The Foundational Cryptography Framework

Adam Petcher
Harvard University and MIT Lincoln Laboratory
apetcher@seas.harvard.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

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.

Categories and Subject Descriptors F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs; D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory

Keywords Cryptography, Proof Assistant, Mechanized Proof, 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 [6, 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 [18] 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 of the "tuple-set" construction of [10], which is a significant portion of a practical searchable symmetric encryption scheme. This is a complex and sophisticated construction, and the proof requires over 7000 lines of Coq code and includes a core argument involving more than 30 intermediate games.

FCF is heavily influenced by CertiCrypt [4], 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.

Following the release of EasyCrypt, a team of cryptographers and programming language experts (including one of the authors of this paper) attempted [17] to prove the security of a private information retrieval system [13]. This effort did not produce a complete proof because certain required facts could not be proven in EasyCrypt. Specifically, it was impossible to prove particular equivalences involving loop fusion and order permutation within a loop. In order to allow these equivalences in EasyCrypt, it would be necessary to prove them correct on paper and then modify the EasyCrypt code to include appropriate rules. EasyCrypt has seen significant improvements since its release, and it is possible that these sorts of equivalence are now supported, but a developer may encounter some other goal that EasyCrypt does not support.

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. In such a framework, the problem of the previous paragraph can be addressed by proving the appropriate theorem *within* the framework, and then by using that theorem to obatain the desired equivalence. 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 [20]). 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 trusted 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 [6]).

It can be cumbersome to work directly in the semantics, so we provide an equational theory (Section 3.2) of distributions 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 3.5). 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 3.6) allows conclusions like "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 3.7) and a characterization of non-uniform polynomial time Turing machines.[1]

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 [21], that supports drawing random bit vectors from an input tape. Listing 1 contains an example of a valid FCF program that implements a one-time pad using 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 $p \oplus x$.

