The Foundational Cryptography Framework

Adam Petcher[1,2] and Greg Morrisett[1]

[1] Harvard University, Cambridge, Massachusetts, USA
apetcher,greg@seas.harvard.edu
[2] MIT Lincoln Laboratory, Lexington, Massachusetts, USA

Abstract. We present the Foundational Cryptography Framework (FCF) for developing and checking complete proofs of security for cryptographic schemes within a proof assistant. This is a general-purpose framework that is capable of modeling and reasoning about a wide range of cryptographic schemes, security definitions, and assumptions. Security is proven in the computational model, and the proof provides concrete bounds as well as asymptotic conclusions. FCF provides a language for probabilistic programs, a theory that is used to reason about programs, and a library of tactics and definitions that are useful in proofs about cryptography. The framework is designed to leverage fully the existing theory and capabilities of the Coq proof assistant in order to reduce the effort required to develop proofs.

Keywords: Cryptography, Protocol Verification, Proof Assistant, Coq

1 Introduction

Cryptographic algorithms and protocols are becoming more numerous, specialized, and complicated. As a result, it is likely that security vulnerabilities will slip by peer review. To address this problem, some cryptographers [7][16] have proposed an increased level of rigor and formality for cryptographic proofs. It is our hope that eventually, cryptographers will be able to describe cryptographic schemes and security proofs using a formal language, and the proofs can be checked automatically by a highly trustworthy mechanized proof checker.

To enable such mechanically-verified proofs, we have developed The Foundational Cryptography Framework (FCF). This framework embeds into the Coq proof assistant [17] a simple probabilistic programming language to allow the specification of cryptographic schemes, security definitions, and assumptions. The framework also includes useful theory, tactics, and definitions that assist with the construction of proofs of security. Once complete, the proof can be checked by the Coq proof checker. Facts proven in FCF include the security of El Gamal encryption [14], and of the encryption scheme described in Section 4 of this paper. We have also proven the security and correctness of the "tuple-set" construction of [11], and shown how this primitive can be used to construct a searchable symmetric encryption scheme supporting single keyword queries.

This is a complex and sophisticated construction with a proof of over 15000 lines of Coq code which includes a pair of core arguments involving more than 30 intermediate games.

FCF is heavily influenced by CertiCrypt [6], which was later followed by EasyCrypt [5]. CertiCrypt is a framework that is built on Coq, and allows the development of mechanized proofs of security in the computational model for arbitrary cryptographic constructions. Unfortunately, proof development in CertiCrypt is time-consuming, and the developer must spend a disproportionate amount of time on simple, uninteresting goals. To address these limitations, the group behind CertiCrypt developed EasyCrypt, which has a similar semantics and logic, and uses the Why3 framework and SMT solvers to improve proof automation. EasyCrypt takes a huge step forward in terms of usability and automation, but it sacrifices some trustworthiness due to that fact that the trusted computing base is larger and the basis of the mechanization is a set of axiomatic rules.

FCF is a *foundational* framework like CertiCrypt, in which the rules used to prove equivalence of programs (or any fact) are mechanized proofs derived from the semantics or other core definitions. An important difference between CertiCrypt and FCF is that CertiCrypt uses a deep embedding of a probabilistic programming language whereas FCF uses a shallow embedding (similar to [19]). The shallow embedding allows us to easily extend the language, and to make better use of Coq's tactic language and existing automated tactics to reduce the effort required to develop proofs. The result is a framework that is foundational and easily extensible, but in which proof development effort is greatly reduced.

2 Design Goals

Based on our experience working with EasyCrypt, we formulated a set of idealized design goals that a practical mechanized cryptography framework should satisfy. We believe that FCF achieves many of these goals, though there is still some room for improvement, as discussed in Section 5.

Familiarity. Security definitions and descriptions of cryptographic schemes should look similar to how they would appear in cryptography literature, and a cryptographer with no knowledge of programming language theory or proof assistants should be able to understand them. Furthermore, a cryptographer should be able to inspect and understand the foundations of the framework itself.

Proof Automation. The system should use automation to reduce the effort required to develop a proof. Ideally, this automation is extensible, so that the developer can produce tactics for solving new kinds of goals.

Trustworthiness. Proofs should be checked by a trustworthy procedure, and the core definitions (*e.g.*, programming language semantics) that must be inspected in order to trust a proof should be relatively simple and easy to understand.

Extensibility. It should be possible to directly incorporate any existing theory that has been developed for the proof assistant. For example, it should be possible to directly incorporate an existing theory of lattices in order to support cryptography that is based on lattices and their related assumptions.

Concrete Security. The security proof should provide concrete bounds on the probability that an adversary is able to defeat the scheme. Concrete bounds provide more information than asymptotic statements, and they inform the selection of values for system parameters in order to achieve the desired level of security in practice.

Abstraction. The system should support abstraction over types, procedures, proofs, and modules containing any of these items. Abstraction over procedures and primitive types is necessary for writing security definitions, and for reasoning about adversaries in a natural way. The inclusion of abstraction over proofs and structures adds a powerful mechanism for developing sophisticated abstract arguments that can be reused in future proofs.

Code Generation. The system should be able to generate code containing the procedures of the cryptographic scheme that was proven secure. This code can then be used for basic testing, prototyping, or as an executable model to which future implementations will be compared during testing.

3 Framework Components

In a typical cryptographic proof, we specify cryptographic schemes, security definitions, and (assumed) hard problems, and then we prove a reduction from a properly-instantiated security definition to one or more problems that are assumed to be hard. In other words, we assume the existence of an effective adversary against the scheme in question, and then prove that we can construct a procedure that can effectively solve a problem that is assumed to be hard. This reduction results in a contradiction that allows us to conclude that an effective adversary against the scheme cannot exist.

The cryptographic schemes, security definitions, and hard problems are probabilistic, and FCF provides a common probabilistic programming language (Section 3.1) for describing all three. Then we provide a denotational semantics (Section 3.1) that allows reasoning about the probability distributions that correspond to programs in this language. This semantics assigns a numeric value to an event in a probability distribution, and it also allows us to conclude that two distributions are equivalent and we can replace one with the other (which supports the game-hopping style of [7]).

It can be cumbersome to work directly in the semantics, so we provide a theory of distributions (Section 3.2) that can be used to prove that distributions are related by equality, inequality or "closeness." A program logic (Section 3.3) is also provided to ease the development of proofs involving state or looping behavior. To reduce the effort required to develop a proof, the framework provides a library of tactics (Section 3.4) and a library of common program elements with

associated theory (Section 3.4). The equational theory, program logic, tactics, and programming library greatly simplify proof development, yet they are all derived from the semantics of the language, and using them to complete a proof does not reduce the trustworthiness of the proof.

