A Mechanized Proof of Security for Searchable Symmetric Encryption

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

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

***Abstract*—We present a mechanized proof of security for an efficient Searchable Symmetric Encryption (SSE) scheme completed in the Foundational Cryptography Framework (FCF). FCF is a Coq library for reasoning about cryptographic schemes in the computational model that features a small trusted computing base and an extensible design. Through this effort, we provide the first mechanized proof of security for an efficient SSE scheme, and we demonstrate that FCF is well-suited to reasoning about such complex protocols.**

I. Introduction

Complex cryptographic systems are increasingly prevalent. Within the last decade, existing technologies such as TLS have become more important due to increased use, and a number of new cryptographic schemes have emerged to support online anonymity, secure financial transactions, anonymous currencies, and outsourced storage and computation. The security of such a system is traditionally ensured by the development of a mathematical proof of security, or by widespread efforts to find weaknesses in the system. The latter approach is probably impractical for specialized systems, and the former approach suffers from the issue that many of these proofs are not carefully verified [20, 10].

The use of formal methods is a promising solution to this problem, and several systems have been developed to formally verify proofs of security for cryptographic schemes. FCF [23] is a library for the general-purpose Coq proof assistant that can be used to develop and verify such proofs of security in the computational model. This framework supports the "sequence of games" [10] style by design, and proofs provide concrete bounds on probability values. FCF provides a language for probabilistic programs, and a semantics that relates programs to discrete probability distributions. The framework also provides a rich 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.

This paper demonstrates the viability of using FCF to construct formal proofs of security for complex cryptographic schemes by proving the security of the efficient Searchable Symmetric Encryption (SSE) scheme described in [15]. In such a scheme, a client can store a large database on an untrusted server, and the server can query the database on behalf of the client without learning the contents of the database or the query. This scheme is accompanied by a proof of security on paper, but we can gain greater assurance of the security of this scheme by developing a mechanized proof of security in FCF. Note that the scheme we verified in this effort is exactly the Single-Keyword Search (SKS) scheme described in [15], and our formal proof was inspired by the proof presented in the paper.

Following the release of EasyCrypt [7], a team of cryptographers and programming language experts attempted [21] to prove the security of a Private Information Retrieval (PIR) system [17] which can be viewed as a predecessor to the SSE scheme of [15]. This effort did not produce a complete proof because certain required facts could not be proven in EasyCrypt. Specifically, it was impossible at the time to prove particular equivalences involving loop fusion and order permutation within a loop without modifying the EasyCrypt code to accept these equivalences.

EasyCrypt has seen significant improvement since its release, and a proof of security for a greatly simplified form of this PIR scheme [25] has been completed in EasyCrypt. In parallel, FCF was developed in order to find a more general solution to the problem of "missing" theory in cryptographic frameworks such as EasyCrypt. Due to the foundational nature of FCF, any required theorem can be formally derived from the semantics without increasing the trusted computing base. We rely on this trustworthy extensibility of FCF to develop the additional theory required to complete the proof described in this paper.

The novel contributions described in this paper are the following: (1) We developed the first mechanized proof of security for an efficient SSE scheme. This proof demonstrates that it is possible to increase the trustworthiness of outsourced searchable data storage through mechanized verification. (2) The proof described in this paper is among the most complex mechanized cryptographic proofs that has been completed to date, and we demonstrate that FCF is capable of scaling to such large proofs. Further we show how the higher-order abstraction available in Coq helps FCF manage the complexity of such proofs through decomposition. Table I (in Section VII) summarizes the complexity of this proof, which comprises several cryptographic reductions including over 14,000 lines of Coq code and 58 intermediate games. This development effort also produced a significant amount of FCF theory related

This work is sponsored by the Intelligence Advanced Research Projects Activity under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

to loop transformations, hybrid arguments, sampling without replacement, and constructions involving repeated independent trials. We added this theory to the standard library of FCF in order to assist with future proof development efforts. (3) This proof can serve as an example to help developers produce similar proofs in FCF, EasyCrypt, or other systems.[1]

II. The Foundational Cryptography Framework

This section provides a brief introduction to the Foundational Cryptography Framework. FCF is explained by example, and all of the examples in this section are elements of the SSE proof described later in the paper. A more detailed description of the theory of FCF is available in [23].