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

Listing 1. An Example of a Probabilistic Program

The syntax of the language is defined by an inductive type called `Comp` and is shown in Listing 2. 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.

```
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 2. Probabilistic Computation Syntax

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

[1] The current release of the FCF code for version 8.4 of Coq is included as auxiliary material.

a proof of decidable equality for the underlying return type, which is necessary to provide a computational semantics as seen 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 as seen in Listing 1. 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 *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 bitvectors 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 3.7.

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

Figure 1. Semantics of Probabilistic Computations

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 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`.

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

Figure 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)**.**

$$[\![a \overset{\$}{\leftarrow} \texttt{ret}\ \ b; fa]\!] = [\![(f\ b)]\!]$$

$$[\![a \overset{\$}{\leftarrow} c; \texttt{ret}\ \ a]\!] = [\![c]\!]$$

$$[\![a \overset{\$}{\leftarrow} (b \overset{\$}{\leftarrow} c_1; c_2\ b); c_3\ a]\!] = [\![b \overset{\$}{\leftarrow} c_1; a \overset{\$}{\leftarrow} c_2\ b; c_3\ a]\!]$$

Theorem 2 (Commutativity)**.**

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

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

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

Theorem 4 (Distribution Isomorphism)**.** For any f which is a bijection from $supp([\![c_2]\!])$ to $supp([\![c_1]\!])$,

$$\forall x \in supp([\![c_2]\!]), [\![c_1]\!](f\ x) = [\![c_2]\!]x$$

$$\wedge \forall x \in supp([\![c_2]\!]), [\![f_1\ (f\ x)]\!]\ v_1 = [\![f_2\ x]\!]v_2$$

$$\Rightarrow [\![a \overset{\$}{\leftarrow} c_1; f_1\ a]\!]\ v_1 = [\![a \overset{\$}{\leftarrow} c_2; f_2\ a]\!]\ v_2$$

Theorem 5 (Identical Until Bad)**.**

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

$$[\![a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ \ (P\ a, B\ a)]\!](x, \text{false}) =$$

$$[\![a \overset{\$}{\leftarrow} c_2; \texttt{ret}\ \ (P\ a, B\ a)]\!](x, \text{false}) \Rightarrow$$

$$|\ [\![a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ \ (P\ a)]\!]\ x - [\![a \overset{\$}{\leftarrow} c_2; \texttt{ret}\ \ (P\ a)]\!]\ x\ | \leq$$

$$[\![a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ \ (B\ a)]\!]\ \text{true}$$

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 [6], 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`, $p \sim q\{\Phi\}$ iff,

$$\exists\,(d : \texttt{Comp (A * B)}), \forall (x, y) \in supp([\![d]\!]), \Phi\ x\ y\ \wedge$$

$$[\![p]\!] = [\![x \overset{\$}{\leftarrow} d; \texttt{ret}\ \ (fst\ x)]\!] \wedge [\![q]\!] = [\![x \overset{\$}{\leftarrow} d; \texttt{ret}\ \ (snd\ x)]\!]$$

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`, $\{\Psi\}p \sim q\{\Phi\}$ iff, $\forall a\ b, \Psi\ a\ b \Rightarrow (p\ a) \sim (q\ b)\{\Phi\}$.

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 w.r.t. Equality)**.**

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

Theorem 7 (Soundness/Completeness w.r.t. Inequality)**.**

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

Theorem 8 (Sequence Rule)**.**

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

Theorem 9 (Oracle Equivalence)**.** Given an *OracleComp* c, and a pair of oracles, o and p with initial states s and t,

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

Theorems 6 and 7 relate judgments in the program logic to relations on probability distributions. The forward direction (soundness) is typically used in a proof to transform the goal into the program logic in order to accurately reason about loops and/or state. Once the goal is in the program logic, the backward direction (completeness) can be used to return to a goal about distributions, or to apply an existing theorem that describes relations on probability distributions. Theorem 8 is the relational form of the standard Hoare logic sequence rule, and it supports the decomposition of program logic judgments.

Theorem 9 allows the developer to replace some oracle with an observationally equivalent oracle. This theorem takes a relational invariant P on the states of the oracles, and requires the developer to prove that if P holds on the states of the oracles and they are given identical input, then P holds on the resulting states and the outputs are identical. As long as P holds on the initial state of the oracle, this theorem concludes that the values returned by the program interacting with the oracles are equal, and that P holds on the final state. 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, and the interaction only produces equivalent results if the state does not 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

The framework includes several tactics that can be used to transform goals using the facts in Sections 3.2 and 3.3. For example, the `comp_swap_l` tactic applies commutativity (Theorem 2) to swap two independent statements at the beginning of the program on the left (of the equality, inequality, or program logic specification) in the current goal. There are similar tactics for manipulating games based on Left Identity (`comp_ret`), Associativity (`comp_inline`), Distribution Irrelevance (`comp_irr`), and the special case of Distribution Isomorphism in which the bijection is the identity function (`comp_skip`). Many of these tactics can be applied to goals related to probability distributions as well as goals in the program logic.

A tactic called `dist_compute` is provided to automatically discharge goals involving simple computations for which the corresponding distribution obviously has some desired property—typically that the probability of some event equals some specific value. A common proof technique is to develop a program in which the probability value of a particular event is obvious, and then relate other programs to this one by equivalence proofs. Then `dist_compute` can be used to automatically compute the desired probability value for this program. The tactic works by producing an arithmetic expression from the computation(s) and then performing case splits in appropriate ways in order to get goals that can be solved automatically by existing Coq decision procedures (such as `intuition` and `omega`).