By combining all of the components described above, a developer can produce a proof relating the probability that some adversary defeats the scheme to the probability that some other adversary is able to solve a problem that is assumed to be hard. This is a result in the *concrete setting*, in which probability values are given as expressions, and certain problems are assumed to be hard for particular constructed adversaries. In such a result, it may be necessary to inspect an expression describing a probability value to ensure it is sufficiently "small," or to inspect a procedure to ensure it is in the correct complexity class. FCF provides additional facilities to obtain more traditional asymptotic results, in which these procedures and expressions do not require inspection. A set of asymptotic definitions (Section 3.5) allows conclusions such as "this probability is negligible" or "this procedure executes a polynomial number of queries." In order to apply an assumption about a hard problem, it may be necessary to prove that some procedure is efficient in some sense. So FCF provides an extensible notion of efficiency (Section 3.5) and a characterization of non-uniform polynomial time Turing machines.[3]

3.1 Probabilistic Programs

We describe probabilistic programs using Gallina, the purely functional programming language of Coq, extended with a computational monad in the spirit of Ramsey and Pfeffer [20], that supports drawing uniformly random bit vectors. Listing 2 contains an example of a valid FCF program that implements a one-time pad on bit vectors. This program accepts a bit vector argument x, samples a random bit vector of length c (where c is a constant declared outside of this function) and assigns the result to variable p, then returns $p \oplus x$.

The syntax of the language is defined by an inductive type called `Comp` and is shown in Listing 1. At a high-level, `Comp` is an embedded domain-specific language that inherits the host language Gallina, and extends it with operations for generating and working with random bits.

The most notable primitive operation is `Rnd`, which produces n uniformly random bits. The `Repeat` operation repeats a computation until some decidable predicate holds on the value returned. This operation allows a restricted form of non-termination that is sometimes useful (*e.g.*, for sampling natural numbers in a specified range). The operations `Bind` and `Ret` are the standard monadic constructors, and allow the construction of sequences of computations, and computations from arbitrary Gallina terms and functions, respectively. However, note that the `Ret` constructor requires a proof of decidable equality for the underlying return type, which is necessary to provide a computational semantics as seen

[3] The current release of the FCF code for version 8.4 of Coq is available from `http://people.seas.harvard.edu/~apetcher/FCF_14.10.14.zip`

```
Inductive Comp : Set -> Type :=
| Ret : forall {A : Set}
  {H: EqDec A}, A -> Comp A
| Bind : forall {A B : Set}, Comp B
  -> (B -> Comp A) -> Comp A
| Rnd : forall n, Comp (Bvector n)
| Repeat : forall {A : Set}, Comp A
  -> (A -> bool) -> Comp A.
```

Listing 1. Probabilistic Computation Syntax

```
Definition OTP (x : Bvector c) :
     Comp (Bvector c)
  := p <-$ {0, 1}^c; ret (p xor x)
```

Listing 2. An Example of a Probabilistic Program

$$
\begin{aligned}
[\![\texttt{ret}\ a]\!] &= \mathbf{1}_{\{a\}} \\
[\![x \xleftarrow{\$} c; f\ x]\!] &= \lambda x. \sum_{b \in \mathrm{supp}([\![c]\!])} ([\![f\ b]\!]\ x) * ([\![c]\!]\ b) \\
[\![\{0,1\}^n]\!] &= \lambda x.\ 2^{-n} \\
[\![\texttt{Repeat}\ c\ P]\!] &= \lambda x. (\mathbf{1}_P\ x) * ([\![c]\!]\ x) * \left(\sum_{b \in P} ([\![c]\!]\ b)\right)^{-1}
\end{aligned}
$$

Fig. 1. Semantics of Probabilistic Computations

later in this section. In the remainder of this paper, we will use a more natural notation for these constructors: $\{0,1\}^n$ is equivalent to `(Rnd n)`, $x \xleftarrow{\$} c; f$ is the same as `(Bind c (fun x ⇒ f)`, and `ret e` is `(Ret _ e)`. The framework includes an ASCII form of this notation used in Listing 2. In the case of `Ret`, the notation serves to hide the proof of decidable equality, which is irrelevant to the programmer and is usually constructed automatically by proof search.

FCF uses a (mostly) *shallow embedding*, in which functions in the object language are realized using functions in the metalanguage. In contrast, CertiCrypt uses a *deep embedding*, in which the data type describing the object language includes constructs for specifying and calling functions, as well as all of the primitives such as bit-vectors and `xor`.

We have found that there are key benefits to shallow embedding. The primary benefit is that we immediately gain all of the capability of the metalanguage, including (in the case of Coq) dependent types, higher-order functions, modules, *etc.* Another benefit is that it is very simple to include any necessary theory in a security proof, and all of the theory that has been developed in the proof assistant can be directly utilized. One benefit that is specific to Coq (and other proof assistants with this property) is that Gallina functions are necessarily terminating, and Coq provides some fairly complex mechanisms for proving that a function terminates. By combining this restriction on functions with additional restrictions on `Repeat`, we can ensure that a computation (eventually) terminates, and that this computation corresponds with a distribution in which the total probability mass is 1.

On the other hand, the shallow embedding approach does have some drawbacks. The main drawback is that a Gallina function is opaque; we can only reason about a Gallina function based on its input/output behavior. The most significant effect of this limitation is that we cannot directly reason about the computational complexity of a Gallina function. We address this issue in Section 3.5.

The denotational semantics of a probabilistic computation is shown in Figure 1. The denotation of a term of type `Comp A` is a function in $A \to \mathbb{Q}$ which

should be interpreted as the probability mass function of a distribution on `A`. In FCF, all distributions are discrete and have finite support. In Figure 1, $\mathbf{1}_S$ is the indicator function for set S. So the denotation of `ret a` is a function that returns 1 when the argument is definitionally equal to a, and 0 otherwise. We can view the denotation of $x \xleftarrow{\$} c; f\ c$ as a marginal probability of the joint distribution formed by c and f. We know the probability of all events in c, but we only know the probability of events in f conditioned on events in c, so we can compute the probability of any event in this marginal distribution using the law of total probability. The fact that random bits are uniform and independent is encoded in the denotation of $\{0,1\}^n$, which is a function that ignores the argument and returns the probability that any n-bit value is equal to a randomly chosen n-bit value. The probability that `Repeat` $c\,P$ produces x is the conditional probability of x given P in c—which is equivalent to the function shown in Figure 1.