A. Probabilistic Programs

FCF provides a common probabilistic programming language for describing all cryptographic constructions, security definitions, and problems that are assumed to be hard. Probabilistic programs are described using Gallina, the purely functional programming language of Coq, extended with a computational monad that adds sampling uniformly random bit vectors. The type of probabilistic computations that return values of type A is Comp A. The code uses {0,1}^n to describe sampling a bit vector of length n. Arrows (e.g. <-$) denote sequencing (i.e. bind) in the monad. Other notation used in the listings will be described when its meaning is not apparent.

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

Listing 1. Example Program: One-Time Pad

Listing 1 contains an example program implementing a one-time pad on bit vectors of length c (for any natural number c). The program produces a random bit vector and stores it in p, then returns the *xor* (using the standard Coq function BVxor) of p and the argument x.

B. Semantics and Probability Theory

The language of FCF has a denotational semantics in the style of Nowak [22] that relates programs to discrete, finite probability distributions. A distribution on type A is modeled as a function in $A \rightarrow \mathbb{Q}$ which should be interpreted as a probability mass function. This semantics can be used to show that the probabilities of two events are equal, related by an inequality, or distant by at most some value. All of these claims are necessary in order to complete proofs in the "sequence of games" style, in which several games are provided, and relations on adjacent pairs of games are proven. The semantics can also be used to determine an exact value for the probability of an event, which is necessary to provide concrete bounds in security proofs.

FCF provides a theory of distributions that can be used to complete proofs without appealing directly to the semantics. FCF also provides a library of tactics that apply individual theorems, sequences of theorems, or perform non-trivial computations in order to discharge goals. The theory is all proven in Coq from the semantics, and the tactics only apply theorems, so these objects are not in the trusted computing base of FCF.

Using the theory and tactics, we can complete proofs as shown in Listing 2. In this proof, we show that a one-time pad applied to an arbitrary value has the same distribution as a random bit vector. In the statement of the theorem, D represents the denotational semantics, which is used to obtain the distribution corresponding to the program that follows it. Because these distributions are represented as functions, we compare them with respect to an arbitrary value in the distribution y. We also use the notation Pr[c] to represent the probability that Boolean computation c produces true. The == represents equality for rational numbers.

The proof proceeds by using tactics to transform the goal or hypotheses until we get a goal that is trivial and can be automatically discharged. We use intuition to introduce all variables, then we unfold the definition of OTP to replace D(OTP x) with the body defined in Listing 1. r_ident_r is an FCF tactic that uses Coq's *rewrite* tactic along with a monadic right identity theorem to replace D({0,1}^c) with D(a <-$ {0,1}^c; ret a). This transformation puts the goal into a form where we can apply the distribution isomorphism theorem (Theorem 1) to complete the proof. This theorem takes a bijection and its inverse, and we supply the involution BVxor c x for both. When this theorem is applied, several simpler goals are produced. These goals are either trivial equalities or simple facts about the BVxor function (e.g. commutativity, identity) which can be discharged by the specialized xorTac tactic.

```
Theorem OTP_eq_Rnd:
  forall (x y : Bvector c),
    D (OTP x) y == D ({0, 1}^c) y.

    intuition. unfold OTP.
    r_ident_r.
    eapply (dist_iso (BVxor c x) (BVxor c x));
      intuition; xorTac.
Qed.
```

Listing 2. Example Proof: Equivalence of One-Time Pad

Theorem 1 (Distribution Isomorphism)**.** For any bijection f in supp($\llbracket c_2 \rrbracket$) $\rightarrow$ supp($\llbracket c_1 \rrbracket$),

$$\begin{aligned} &\forall x \in \text{supp}(\llbracket c_2 \rrbracket), \llbracket c_1 \rrbracket (f\ x) = \llbracket c_2 \rrbracket x \\ &\wedge \forall x \in \text{supp}(\llbracket c_2 \rrbracket), \llbracket f_1\ (f\ x) \rrbracket\ v_1 = \llbracket f_2\ x \rrbracket v_2 \\ &\Rightarrow \llbracket a \overset{\$}{\leftarrow} c_1; f_1\ a \rrbracket\ v_1 = \llbracket a \overset{\$}{\leftarrow} c_2; f_2\ a \rrbracket\ v_2 \end{aligned}$$

Once we have proven the theorem in Listing 2 we can use this theorem to rewrite anything that unifies with either expression. We can also use other theorems and tactics to focus on some location in the program and perform this rewrite at that location. The ability to perform such rewrites provides the basis for completing proofs composed of sequences of games.