3.5 Programming Library

The framework 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 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 and has the desired properties.

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.

3.6 Asymptotic Theory

The bulk of the effort in a security proof will be spent obtaining some result in the concrete setting. From there, a little more effort is required to produce a proof of some asymptotic fact that one would typically encounter in cryptography literature. To enable such asymptotic definitions and proofs, FCF includes a library of standard asymptotic definitions such as Definitions 3 and 4. The library also includes theorems that can be used to prove that functions are polynomial or negligible based on their composition(*e.g.*, the sum of polynomials is polynomial, the quotient of polynomial and exponential is negligible).

Definition 3 (At Most Polynomial)**.** A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *at most polynomial* iff $\exists x\ c_1\ c_2, \forall n, f(n) \leq c_1 * n^x + c_2$

Definition 4 (Negligible Function)**.** A function $f : \mathbb{N} \rightarrow \mathbb{Q}$ is *negligible* iff $\forall c, \exists n, \forall x > n, f(x) < {}^{1}/{}_{x^c}$

3.7 Efficient Procedures

A typical asymptotic security property states that a family of cryptographic schemes has some desirable property for all efficient adversaries. So in order to prove and apply these properties, we require some notion of "efficient" (families of) procedures. The language of computations used in FCF does not imply any particular model of computation—it is just a mechanism to specify probability distributions in a computational manner. Any notion of "efficiency" must first fix a model of computation, and then a complexity class on that model. We want this notion of efficiency to be flexible and extensible, so we can support several different models of computation and complexity classes.

To accomplish this flexibility, we parameterize asymptotic security definitions by an "admissibility predicate" indicating the class of adversaries against which a problem is assumed to be hard, or a scheme is proven to be secure. In this setting, the adversary is a family of procedures indexed by a natural number which indicates the value of the security parameter. The admissibility predicate can describe the efficiency of the adversary as well as other properties such as well-formedness or the number of allowed oracle queries as a function of the security parameter.

FCF includes a simple cost model and an associated admissibility predicate describing non-uniform worst-case polynomial time Turing machines that perform a (worst case) polynomial number of oracle queries. This admissibility predicate is constructed using a concrete cost model that assigns numeric costs to particular Coq functions, `Comp` values, and `OracleComp` values. In this cost model, the cost of executing a function is in $\mathbb{N}$, indicating the worst-case (over all arguments) execution time. The cost of running a `Comp` is in $\mathbb{N}$, indicating the worst-case execution time over all outcomes. The cost of executing an `OracleComp` is in $\mathbb{N} \to \mathbb{N}$, and is a function from the cost of executing the oracle to the cost of executing the computation, including the cost of executing all oracle queries.

The cost model for Gallina functions is axiomatic, as there is no direct way to capture such an intensional property for these terms. Our cost model includes axioms for primitive operations as well as a set of combinators for building more complicated functions. For example, the model includes an axiom stating that the `xor` operation for bit vectors of length c has a cost of c. As other examples, the model includes axioms stating that the cost of f composed with g is the sum of the costs of f and g, and the cost of `if` e_1 `then` e_2 `else` e_3 is the cost of e_1 plus the maximum of the costs of e_2 and e_3. Obviously, our cost axioms are incomplete, but in practice, the number required is relatively small since it is only necessary to reason about the functions used by a constructed adversary in a proof. Of course, the axioms need to be carefully inspected to ensure they accurately describe the desired complexity class.[2] But of course, a similar kind of inspection is needed to ensure the faithfullness of a cost model for a deeply-embedded language.

3.8 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. The denotational semantics of probabilistic computations relates a computation to a probability distribution, but it does not contain sufficient information to allow us to reason about the behavior of such computations on a traditional computer. So we developed a small-step operational semantics that describes the behavior of these computations on a machine in which the memory contains values rather than probability distributions. The 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. In order to compare the operational semantics with the denotational semantics, we want to view the operational semantics as a relation between computations and distributions. So the distribution related to computation c by the operational semantics is $\lim_{n\to\infty}[c]_n$. The statement of equivalence between the semantics is shown in Theorem 10.

Theorem 10. If c is well-formed, then

$$\lim_{n\to\infty}[c]_n = [\![c]\!]$$

FCF contains a proof of Theorem 10 as a validation of the operational semantics used for extraction, but this theorem 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. For example, theorems related to unrolling `Repeat` statements are trivial to prove under the operational semantics.

Another benefit of the operational semantics and proof of equivalence is that this 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.

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.

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). These security definitions, and the formal description of the construction, are provided in later sub-sections.

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 [10]) 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.

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.

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,

[2] Furthermore, a proof that a Gallina term has a cost described by these axioms does not mean that the extracted OCaml code will have this complexity, but rather, there exists some (propositionally) equivalent term which has the described cost. Since we are only trying to show the existence of an effective procedure, this is sufficient for our purposes.