It is important to note that this language is purely functional, but the monadic style gives programs an imperative appearance. This appearance supports the *Familiarity* design goal since cryptographic definitions and games are typically written in an imperative style.

It is sometimes necessary to include some state in a cryptographic definition or proof. This can be easily accomplished by layering a state monad on top of `Comp`. However, this simple approach does not allow the development of definitions in which an adversary has access to an oracle that must maintain some hidden state across multiple interactions with the adversary. The definition could not simply pass the state to the adversary, because then the adversary could inspect or modify it. So FCF provides an extension to `Comp` for probabilistic procedures with access to a stateful oracle. The syntax of this extended language (Listing 3) is defined in another inductive type called `OracleComp`, where `OracleComp A B C` is a procedure that returns a value of type `C`, and has access to an oracle that takes a value of type `A` and returns a value of type `B`.

```
Inductive OracleComp :  Set -> Set -> Set -> Type :=
| OC_Query : forall (A B : Set), A -> OracleComp A B B
| OC_Run : forall (A B C A' B' S : Set), EqDec S -> EqDec B -> EqDec A ->
    OracleComp A B C -> S -> (S -> A -> OracleComp A' B' (B * S)) ->
    OracleComp A' B' (C * S)
| OC_Ret : forall A B C, Comp C -> OracleComp A B C
| OC_Bind : forall A B C C', OracleComp A B C ->
    (C -> OracleComp A B C') -> OracleComp A B C'.
```

Listing 3. Computation with Oracle Access Syntax

The `OC_Query` constructor is used to query the oracle, and `OC_Run` is used to run some program under a different oracle that is allowed to access the current oracle. The `OC_Bind` and `OC_Ret` constructors are used for sequencing and for promoting terms into the language, as usual. In the rest of this paper, we overload the sequencing and `ret` notation in order to use them for `OracleComp` as well as `Comp`. We use `query` and `run`, omitting the additional types and decidable equality proofs, as notation for the corresponding constructors of `OracleComp`.

$$
\begin{aligned}
[\![\texttt{query}\ a]\!] &= \lambda o\ s.(o\ s\ a) \\
[\![\texttt{run}\ c'\ s'\ o']\!] &= \lambda o\ s.[\![c'(\lambda x\ y.[\![(o'(fst\ x)\ y)\ o\ (snd\ x)]\!])\ (s', s)]\!] \\
[\![\texttt{ret}\ c]\!] &= \lambda o\ s.x \xleftarrow{\$} c; \texttt{ret}\ (x, s) \\
[\![x \xleftarrow{\$} c; f\ x]\!] &= \lambda o\ s.[x, s'] \xleftarrow{\$} [\![c\ o\ s]\!]; [\![(f\ x)\ o\ s']\!]
\end{aligned}
$$

Fig. 2. Semantics of Computations with Oracle Access

The denotation of an `OracleComp` is a function from an oracle and an oracle state to a `Comp` that returns a pair containing the value provided by the `OracleComp` and the final state of the oracle. The type of an oracle that takes an `A` and returns a `B` is `(S -> A -> Comp(B * S))` for some type `S` which holds the state of the oracle. The denotational semantics is shown in Figure 2.

3.2 (In)Equational Theory of Distributions

A common goal in a security proof is to compare two distributions with respect to some particular value (or pair of values) in the distributions. To assist with such goals, we have provided an (in)equational theory for distributions. This theory contains facts that can be used to show that two probability values are equal, that one is less than another, or that the distance between them is bounded by some value. For simplicity of notation, equality is overloaded in the statements below in order to apply to both numeric values and distributions. When we say that two distributions (represented by probability mass functions) are equal, as in $D_1 = D_2$, we mean that the functions are extensionally equal, that is $\forall x, (D_1\ x) = (D_2\ x)$.

Theorem 1 (Monad Laws).

$$
[\![a \xleftarrow{\$} \texttt{ret}\ b; f a]\!] = [\![(f\ b)]\!] \qquad [\![a \xleftarrow{\$} c; \texttt{ret}\ a]\!] = [\![c]\!]
$$
$$
[\![a \xleftarrow{\$} (b \xleftarrow{\$} c_1; c_2\ b); c_3\ a]\!] = [\![b \xleftarrow{\$} c_1; a \xleftarrow{\$} c_2\ b; c_3\ a]\!]
$$

Theorem 2 (Commutativity).

$$
[\![a \xleftarrow{\$} c_1; b \xleftarrow{\$} c_2; c_3\ a\ b]\!] = [\![b \xleftarrow{\$} c_2; a \xleftarrow{\$} c_1; c_3\ a\ b]\!]
$$

Theorem 3 (Distribution Irrelevance). *For well-formed computation c,*

$$
(\forall x \in supp([\![c]\!]), [\![f\ x]\!] y = v) \Rightarrow [\![a \xleftarrow{\$} c; f\ a]\!] y = v
$$

Theorem 4 (Distribution Isomorphism). *For any bijection f,*

$$
\begin{aligned}
&\forall x \in supp([\![c_2]\!]), [\![c_1]\!](f\ x) = [\![c_2]\!] x \\
\wedge\ &\forall x \in supp([\![c_2]\!]), [\![f_1\ (f\ x)]\!]\ v_1 = [\![f_2\ x]\!] v_2 \\
\Rightarrow\ &[\![a \xleftarrow{\$} c_1; f_1\ a]\!]\ v_1 = [\![a \xleftarrow{\$} c_2; f_2\ a]\!]\ v_2
\end{aligned}
$$

Theorem 5 (Identical Until Bad).

$$[\![a \xleftarrow{\$} c_1; \texttt{ret}\ (B\ a)]\!] = [\![a \xleftarrow{\$} c_2; \texttt{ret}\ (B\ a)]\!] \wedge$$

$$[\![a \xleftarrow{\$} c_1; \texttt{ret}\ (P\ a, B\ a)]\!](x, \mathit{false}) = [\![a \xleftarrow{\$} c_2; \texttt{ret}\ (P\ a, B\ a)]\!](x, \mathit{false}) \Rightarrow$$

$$|\ [\![a \xleftarrow{\$} c_1; \texttt{ret}\ (P\ a)]\!]\ x - [\![a \xleftarrow{\$} c_2; \texttt{ret}\ (P\ a)]\!]\ x\ | \leq [\![a \xleftarrow{\$} c_1; \texttt{ret}\ (B\ a)]\!]\ \mathit{true}$$

The meaning and utility of many of the above theorems is direct (such as the standard monad properties in Theorem 1), but others require some explanation. Theorem 3 considers a situation in which the probability of some event y in $[\![f\ x]\!]$ is the same for all x produced by computation c. Then the distribution $[\![c]\!]$ is irrelevant, and it can be ignored. This theorem only applies to *well-formed* computations: A well-formed computation is one that terminates with probability 1, and therefore corresponds to a valid probability distribution.