The language of FCF also includes a (Repeat c P) statement that repeats computation c until a decidable predicate P holds on the result. This is equivalent to conditioning the distribution corresponding to c on the event P.

A simple program that uses Repeat to sample uniformly-distributed natural numbers in $[0, n)$ is shown in Listing 3. RndNat_h is a helper function that samples a natural number

[1] The Coq code is available at http://github.com/adampetcher/fcf

with the appropriate number of bits. In this function, lognat computes the base-2 logarithm (rounded up) of the argument and bvToNat converts a bit vector to the corresponding natural number. The RndNat procedure repeats RndNat_h until the result is less than n, as determined by the function ltNat. It is possible to show that this procedure corresponds with a uniform distribution on numbers in the specified range, and this theorem is present in the FCF library.

```
Definition RndNat_h(n : nat) :=
  v <-$ {0,1} ^ (lognat n); ret (bvToNat v).

Definition RndNat(n : nat) :=
  (Repeat (RndNat_h n) (fun x => (ltNat x n))).
```

Listing 3. Example Program: Random Natural Numbers

C. Program Logic

Many proofs can be completed using the theory of distributions alone, but it can be difficult to complete a proof involving state or looping behavior in this manner. To assist with such proofs, FCF includes a program logic in the style of EasyCrypt. The program logic allows relational judgments on pairs of probabilistic programs. The syntax of a judgment is comp_spec P c1 c2, indicating that relational predicate P holds (probabilistically) on the values produced by programs c1 and c2. A more detailed description of the program logic is provided in [23].

Listing 4 illustrates the program logic using the compMap construction, which maps a computation over a list. This function uses Coq's Fixpoint to destruct the list and apply the computation to the first element, then recursively call compMap on the remainder of the list. Later listings will use a foreach notation to refer to compMap.

The compMap_rel theorem describes a relational program logic judgment for this construction. This judgment requires that some predicate P1 holds on all corresponding pairs of values in lists lsa and lsb (defined using Coq's Forall2). Additionally, for any pair of values a and b on which P1 holds, the relation P2 must hold on the outputs of (c1 a) and (c2 b). Then the theorem states that P2 holds on all corresponding pairs of values in the lists resulting from the map operation.

The relational program logic is a powerful tool for completing proofs of security involving sequences of games. In such a proof, it is necessary to prove that some relation holds on each adjacent pair of games in the sequence. The program logic provides a general mechanism for proving that arbitrary relations hold on subprograms appearing within those games. These judgments can be combined to prove judgments on the entire games, including judgments that correspond to equality, inequality, and closeness of probability distributions.

The compMap_fission theorem is another judgment on compMap describing equivalence of loop fission. Various forms of this theorem, along with similar theorems for probabilistic fold operations, are used extensively in the proofs related to SSE. This theorem can be proved by induction on the list using existing program logic facts and tactics.

```
Fixpoint compMap(c : A -> Comp B)(ls : list A) :
    Comp (list B) :=
  match ls with
    | nil => ret nil
    | a :: lsa' =>
      b <-$ c a;
      lsb' <-$ compMap c lsa';
      ret (b :: lsb')
  end.

Theorem compMap_rel :
  forall (P1 : A -> B -> Prop)(P2 : C -> D -> Prop)
           (lsa : list A)(lsb : list B)
           (c1 : A -> Comp C)(c2 : B -> Comp D),
    Forall2 P1 lsa lsb ->
    (forall a b, In a lsa -> In b lsb ->
      P1 a b -> comp_spec P2 (c1 a) (c2 b)) ->
    comp_spec (Forall2 P2)
                   (compMap c1 lsa)
                   (compMap c2 lsb).

Theorem compMap_fission :
  forall (c1 : A -> Comp B)(c2 : B -> Comp C)
    (ls : list A),
    comp_spec eq
      (compMap (fun a => b <-$ c1 a; c2 b) ls)
      (ls' <-$ compMap c1 ls; compMap c2 ls').
```

Listing 4. Probabilistic Map

D. Additional Features

FCF includes a library of standard programming constructs and associated theory. This library includes the compMap operation seen in Listing 4 as well as other list operations such as probabilistic fold and summation. The library also includes additional constructed sampling routines such as sampling from lists, groups, and arbitrary Bernoulli distributions with rational success probability. FCF also includes a notion of oracles and procedures that have access to oracles. This oracle system is necessary for reasoning about an adversary that is allowed to query a stateful oracle. FCF also provides a conventional operational semantics for its language in order to allow extraction of OCaml programs from FCF constructions. This operational semantics is proven equivalent to the denotational semantics used to reason about programs in security proofs. More information about these additional features of FCF can be found in [23].