```
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

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 5. 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 5, 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 6.

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 ½. 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

```
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 5. IND-CPA 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 6. Encryption using a PRF

which this probability is exactly ½. Each transformation may add some concrete value to the bounds, and we want to ensure that the sum of these values is small.

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. A detailed description of each game transformation follows:

1. Instantiate the IND-CPA definition with the construction. Unfold definitions and simplify. (Listing 7)
2. Apply the PRF definition to replace the PRF with a random function. (Listing 8)
3. Replace the random function output used to encrypt the challenge ciphertext with a bit vector selected completely at random. With overwhelming probability, the adversary does not notice this change. (Listing 9)

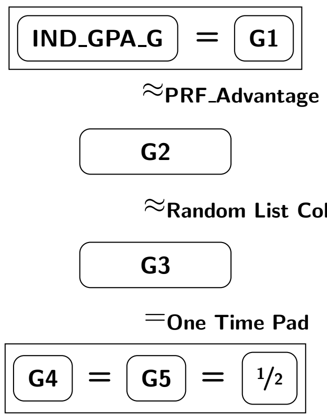

Figure 3. Sequence of Games Diagram

4. We have modified the game to the point that encryption of the challenge plaintext is by one-time pad. So we can replace the ciphertext with a randomly-chosen value. (Listing 10)
5. Now the ciphertext is independent from the plaintext, and thus independent from the random bit that was used to select the "world." This means we can move this coin flip to *after* the adversary guesses which world he is in. In this game, it is obvious that the probability that the adversary guesses the correct outcome of the coin flip is exactly one half. (Listing 11)

```
Definition G1 :=
  key <-$ PRFE_KeyGen;
  [b, _] <-$2
  (
    [p0, p1, s_A] <--$3 A1;
    b <--$$ {0, 1};
    pb <- if b then p1 else p0;
    c <--$$ PRFE_Encrypt key pb;
    b' <--$ (A2 s_A c);
    $ ret (eqb b b')
  )
  (PRFE_EncryptOracle key) tt;
  ret b.
```

Listing 7. Game 1

```
Definition PRFE_RandomFunc := @randomFunc
  (Bvector eta) (Bvector eta) ({0,1}^eta) _.

Definition RF_Encrypt s p :=
  r <-$ {0, 1} ^ eta;
  [pad, s] <-$2 PRFE_RandomFunc s r;
  ret (r, p xor pad, s).

Definition G2 :=
  [a, o] <-$2 A1 (RF_Encrypt) nil;
  [p0, p1, s_A] <-3 a;
  b <-$ {0, 1};
  pb <- if b then p1 else p0;
  [c, o] <-$2 RF_Encrypt o pb;
  [b', o] <-$2 (A2 s_A c) RF_Encrypt o;
  ret (eqb b b').
```

Listing 8. Game 2

4.4 Equivalence Proofs

The next step is to prove the appropriate sort of equivalence between each pair of games in the sequence. In the case of PRF Encryption, the goal is to show that the distance between the IND-CPA game and 1/2 is very small, and we accomplish this by showing that each pair of games in the sequence is either identical or "close."

```
Definition G3 :=
  [a, o] <-$2 A1 (RF_Encrypt) nil;
  [p0, p1, s_A] <-3 a;
  b <-$ {0, 1};
  pb <- if b then p1 else p0;
  r <-$ {0, 1}^eta;
  pad <-$ {0, 1}^eta;
  c <- (r, pb xor pad);
  [b', o] <-$2 (A2 s_A c) RF_Encrypt o;
  ret (eqb b b').
```

Listing 9. Game 3

```
Definition G4 :=
  [a, o] <-$2 A1 (RF_Encrypt) nil;
  [p0, p1, s_A] <-3 a;
  b <-$ {0, 1};
  pb <- if b then p1 else p0;
  r <-$ {0, 1}^eta;
  pad <-$ {0, 1}^eta;
  c <- (r, pad);
  [b', o] <-$2 (A2 s_A c) RF_Encrypt o;
  ret (eqb b b').
```

Listing 10. Game 4