Theorem 4 is a powerful theorem that corresponds to the common informal argument that two random variables "have the same distribution." More formally, assume distributions $[\![c_1]\!]$ and $[\![c_2]\!]$ assign equal probability to any pair of events $(f\ x)$ and x for some bijection f. Then a pair of sequences beginning with c_1 and c_2 are denotationally equivalent as long as the second computations in the sequences are equivalent when conditioned on $(f\ x)$ and x. A special case of this theorem is when f is the identity function, which allows us to simply "skip" over two semantically equivalent computations at the beginning of a sequence.

Theorem 5, also known as the "Fundamental Lemma" from [7], is typically used to bound the distance between two games by the probability of some unlikely event. Computations c_1 and c_2 produce both a value of interest and an indication of whether some "bad" event happened. We use (decidable) predicate B to extract whether the bad event occurred, and projection P to extract the value of interest. If the probability of the "bad" event occurring in c_1 and c_2 is the same, and if the distribution of the value of interest is the same in c_1 and c_2 when the bad event does not happen, then the distance between the probability of the value of interest in c_1 and and c_2 is at most the probability of the "bad" event occurring.

3.3 Program Logic

The final goal of a cryptographic proof is always some relation on probability distributions, and in some cases it is possible to complete the proof entirely within the equational theory described in 3.2. However, when the proof requires reasoning about loops or state, a more expressive theory may be needed in order to discharge some intermediate goals. For this reason, FCF includes a program logic that can be used to reason about changes to program state as the program executes. Importantly, the program logic is related to the theory of probability distributions through completeness and soundness theorems which allow the developer to derive facts about distributions from program logic facts, and vice-versa.

The core logic is a Probabilistic Relational Postcondition Logic (PRPL), that behaves like a Hoare logic, except there are no preconditions. The definition of a PRPL specification is given in Definition 1. In less formal terms, we say that computations p and q are related by the predicate Φ if both p and q are marginals of the same joint probability distribution, and Φ holds on all values in the support of that joint distribution.

Definition 1 (PRPL Specification). *Given p : `Comp A` and q : `Comp B`,*

$$p \sim q\{\Phi\} \Leftrightarrow \begin{pmatrix} \exists\,(d : \texttt{Comp (A * B)}), \forall (x,y) \in supp([\![d]\!]), \Phi\, x\, y \wedge \\ [\![p]\!] = [\![x \xleftarrow{\$} d; \texttt{ret}\ (fst\, x)]\!] \wedge [\![q]\!] = [\![x \xleftarrow{\$} d; \texttt{ret}\ (snd\, x)]\!] \end{pmatrix}$$

Using the PRPL, we can construct a Probabilistic Relational Hoare Logic (PRHL) which includes a notion of precondition for functions that return computations as shown in Definition 2. The resulting program logic is very similar to the Probabilistic Relational Hoare Logic of EasyCrypt [5], and it has many of the same properties.

Definition 2 (PRHL Specification). *Given p : `A -> Comp B` and q : `C -> Comp D`,* $\{\Psi\}p \sim q\{\Phi\} \Leftrightarrow \forall a\, b, \Psi\, a\, b \Rightarrow (p\, a) \sim (q\, b)\{\Phi\}$.

Several theorems are provided along with the program logic definitions to simplify reasoning about programs. In order to use the program logic, one only needs to apply the appropriate theorem, so it is not necessary to produce the joint distribution described in the definition of a PRPL specification unless a suitable theorem is not provided. Theorems are provided for reasoning about the basic programming language constructs, interactions between programs and oracles, specifications describing equivalence, and the relationship between the program logic and the theory of probability distributions. Some of the more interesting program logic theorems are described below.

Theorem 6 (Soundness/Completeness).

$$p \sim q\{\lambda\, a\, b.a = x \Leftrightarrow b = y\} \Leftrightarrow [\![p]\!]\, x = [\![q]\!]\, y$$
$$p \sim q\{\lambda\, a\, b.a = x \Rightarrow b = y\} \Leftrightarrow [\![p]\!]\, x \leq [\![q]\!]\, y$$

Theorem 7 (Sequence Rule).

$$p \sim q\{\Phi'\} \Rightarrow \{\Phi'\}r \sim s\{\Phi\} \Rightarrow (x \xleftarrow{\$} p; r\, x) \sim (x \xleftarrow{\$} q; s\, x)\{\Phi\}$$

Theorem 8 (Oracle Equivalence). *Given an `OracleComp` c, and a pair of oracles, o and p with initial states s and t,*

$$\Phi = \lambda\, x\, y.(fst\, x) = (fst\, y) \wedge P\, (snd\, x)(snd\, y) \Rightarrow$$
$$(\forall a\, s'\, t', P\, s'\, t' \Rightarrow (o\, s'\, a) \sim (p\, t'\, a)\{\Phi\}) \Rightarrow P\, s\, t \Rightarrow ([\![c]\!]\, o\, s) \sim ([\![c]\!]\, p\, t)\{\Phi\}$$

Theorem 6 relates judgments in the program logic to relations on probability distributions. Theorem 7 is the relational form of the standard Hoare logic

sequence rule, and it supports the decomposition of program logic judgments. Theorem 8 allows the developer to replace some oracle with an observationally equivalent oracle. There is also a more general form of this theorem (omitted for brevity) in which the state of the oracle is allowed to go bad. This more general theorem can be combined with Theorem 5 to get "identical until bad" results for program/oracle interactions.

3.4 Tactics and Programming Library

The framework includes several tactics that can be used to transform goals using the facts in Sections 3.2 and 3.3. An example proof in section 4 uses the `comp_simp`, `inline_first` and `comp_skip` tactics. These tactics simplify programs (e.g. by applying left identity to remove unnecessary `ret` statements), pull out nested statements by applying associativity, and remove identical statements at the beginning of a pair of programs, respectively. Also included is a more sophisticated tactic called `dist_compute` that attempts to automatically discharge goals involving simple computations.

