem introduced by Boneh et al. (2005) and uses SMT solvers for checking constraint satisfiability and computer algebra systems for linear algebra and Gröbner Basis computations.

3 Discussions and conclusions

The primary motivation for our work is to support the construction of independently verifiable proofs of security for cryptographic systems. It is indeed folklore that there is a very significant asymmetry between building and checking a formal proof, and that one can achieve trust in formal proofs by inspecting their statements and the definitions that they use, but without actually inspecting the proofs themselves. On this account, formal proofs provide a pragmatic solution to the unverifiability of cryptographic proofs, and allow the proof reader to shift his focus on checking that definitions and statements are indeed appropriate for the claimed results, which is another important source of mistakes in cryptography. The GenericGroupAnalyzer also makes a step in the direction of validating new security definitions by automatically proving new assumptions are secure with respect to a generic model of computation.

More generally, our experience with the various tools described here tends to show that a careful mix of interactivity and automation is highly desirable when dealing with complex proofs. However, rather than adopting the usual "mostly-automated" approach usually taken in general-purpose program verification, we choose to support a "mostly-interactive" approach to proof building. This allows us to deal with the complex number theoretic and algebraic arguments that appear in cryptography whilst using the automated techniques to discharge the trivial-but-tedious proof obligations generated by the program verification part of the tool. In addition, this mostly-interactive approach encourages the construction of layered tools that may make use of EasyCrypt as a back-end with little to no fear that automation will fail and break the proof. In turn, this allows the development of fully-automated – albeit specialized – tools, or of more intuitive proof construction interfaces that may be used, for example, to teach provable security.[5]

Finally, the development of a formal framework to reason about discrete probabilistic programs also allows developments outside of the realm of proofs. Indeed, we have recently developed an EasyCrypt-backed fault attack synthesis algorithm. Given an implementation and a *fault condition* (a set of final states known to yield usable information on the secrets), our algorithm finds variants of the program that follow a chosen fault model and fault policy and guarantee the fault condition, ensuring a successful attack.

About the tools. More information on the tools and projects presented in this chapter, including downloads, documentation and tutorials, can be obtained by contacting its authors at `appa14@projects.easycrypt.info` or by visiting `https://www.easycrypt.info`.

[5] For example, ZooCrypt's logic for CPA proofs has been used to construct a proof tutor, accessible through `https://www.easycrypt.info/trac/wiki/ZooCrypt`.

Bibliography

Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568—589, 2009.

Gilles Barthe, Benjamin Grégoire, and Santiago 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.

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago 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.

Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 1247–1260. ACM Press, 2013.

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014a. ISBN 978-3-319-10081-4. doi: 10.1007/978-3-319-10082-1_6. URL `http://dx.doi.org/10.1007/978-3-319-10082-1_6`.

Gilles Barthe, Edvard Fagerholm, Dario Fiore, John Mitchell, Andre Scedrov, and Benedikt Schmidt. Automated analysis of cryptographic assumptions in generic group models. In *Advances in Cryptology – CRYPTO 2014*, volume 8616 of *Lecture Notes in Computer Science*, pages 95–112, Heidelberg, 2014b. Springer.

Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111, Heidelberg, 1994. Springer.

Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. `http://eprint.iacr.org/`.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*. Université Paris-Sud, CNRS, INRIA, March 2013. Version 0.81.

Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 440–456, Aarhus, Denmark, May 22–26, 2005. Springer, Berlin, Germany.

Jean Paul Degabriele, Kenneth G. Paterson, and Gaven J. Watson. Provable security in the real world. *IEEE Security & Privacy*, 9(3):33–41, 2011.

Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.

Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

Anna Lysyanskaya, RonaldL. Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In Howard Heys and Carlisle Adams, editors, *Selected Areas in Cryptography*, volume 1758 of *Lecture Notes in Computer Science*, pages 184–199. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67185-5. doi: 10.1007/3-540-46513-8_14. URL `http://dx.doi.org/10.1007/3-540-46513-8_14`.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer, Berlin, Germany.

Victor Shoup. OAEP reconsidered. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.

Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.