```
Definition G5 :=
    [a, o] <-$2 A1 (RF_Encrypt) nil;
    [p0, p1, s_A] <-3 a;
    r <-$ {0, 1}^eta;
    pad <-$ {0, 1}^eta;
    c <- (r, pad);
    [b', o] <-$2 (A2 s_A c) RF_Encrypt o;
    b <-$ {0, 1};
    ret (eqb b b').
```

Listing 11. Game 5

The first step is to show that Game 1 (Listing 7) really is the IND-CPA game instantiated with this encryption scheme. This fact (Listing 12) is obvious, and the proof can be completed using Coq's `reflexivity` tactic. `A1` and `A2` are parameters representing the procedures of the adversary against the encryption scheme. In the statement of this theorem, `==` is equality for rational numbers. This equality is registered with Coq's setoid system to enable tactics such as `reflexivity` and rewriting.

```
Theorem G1_equiv :
  Pr[IND_CPA_SecretKey_G PRFE_KeyGen PRFE_Encrypt
       A1 A2] == Pr[G1].
```

Listing 12. Equivalence of the Security Definition and Game 1

Next we show that the distance between Games 1 and 2 is exactly the advantage of some adversary against a PRF. The adversary against the PRF (Listing 13) 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` using the `OC_Run` operation.

To prove the "closeness" of Games 1 and 2, first we prove that the interaction between `PRF_A` and the PRF oracle is equivalent to Game 1 (Theorem 14). Then we prove that the interaction between `PRF_A` and the random function oracle is equivalent to Game 2 (Theorem 15). Finally we apply the results of parts 1 and 2 and unify with the definition of a PRF (Theorem 16).

To prove Theorems 14 and 15, we mostly perform simple manipulations such as applying the denotational semantics of `OracleComp`, inlining, and removing identical statements at the beginning of the game. In both of these proofs, the adversary is

```
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 <--$$ {0, 1};
  pb <- if b then p1 else p0;
  r <--$$ {0, 1} ^ eta;
  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 13. The Constructed Adversary Against the PRF

```
Theorem G1_PRF_A_equiv :
  Pr[k <-$ {0, 1}ˆ eta;
    [b, _] <-$2 PRF_A (f_oracle k) tt;
    ret b] == Pr[G1].
```

Listing 14. Equivalence PRF_A and G1

```
Theorem G2_PRF_A_equiv :
  Pr[[b, _] <-$2 PRF_A PRFE_RandomFunc nil;
    ret b] == Pr[G2].
```

Listing 15. Equivalence PRF_A and G2

```
Theorem G1_G2_close : | Pr[G1] - Pr[G2] | ==
  PRF_Advantage ({0, 1}ˆeta) ({0, 1}ˆeta)
    f PRF_A.
```

Listing 16. Closeness of Game 1 and Game 2

interacting with two different, but observationally equivalent, oracles. So we use the program logic and Theorem 9 to prove that these interactions produce equivalent results.

Next we show that Games 2 and 3 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 (`r` in Game 2) 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. So we divide this proof into two parts, one for each adversary procedure, where each part includes an "identical until bad" argument. In the first step, we produce the `pad` value randomly (without using the random function), but then add an entry for `r` and `pad` to the state of the random function. In the second step, we produce `pad` randomly, and do not add an entry to the random function.

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. By combining the "identical until bad" proofs with these arguments to get expressions bounding the probabilities of the bad events, we obtain the result of Listing 17.

The next step is to use a one-time-pad argument to replace the challenge ciphertext with a randomly-chosen value. The library contains a generic one-time-pad argument that we can apply here.

```
Theorem G2_G3_close :
  | Pr[G2] - Pr[G3] | <=
    q1 / (2 ˆ eta) + q2 / (2 ˆ eta).
```

Listing 17. Closenes of Game 2 and Game 3

We transform this game into an equivalent game that unifies with the one-time-pad argument, then we apply the argument to get the result shown in Listing 18.

```
Theorem G3_G4_equiv : Pr[G3] == Pr[G4].
```