FCF also includes a library containing useful programming structures and their related theory. For example, the library includes several sampling routines, such as drawing a natural number from a specified range; drawing an element from a finite list, set, or group; or sampling from an arbitrary Bernoulli distribution. These sampling routines are all computations based on the `Rnd` statement provided by the language, and each routine is accompanied by a theory establishing that the resulting distribution is correct. The `CompFold` package contains *higher-order* functions for folding and mapping a computation over a list. This package uses the program logic extensively, and many of the theorems take a specification on a pair of computations as an argument, and produce a specification on the result of folding/mapping those computations over a list. The package also contains theorems about typical list and loop manipulations such as appending, flattening, fusion/fission and order permutation.

3.5 Asymptotic Theory and Efficient Procedures

Using the tools described in the previous sections, it is possible to complete a proof of security in the *concrete setting*. That is, the probability that an adversary wins a game is given as an expression which may include some value (or set of values) η that we can interpret as the security parameter. To get a typical asymptotic security result, we must show that this expression, when viewed as a function of η, is negligible. To assist with these sorts of conclusions, FCF provides a library of asymptotic definitions and theory.

An additional challenge is that the expression in the concrete security result may contain a value describing the probability that some other procedure wins some other game. We can apply a standard security assumption to conclude that this value is negligible in η, but in order to do so we need to show that the procedure is in the appropriate complexity class. FCF utilizes an extensible

notion of complexity, and it includes a simple predicate that accepts non-uniform worst-case polynomial time Turing machines. This predicate is constructed using a concrete cost model that assigns numeric costs to particular Coq functions, `Comp` values, and `OracleComp` values. The cost model for Gallina functions is necessarily axiomatic, since there is no way to directly reason about intensional properties of Coq functions. It includes axioms for some primitive operations as well as a set of combinators for determining the cost of more complicated functions. A proof must assume additional cost axioms for the set of functions used by constructed adversaries, which is relatively small in practice. The axioms need to be carefully inspected to ensure they accurately describe the desired complexity class, though a similar kind of inspection is needed to ensure the faithfullness of a cost model for a deeply-embedded language.

3.6 Code Extraction

FCF provides a code extraction mechanism that includes a strong guarantee of equivalence between a model of a probabilistic program and the code extracted from that model. We developed a small-step operational semantics that describes the behavior of these computations on a traditional machine (in which the memory contains values rather than probability distributions). This operational semantics (omitted for brevity) is an oracle machine that is given a finite list of bits representing the "random" input, and it describes how a computation takes a single step to produce a new computation, a final value, or fails due to insufficient input bits.

To show that this semantics is correct, we consider $[c]_n$, the multiset of results obtained by running a program c under this semantics on the set of all input lists of length n. We can view $[c]_n$ as a distribution, where the mass of some value a in the distribution is the proportion of input strings that cause the program to terminate with value a. The statement of equivalence between the semantics is shown in Theorem 9.

Theorem 9. *If c is well-formed, then* $\lim_{n\to\infty} [c]_n = [\![c]\!]$

FCF contains a proof of Theorem 9 as a validation of the operational semantics used for extraction. Now that we have an operational semantics, we can simply use the standard Coq extraction mechanism to extract it along with the model of interest and all supporting types and functions. Of course, the trustworthiness of the extracted code depends on the correctness of Coq's extraction mechanism. Gallina does not allow infinite recursion, so the framework includes OCaml code that runs a computation under the operational semantics until a value is obtained. The final step is instantiating any abstract types and functions with appropriate OCaml code. This extraction mechanism does not produce production-quality code, but the code could be used for purposes related to prototyping and testing.

This alternate semantics also provides other benefits. Because limits are unique, if two programs are equivalent under the operational semantics, then

they are also equivalent under the denotational semantics. This allows us to prove equivalence of two programs using the operational semantics when it is more convenient to do so. Another benefit is that the operational semantics can be considered to be the basic semantics for computations, and the denotational semantics no longer needs to be trusted. Some may prefer this arrangement, since the operational semantics more closely resembles a typical model of computation, and may be easier to understand and inspect. The operational semantics can also be used as a basis for a model of computation used to determine whether programs are efficient.

4 Security Proof Construction

This section uses an example to describe the process of constructing a proof of security using the general process described at the beginning of Section 3. We consider a simple encryption scheme constructed from a pseudorandom function (PRF), and we prove that ciphertexts produced by this scheme are indistinguishable under chosen plaintext attack (IND-CPA). This example proof is relatively simple, yet it contains many elements that one would find in a typical cryptographic argument, and so it allows us to exercise all of the key functionality of the framework. A more complex mechanized proof (*e.g.*, the proof of [11]) may have more intermediate games and a different set of arguments to justify game transformations, but the structure is similar to the proof that follows. The omitted details of the proof can be found in the longer form of this paper available at `http://arxiv.org/abs/1410.3735`.

4.1 Concrete Security Definitions

In FCF, security definitions are used to describe properties that some construction is proven to have, as well as problems that are assumed to be hard. In the PRF encryption proof, we use the definition of a PRF to assume that such a PRF exists, and we use that assumption to prove that the construction in question has the IND-CPA property. A concrete security definition typically contains some game and an expression that describes the *advantage* of some adversary – *i.e.*, the probability that the adversary will "win" the game.

The game used to define the concrete security of a PRF is shown in Listing 4. Less formally, we say that f is a PRF for some adversary A if A cannot effectively distinguish f from a random function. So this means that we expect that `PRF_Advantage` is "small" as long as `A` is an admissible adversary.

The function `f_oracle` simply puts the function `f` in the form of an oracle, though a very simple one with no state and with deterministic behavior. The procedure `RndR_func` is an oracle implementing a random function constructed using the provided computation `RndR`. The expressions involving `A` use a coercion in Coq to invoke the denotational semantics for `OracleComp`, and therefore ensure that `A` can query the oracle but has no access to the state of the oracle.

```
Variable Key D R : Set.
Variable RndKey : Comp Key.
Variable RndR : Comp R.
Variable A : OracleComp D R bool.
Variable f : Key -> D -> R.

Definition f_oracle(k: Key)(x: unit)
  (d : D) : Comp (R * unit) :=
  ret (f k d, tt).
Definition PRF_G_A : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt;
  ret b.
Definition PRF_G_B : Comp bool :=
  [b, _] <-$2 A (RndR_func) nil;
  ret b.
Definition PRF_Advantage :=
  | Pr[PRF_G_A] - Pr[PRF_G_B] |.
```

Listing 4. PRF Concrete Security Definition

```
Variable eta : nat.
Variable f : Bvector eta ->
  Bvector eta -> Bvector eta.

Definition PRFE_KeyGen :=
  {0, 1} ^ eta.
Definition PRFE_Encrypt
  (k : Key )(p : Plaintext) :=
  r <-$ {0, 1} ^ eta;
  ret (r, p xor (f k r)).
Definition PRFE_Decrypt
  (k : Key)(c : Ciphertext) :=
  (snd c) xor (f k (fst c)).
```

Listing 5. Encryption using a PRF

```
Variable Plaintext Ciphertext Key
    State : Set.
Variable KeyGen : Comp Key.
Variable Encrypt : Key -> Ciphertext
  -> Comp Plaintext.
Variable A1 : OracleComp
  Plaintext Ciphertext
  (Plaintext * Plaintext * State).
Variable A2 : State -> Ciphertext ->
  OracleComp Plaintext Ciphertext
      bool.

Definition EncryptOracle
  (k: Key)(x: unit)(p: Plaintext) :=
  c <-$ Encrypt k p;
  ret (c, tt).
Definition IND_CPA_SecretKey_G :=
  key <-$ KeyGen ;
  [b, _] <-$2
  (
    [p0, p1, s_A] <--$3 A1;
    b <--$$ {0, 1};
    pb <- if b then p1 else p0;
    c <--$$ Encrypt key pb;
    b' <--$ A2 s_A c;
    $ ret eqb b b'
  )
  (EncryptOracle key) tt;
  ret b.
Definition
  IND_CPA_SecretKey_Advantage :=
  | Pr[IND_CPA_SecretKey_G] - 1/2 |.
```

Listing 6. IND-CPA Concrete Security Definition

At a high level, this definition involves two games describing two different "worlds" in which the adversary may find himself. In one world (`PRF_G_A`) the adversary interacts with the PRF, and in the other (`PRF_G_B`) the adversary interacts with a random function. In each game, the adversary interacts with the oracle and then outputs a bit. The advantage of the adversary is the difference between the probability that he outputs 1 in world `PRF_G_A` and the probability that he outputs 1 in world `PRF_G_B`. If `f` is a PRF, then this advantage should be small.

The concrete security definition for IND-CPA encryption is shown in Listing 6. In this definition, `KeyGen` and `Encrypt` are the key generation and encryption procedures. The adversary comprises two procedures, `A1` and `A2` with different signatures, and the adversary is allowed to share arbitrary state information between these two procedures. This definition uses a slightly different style than the PRF definition—there is one game and the "world" is chosen at random within that game. Then the adversary attempts to determine which world was chosen.

In Listing 6, the game produces an encryption oracle from the `Encrypt` function and a randomly-generated encryption key. Then the remainder of the game, including the calls to `A1` and `A2`, may interact with that oracle. The code for this

definition includes some additional notation (different arrows and extra `$` symbols) that is only used to provide hints to the Coq parser and does not change the behavior of the program.

4.2 Construction

The construction, like the security definitions, can be modeled in a very natural way. Of course, one must take care to ensure that the construction has the correct signature as specified in the desired security property. The PRF encryption construction is shown in Listing 5.

In the PRF Encryption construction, we assume a `nat` called `eta` (η) which will serve as the security parameter. The encryption scheme is based on a function `f`, and the scheme will only be secure if `f` is a PRF. The type of keys and plaintexts is bit vectors of length `eta`, and the type of ciphertexts is pairs of these bit vectors. The decryption function is included for completeness, but it is not needed for this security proof.

4.3 Sequence of Games

The sequence of games represents the overall strategy for completing the proof. In the case of PRF Encryption, we want to show that the probability that the adversary will correctly guess the randomly chosen "world" is close to 1/2. We accomplish this by instantiating the IND-CPA security definition with the construction, and then transforming this game, little by little, until we have a game in which this probability is exactly 1/2. Each transformation may add some concrete value to the bounds, and we want to ensure that the sum of these values is small.

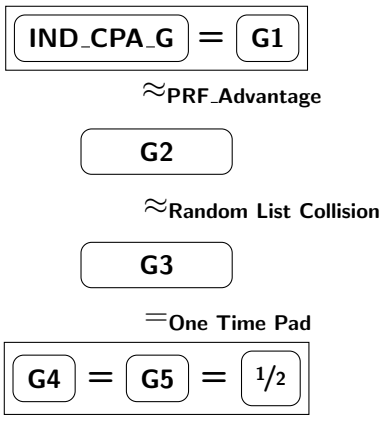

Fig. 3. Sequence of Games Diagram

```
Definition PRFE_Encrypt_OC (x : unit)
  (p : Plaintext) : OracleComp
  (Bvector eta) (Bvector eta)
  (Ciphertext * unit) :=
  r <--$$ {0,1} ^ eta;
  pad <--$ OC_Query r;
  $ (ret (r, p xor pad, tt)).

Definition PRF_A : OracleComp
  (Bvector eta) (Bvector eta) bool :=
  [a, n] <--$2 OC_Run A1
      PRFE_Encrypt_OC tt;
  [p0, p1, s_A] <-3 a;
  b <--$$ {0,1}; r <--$$ {0,1}^eta;
  pb <- if b then p1 else p0;
  pad <--$ OC_Query r;
  c <- (r, pb xor pad);
  z <--$ OC_Run (A2 s_A c)
      PRFE_Encrypt_OC n;
  [b',_] <-2 z; $ ret (eqb b b').
```

Listing 7. The Constructed Adversary Against the PRF

The diagram in Figure 3 shows the entire sequence of games, as well as the relationship between each pair of games in the sequence. In this diagram, two games are related by $=$ if they are identical, and by $\approx$ if they are close. When the equivalence is non-trivial, the diagram gives an argument for the equivalence, which implies a bound on the distance between the games when they are not equal. The intermediate game code is omitted for brevity, but a detailed description of each game transformation follows.

We begin by instantiating the IND-CPA definition with the construction and simplifying to produce game G1. This equivalence is obvious, and the proof can be completed using Coq's `reflexivity` tactic.

Next we replace the function `f` with a random function, and the distance between G1 and G2 is exactly the advantage of some adversary against a PRF. The adversary against the PRF (Listing 7) is constructed from `A1` and `A2`. `PRFE_Encrypt_OC` is an encryption oracle that interacts with the PRF as an oracle. `PRF_A` provides this encryption oracle to `A1` and `A2` (the two adversary procedures in the IND-CPA definition) using the `OC_Run` operation. This proof can be completed by performing simple manipulations and then unifying with `PRF_Advantage`.