Listing 18. Equivalence of Game 3 and Game 4

Now that the ciphertext is independent of the challenge bit, we produce a new game by moving the sampling of the challenge bit to the end of the game. To prove this fact (Listing 19), we simply unfold the required definitions, skip over all of the identical pairs of statements at the beginning of the proof, then swap the order of independent statements in the game on the left in order to make these statements align with the identical statements in the game on the right.

```
Theorem G4_G5_equiv :
  Pr[G4] == Pr[G5].
    unfold G4, G5.
    do 3 (comp_skip; comp_simp; comp_swap_l).
    comp_skip; comp_simp.
    reflexivity.
  Qed.
```

Listing 19. Equivalence of Game 4 and Game 5

Finally, we develop the proof that the adversary wins Game 5 with probability exactly $1/2$. This proof (Listing 20) proceeds by discarding all of the initial statements in the game using the `comp_irr_l` tactic. Note that this tactic produces an obligation to prove that the statement being discarded is a well-formed computation, which can be discharged with the tactic `wftac`. Then what remains is a very simple game, and `dist_compute` can automatically compute the probability that this game returns *true*.

```
Theorem G5_one_half :
  Pr[G5] == 1/2.
    do 4 comp_irr_l; wftac.
    dist_compute.
Qed.
```

Listing 20. Probability of Winning Game 5

By combining the equivalences of each pair of intermediate games, we get the final concrete security result shown in Listing 21. 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.

The concrete security result in Listing 21 may be sufficient for many purposes. We have an expression describing the advantage of the adversary, and we can inspect this expression to see whether this advantage is sufficiently small. We also must inspect the definition of the adversary `PRF_A`, which appears in this result, and ensure that this adversary is "efficient" according to the desired complexity class. Next we will show how to derive an asymptotic security result based on this concrete result. A benefit of proving asymptotic security is that this proof removes the requirement to inspect the constructed adversary and the expression describing the adversary's advantage.

```
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 21. Concrete Security Result

4.5 Asymptotic Security Definitions

Now we give the asymptotic security definitions for PRFs and IND-CPA encryption. These definitions are parameterized by an admissibility predicate as described in Section 3.7. The IND-CPA definition accepts two admissibility predicates – one for each adversary procedure.

The asymptotic security definition for a PRF is given in Listing 22. In this definition, RndKey, RndR, and f are nat-indexed families of procedures. Similarly in the IND-CPA definition (Listing 23), KeyGen and Encrypt are nat-indexed families of procedures. Both of these definitions are claims over all admissible nat-indexed adversary families. Note that both definitions reuse the expressions provided in the concrete security definitions. This style provides a convenient method for developing an asymptotic security proof from a concrete security proof.

```
Variable D R Key : nat -> Set.
Variable RndKey : forall n, Comp (Key n).
Variable RndR : forall n, Comp (R n).
Variable f : forall n, Key n -> D n-> R n.

Definition PRF :=
  forall (A : \forall n, OracleComp (D n) (R n)
    bool), admissible_A A ->
    negligible (fun n => PRF_Advantage
      (RndKey n) (RndR n) (@f n) (A n)).
```

Listing 22. Definition of a PRF

```
Variable Plaintext Ciphertext Key State :
  nat -> Set.
Variable KeyGen : forall n, Comp (Key n).
Variable Encrypt : forall n, Key n ->
  Ciphertext n -> Comp (Plaintext n).

Definition IND_CPA_SecretKey :=
  forall (State : nat -> Set)
  (A1 : forall n, OracleComp (Plaintext n)
    (Ciphertext n)
    (Plaintext n * Plaintext n * State n))
  (A2 : forall n, State n -> Ciphertext n ->
      OracleComp (Plaintext n) (Ciphertext n)
      bool),
    admissible_A1 A1 ->
    admissible_A2 A2 ->
    negligible
      (fun n => IND_CPA_SecretKey_Advantage
      (KeyGen n) (@Encrypt n) (A1 n) (A2 n) ).
```

Listing 23. Definition of IND-CPA Encryption

4.6 Efficiency of Constructed Adversaries

The first step in proving an asymptotic security result is to view each constructed adversary in the concrete proof as a nat-indexed family of adversaries, and prove that this family is "efficient" as defined by some complexity class. In the PRF Encryption proof, we use the non-uniform polynomial time complexity class described in Section 3.7. Because this class includes a concrete cost model, we begin with a proof of the concrete cost of each constructed adversary procedure.

We begin by assuming costs for A1 and A2. A1_cost is a function describing the cost of A_1. A2_cost_1 is a number describing how much it costs for A2 to compute an OracleComp that is closed over a state and a ciphertext. Then A2_cost_2 is a function describing the cost of executing this OracleComp. Given these assumptions, we can give a cost to PRF_A as shown in Listing 24. In the statement of this theorem, oc_cost, comp_cost, and cost are the cost models for OracleComp, Comp, and Coq functions, respectively. Note that this cost model is overly conservative and some costs are counted multiple times.