Now we replace the random function output used to encrypt the challenge ciphertext with a bit vector selected completely at random to produce game G3. We show that G2 and G3 are "close" by demonstrating that these games are "identical until bad" in the sense of Theorem 5. The "bad" event of interest is the event that the randomly-generated PRF input used to encrypt the challenge plaintext is also used to encrypt some other value during the interaction between the adversary and the encryption oracle. There are two separate adversary procedures, and each one is capable of encountering `r` during its interaction with the oracle. To get an expression for the probability of the "bad"event, we assume natural numbers q_1 and q_2, and that `A1` performs at most q_1 queries and `A2` performs at most q_2 queries. FCF includes a library module called `RndInList` that includes general-purpose arguments related to the probability of encountering a randomly selected value in a list of a certain length, and the probability of encountering a certain value in a list of randomly-generated elements of a certain length. Using these arguments, we conclude that the distance between G2 and G3 is $q_1/2^n + q_2/2^n$.

The previous equivalences are proven using the program logic described in Section 3.3. Once the random functions are removed, there are no more issues related to state, and the remainder of the proof can be completed by reasoning on the probability distributions using the theory from Section 3.2.

In G3, the encryption of the challenge plaintext is by one-time pad, so we can replace the resulting ciphertext with a randomly-chosen value to produce G4. FCF contains a generic one-time-pad argument that we can apply to show that G3 is equivalent to G4. This step is relatively simple so we include the full code of the proof (Listing 8) for illustration. The one-time pad argument expects the game to be in a particular form, so we develop another intermediate game (`G3_1`), and we start by proving that `G3` is equivalent to `G3_1`. These games

only differ by associativity, so a simple repeated proof script establishes their equivalence. The second proof in Listing 8 focuses on the appropriate context, and then applies the one-time pad argument for `xor`. The custom tactics used in this proof are described in Section 3.4.

```
Theorem G3_G3_1_equiv:
  Pr[G3] == Pr[G3_1].

  unfold G3, G3_1.
  repeat (comp_simp;
    inline_first;
    comp_skip).
Qed.

Theorem G3_1_G4_equiv:
  Pr[G3_1] == Pr[G4].

    unfold G3_1, G4.
    do 4 (comp_skip;
      comp_simp).
    apply xor_OTP_eq.
    reflexivity.
Qed.
```

Listing 8. Proof of Equivalence of G3 and G4

In G4, the challenge bit is independent of all other values in the game, so we can move the sampling of this bit to the end of the game to produce G5. The proof of equivalence is by repeated application of the commutativity theorem (Theorem 2).

Finally, we develop the proof that the adversary wins Game 5 with probability exactly $1/2$. This proof proceeds by discarding all of the statements in the game before the coin flip. Then what remains is a very simple game that flips a coin and compares the result to a fixed value. A provided tactic can automatically determine that the probability that this game returns *true* is $1/2$.

By combining the equivalences of each pair of intermediate games, we get the final concrete security result shown in Listing 9. It is important to note that the statement of this theorem does not reference any of the intermediate games. The sequence of games was only a tool that we used to get the final result, and this sequence does not need to be inspected in order to trust the result.

```
Theorem PRFE_IND_CPA_concrete :
  IND_CPA_SecretKey_Advantage PRFE_KeyGen PRFE_Encrypt A1 A2 <=
  PRF_Advantage ({0,1}^eta) ({0,1}^eta) f PRF_A + (q1 / 2^eta + q2 / 2^eta).
```

Listing 9. Concrete Security Result

This completes the proof of security in the concrete setting. We have also developed an asymptotic security proof based on this result, but a discussion of this proof is omitted for brevity.

5 Comparison to EasyCrypt

This section attempts to evaluate FCF against the design goals listed in Section 2, and to contrast with both CertiCrypt and EasyCrypt.

All three of these frameworks provide concrete bounds, so this criterion is not discussed further. And, all three frameworks use a relatively familiar syntax for security definitions and constructions. We believe that, based on our experience working with cryptographers, they can easily understand these definitions (*e.g.*, Listing 4) after spending a few minutes familiarizing themselves with the notation.

Regarding proof automation, FCF lies somewhere between CertiCrypt and EasyCrypt. EasyCrypt achieves a significant level of automation by using SMT

solvers to discharge simple logical goals, but higher-level goals still need to be addressed manually by applying tactics. FCF achieves a similarly high level of automation through the use of existing and custom Coq tactics. These tactics are not as powerful as modern SMT solvers, so the developer may need to manually address some goals in FCF that would be discharged automatically in EasyCrypt. However, the semantics of programs in FCF is computational, so Coq is able to immediately compute an expression describing the probability distribution for any program. This allows some simple equivalences to be discharged immediately using the semantics and tactics provided by FCF.

Regarding trust in *extensional properties*, FCF and CertiCrypt are foundational, meaning that the program logic is constructed definitionally from the semantics. In contrast, the relationship between EasyCrypt's semantics and program logic is not mechanized. The trusted computing base of EasyCrypt includes the EasyCrypt front end (the OCaml code that implements EasyCrypt) and the Why3 verification condition generator and one or more SMT solvers (if the proof includes a tactic that invokes the SMT solvers), whereas the TCB of FCF and CertiCrypt includes only the Coq type checker. EasyCrypt provides no support for reasoning about *intensional properties* such as execution time, whereas CertiCrypt and FCF do, though FCF provides this suport using a trusted set of axioms.

EasyCrypt and CertiCrypt are based on simply-typed, first-order languages which are deeply-embedded into higher-order languages. This design makes it difficult to directly support abstraction, extension, and reuse, though these frameworks include elements which support these goals to some extent. In contrast, FCF uses a shallow embedding and the advanced features of Coq, such as dependent types, modules, notation, and higher-order functions, to support abstraction, extensiblity, and reuse. We believe that having such a rich language for describing games, assumptions, and arguments is critical for scaling to larger protocols.

FCF supports code generation with a semantics that is proven to be equivalent to the semantics used to reason about the probabilistic behavior of programs. That is, a program extracted from an FCF model is guaranteed to produce the correct probability distribution when the input bits provided to it are uniformly distributed, assuming the extraction mechanism of Coq preserves meaning. There has been some initial work [3] in producing implementations that correspond to EasyCrypt models, but there is no formal relationship between the semantics of the implementation and the semantics used to reason about the model.

EasyCrypt and FCF solve the same problem in slightly different ways and with different sets of strengths and weaknesses. It is too early to tell which sorts of proofs will benefit from one approach over the other.