```
Theorem PRF_A_cost :
  oc_cost cost (comp_cost cost) PRF_A
    (fun x => (A1_cost (x + (5 * eta))) +
    (A2_cost_2 (x + (5 * eta))) +
     x + 5 * A2_cost_1 + 6 + 7 * eta).
```

Listing 24. Cost of Constructed Procedure PRF_A

This proof is completed by repeatedly applying the rule of the cost model that is relevant to the term in the goal, which is a highly syntax-directed operation that can be mostly automated. Once all these syntax-directed rules are applied, the developer is obligated to prove that the expression obtained in this process is equal to (or less than) the expression in the statement of the theorem. In this last step of the proof, automated tactics such as omega are very useful.

4.7 Asymptotic Security Proof

The final step in the proof is to show that the security definition shown in Listing 23 holds on this construction as long as f is a PRF as defined in Listing 22. The statement of this fact is shown in Listing 25. Note that admissible_oc and admissible_oc_func_2 are the admissibility predicates for OracleComp and for functions with two arguments that produce an OracleComp defined in the simple complexity class described in Section 3.7.

```
Theorem PRFE_IND_CPA :
  PRF Rnd Rnd f (admissible_oc cost) ->
  IND_CPA_SecretKey
    PRFE_KeyGen (fun n => PRFE_Encrypt (@f n))
    (admissible_oc cost)
    (admissible_oc_func_2 cost).
```

Listing 25. Asymptotic Security of PRF Encryption

The primary obligation of this proof is to show that the function defining the advantage of any admissible family of adversaries against this encryption scheme is a negligible function. The fact that this adversary family is admissible allows us to use the result of Listing 24, along with other facts, to conclude that the constructed adversary family against the PRF is admissible. In the course of this proof, we must show that the expression implied by Figure 24 is at most polynomial in η if x is at most polynomial in η and all the costs related to PRF_A1 and PRF_A2 are at most polynomial in η. This fact is proven using the provided theory of polynomial functions (Section 3.6).

From the admissibility of the constructed adversary, and from the fact the f is a PRF against all admissible adversaries, we can conclude that the constructed adversary's advantage against the PRF is negligible. The advantage of this adversary against the PRF is one of the terms that appears in the bounds of the concrete result (Listing 21). The other term is $q_1/2^\eta + q_2/2^\eta$, where q_1 and q_2 are the number of oracle queries performed by the two adversary procedures. The admissibility predicates ensure that each adversary only performs a polynomial number of queries, so q_1 and q_2 must be polynomial in η, and this expression is negligible in η. So the advantage of the adversary against this encryption scheme is the sum of two negligible functions, and is therefore negligible.

The entire proof of security for this encryption scheme requries approximately 1500 lines of Coq code, of which about 700 lines are specification (including 100 lines of cryptographic definitions and intermediate games) and 800 lines are proof. The proof incorporates another 500 lines of code for the reusable arguments (*e.g.*, the one-time pad argument). We expect that a skilled Coq developer could complete such a proof in a matter of days (though he may require the help of a cryptographer to develop the sequence of games and high-level arguments).

5. Evaluation

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 this computation and FCF's `dist_compute` tactic.

Regarding trust in *extensional properties*, FCF and CertiCrypt are foundational, meaning that the program logic is constructed definitionally from the semantics. In contrast, EasyCrypt relies upon a set of axioms for its program logic. EasyCrypt also relies on the correctness of the EasyCrypt front end and the Why3 verification generator, whereas FCF and CertiCrypt only depend on the Coq type checker. EasyCrypt provides no support for reasoning about *intensional properties* like 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. 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 and assumptions 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 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.

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 [4] 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 [20], 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 3.6).

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. CertiCrypt and EasyCrypt extended this work by adding language constructs such as oracles and unrestricted loops, and well as reasoning tools such as the Probabilistic Relational Hoare Logic.

Verypto [8] is a fully-featured framework built on Isabelle [19] 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 [9] 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.

Refinement types [7] 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 proofs are completed using the "sequence of games" style in the asymptotic setting. This approach allows the proofs of security to be fairly simple, but no concrete security claims are proved, so it may be difficult to make practical claims based on such a proof.

Computational soundness [1] provides another mechanism for verifying cryptographic schemes. This approach attempts to derive security in the computational 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 [3]. 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 [11] 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 the ability to extract excutable code. Of course, whether these benefits apply at scale is still an open question, and thus a key direction for us is to prove security for an even wider range of standard constructions, as well as novel cryptographic schemes. In particular, we have proven the security of the "tuple set" construction of [10], and we intend to continue developing this work into a proof of a searchable symmetric encryption scheme.

The biggest limitation of FCF is that it currently lacks definitions for many cost models and complexity classes that are commonly used in cryptography. We hope to develop more cost models and complexity classes, including a complexity class describing (uniform) probabilistic polynomial time Turing machines.

References

[1] M. Abadi and P. Rogaway. 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*, TCS '00, pages 3–22, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67823-9. URL `http://dl.acm.org/citation.cfm?id=647318.723498`.

[2] R. Affeldt, M. Tanaka, and N. Marti. Formal proof of provable security by game-playing in a proof assistant. In *Proceedings of the 1st International Conference on Provable Security*, ProvSec'07, pages 151–168, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75669-8, 978-3-540-75669-9. URL `http://dl.acm.org/citation.cfm?id=1779394.1779408`.

[3] M. Backes and D. Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *21st IEEE Computer Security Foundations Symposium, CSF 2008*, pages 255–269, June 2008. Preprint on IACR ePrint 2008/152.

[4] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009. URL `http://dx.doi.org/10.1145/1480881.1480894`.

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

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

[7] J. Bengtson, K. Bhargavan, C. Fournet, S. Maffeis, and A. D. Gordon. Refinement types for secure implementations. In *In 21st IEEE Computer Security Foundations Symposium (CSF08*, pages 17–32. IEEE, 2008.

[8] M. Berg. *Formal Verification of Cryptographic Security Proofs*. PhD thesis, Saarland University, 2013. URL `http://www.infsec.cs.uni-saarland.de/~berg/publications/thesis-berg.pdf`.

[9] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111, Venice, Italy, July 2007. IEEE.

[10] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rou, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti and J. Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40040-7. .

[11] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of the 32nd international conference on Automata, Languages and Programming*, ICALP'05, pages 16–29, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27580-0, 978-3-540-27580-0. .

[12] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172: 311–358, 2007.

[13] E. De Cristofaro, S. Jarecki, X. Liu, Y. Lu, and G. Tsudik. Privacy-protecting information retrieval, University of Irvine team: Protocol and proofs. Appendix E of SPAR Program BAA: `https://www.fbo.gov/utils/view?id=32750071e5cf4afc3b7e973d` 2010.

[14] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Information Theory, IEEE Transactions on*, 31 (4):469–472, 1985. ISSN 0018-9448. .

[15] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 341–350. ACM, 2011. ISBN 978-1-4503-0948-6.

[16] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. `http://eprint.iacr.org/`.

[17] J. Herzog, C. Meadows, A. Jaggard, A. Stoughton, and J. Katz. MITLL-NRL panel: Easycrypt 0.2 feedback and opinions. `http://web.archive.org/web/20140703170052/https://easycryp` 2013. Accessed: 201407-03.

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

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[20] D. Nowak. A framework for game-based security proofs. Cryptology ePrint Archive, Report 2007/199, 2007. `http://eprint.iacr.org/`.

[21] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 154–165, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. . URL `http://doi.acm.org/10.1145/503272.503288`.