6 Related Work

There has been a large amount of work in the area of verifying cryptographic schemes in recent years. In this section we will describe some of this related work, focusing on systems that attempt to establish security in the computational model. CertiCrypt [6] and EasyCrypt [5] have been thoroughly discussed previously in this paper.

There are several other examples of frameworks for cryptographic security proofs implemented within proof assistants. The most similar work is that of Nowak [19], who was the first to develop proofs of cryptography in Coq using a shallow embedding in which programs have probability distributions as their denotations. FCF builds on this work by adding more tools for modeling and reasoning such as procedures with oracle access (Section 3.1), a program logic (Section 3.3), and asymptotic reasoning (Section 3.5).

The work of [2] is a Coq library utilizing a deeply-embedded imperative programming language. This library is a predecessor to CertiCrypt, and it includes some important elements that were later adopted by CertiCrypt. Notably, the probabilistic programming language in this work is given a semantics in which program states are distributions, and the semantics describes how these distributions are transformed by each command in the language. Though this library lacks some of the features of CertiCrypt such as oracles, unrestricted loops, and the Probabilistic Relational Hoare Logic.

Verypto [9] is a fully-featured framework built on Isabelle [18] that includes a deep embedding of a functional programming language. To allow state information to remain hidden from adversaries, Verypto provides ML-style references, in contrast to the oracle system provided by FCF. To date, Verypto has only been used to prove the security of simple constructions, but this work uses an interesting approach that deserves more exploration.

CryptoVerif [10] is a tool based on a concurrent, probabilistic process calculus that is only able to prove properties related to secrecy and authenticity. CryptoVerif is highly automated to the extent that it will even attempt to locate intermediate games, and so proof development in CryptoVerif requires far less effort compared to FCF or EasyCrypt. However, there are a large number of proofs that could be completed in FCF or EasyCrypt that are impossible in CryptoVerif due to its specialized nature and lack of interactive proof development features.

Refinement types [8] have been used by Fournet et al [15] to develop proofs of security for cryptographic schemes in the computational model. In this system, a security property is specified as an ideal functionality (in the sense of the real/ideal paradigm), and constructions have these properties by assumption. This approach allows the proofs of security to be fairly simple, but many important facts are assumed rather than mechanized, and no concrete security claims are proved.

Computational soundness [1] provides another mechanism for verifying cryptographic schemes. This approach attempts to derive security in the computa-

tional model from security in the symbolic model by showing that any likely execution trace in the computational model also exists in the symbolic model. It is possible to mechanize such a proof as described in [4]. This approach is limited to classes of schemes for which computational soundness results have been discovered. Another limitation with this approach is that it can only produce proofs in the asymptotic setting—there is no way to prove concrete security claims.

Protocol Composition Logic (PCL) [12] provides a logic and proof system for verifying cryptographic schemes in the symbolic model. The system is based on a process calculus and allows reasoning about the results of individual protocol steps. More recent work [13] has extended this logic to allow for proofs in the computational model. In computational PCL, formulas are interpreted against probability distributions on traces and a formula is true if it holds with overwhelming probability. This approach is similar to computational soundness in that low-probability traces are ignored, and proofs of concrete security claims are impossible.

7 Conclusion and Future Work

Our contribution is a complete mechanized framework for specifying and checking cryptographic proofs within a proof assistant. Our framework compares favorably to the current state of the art, and provides many new benefits, such as extensibility through a foundational approach, a powerful language for describing schemes and arguments, and the ability to extract excutable code. Next we intend to demonstrate the scalability of FCF by describing a mechanized proof of security of a complex searchable symmetric encryption scheme ([11]).

References

1. Abadi, M., Rogaway, P.: Reconciling two views of cryptography (the computational soundness of formal encryption). In: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics. pp. 3–22. TCS '00, Springer-Verlag, London, UK (2000), http://dl.acm.org/citation.cfm?id=647318.723498
2. Affeldt, R., Tanaka, M., Marti, N.: Formal proof of provable security by game-playing in a proof assistant. In: Proceedings of the 1st International Conference on Provable Security. pp. 151–168. ProvSec'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1779394.1779408
3. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F.: Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. pp. 1217–1230. CCS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2508859.2516652
4. Backes, M., Unruh, D.: Computational soundness of symbolic zero-knowledge proofs against active attackers. In: 21st IEEE Computer Security Foundations Symposium, CSF 2008. pp. 255–269 (June 2008), preprint on IACR ePrint 2008/152

5. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: Advances in Cryptology – CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 71–90. Springer (2011)
6. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. pp. 90–101. ACM (2009), http://dx.doi.org/10.1145/1480881.1480894
7. Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331 (2004), http://eprint.iacr.org/
8. Bengtson, J., Bhargavan, K., Fournet, C., Maffeis, S., Gordon, A.D.: Refinement types for secure implementations. In: In 21st IEEE Computer Security Foundations Symposium (CSF08. pp. 17–32. IEEE (2008)
9. Berg, M.: Formal Verification of Cryptographic Security Proofs. Ph.D. thesis, Saarland University (2013), http://www.infsec.cs.uni-saarland.de/~berg/publications/thesis-berg.pdf
10. Blanchet, B.: Computationally sound mechanized proofs of correspondence assertions. In: 20th IEEE Computer Security Foundations Symposium (CSF'07). pp. 97–111. IEEE, Venice, Italy (Jul 2007)
11. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rou, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J. (eds.) Advances in Cryptology CRYPTO 2013, Lecture Notes in Computer Science, vol. 8042, pp. 353–373. Springer Berlin Heidelberg (2013)
12. Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol composition logic (PCL). Electronic Notes in Theoretical Computer Science 172, 311–358 (2007)
13. Datta, A., Derek, A., Mitchell, J.C., Shmatikov, V., Turuani, M.: Probabilistic polynomial-time semantics for a protocol security logic. In: Proceedings of the 32nd international conference on Automata, Languages and Programming. pp. 16–29. ICALP'05, Springer-Verlag, Berlin, Heidelberg (2005)
14. Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. Information Theory, IEEE Transactions on 31(4), 469–472 (1985)
15. Fournet, C., Kohlweiss, M., Strub, P.Y.: Modular code-based cryptographic verification. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security. pp. 341–350. ACM (2011)
16. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005), http://eprint.iacr.org/
17. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), http://coq.inria.fr, version 8.0
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Nowak, D.: A framework for game-based security proofs. Cryptology ePrint Archive, Report 2007/199 (2007), http://eprint.iacr.org/
20. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 154–165. POPL '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/503272.503288