III. Cryptographic Arguments in FCF

This section contains some examples to describe how cryptographic arguments are completed in FCF. All of the examples in this section are used in the main proof of this paper.

Listing 5 contains the definition of a non-adaptively secure pseudorandom function (PRF). In this definition, the adversary defined by procedures A1 and A2 attempts to distinguish two "worlds." In both worlds, the adversary produces a list of values (lsD) which are provided to some function, and the corresponding list of outputs (lsR) is given back to the adversary. The adversary may also share arbitrary state (s_A) between these two procedures. In the first world, the outputs are produced by some function f, whereas in the second world these outputs are produced by a random function. This random function is modeled as a stateful oracle called randomFunc that keeps track of previous inputs and outputs using a list.

The oracleMap function is used to map this oracle over the list lsD, and nil is the initial state of the oracle. The second adversary procedure takes the resulting list of function outputs and the state, and produces a bit. This definition ends by defining the *advantage* of the adversary as the distance between the probability that the adversary produces true in these two games. If f is a PRF, then this advantage should be "small."

```
Definition PRF_NA_G_A : Comp bool :=
  [lsD, s_A] <-$2 A1;
  lsR <-$ (k <-$ RndKey; ret (map (f k) lsD));
  A2 s_A lsR.

Definition PRF_NA_G_B : Comp bool :=
  [lsD, s_A] <-$2 A1;
  [lsR, _] <-$2 oracleMap randomFunc nil lsD;
  A2 s_A lsR.

Definition PRF_NA_Advantage :=
  | Pr[PRF_NA_G_A] - Pr[PRF_NA_G_B] |.
```

Listing 5. Non-Adaptively Secure Pseudorandom Function

The security definition in Listing 5 can be used as either the end goal of a proof (in order to show that some function is a PRF) or an assumption (to assume that some function is a PRF). When used as an assumption, we can unify some game with PRF_NA_G_A and another with PRF_NA_G_B and replace the distance between these two games with the corresponding PRF_NA_Advantage. This technique effectively allows us to rewrite one game with another while adding a "small" value to the bounds produced by the proof.

Listing 6 contains the structure of a hybrid argument [19, 18] that bounds the probability that an adversary can distinguish two distributions when given a list of samples from one of the distributions (ListHybrid_Adv). The resulting bound is a function of the advantage of the adversary when attempting to distinguish these two distributions given only a single sample (DistSingle_Adv). If the adversary is unlikely to distinguish these distributions when given a single sample, then the adversary is still unlikely to distinguish these distributions when given polynomially many samples. To make this argument more general, the adversary is able to influence the distribution by providing a value (in the case of DistSingle_G) or a list of values (in the case of ListHybrid_G).

In this listing, B1 and B2 (omitted) compose a nat-indexed family of adversaries constructed from A1 and A2, where the *i*th adversary attempts to distinguish the single sample implied by the *i*th distribution in the appropriate hybrid distribution family. In Single_impl_ListHybrid_sum, the bound is given as a sum over the advantages of these adversaries, and maxA is the maximum size of the list provided by A1. If we include an assumption that a single value (maxAdvantage) bounds the advantage of each of these adversaries, then we can derive the simpler result of Single_impl_ListHybrid.

Note that PRF_NA_Advantage unifies with DistSingle_Adv. So we if we assume that some function is a PRF, then we can use the hybrid argument above to conclude that the function is indistinguishable from a random function even when the adversary provides a list of lists of inputs, and receives the result of the PRF mapped over each list (using a different key for each list). In this way, we can obtain an *iterated* PRF which is useful in the SSE proof.

```
Definition DistSingle_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ c a;
  A2 s_A b.

Definition DistSingle_Adv :=
  | Pr[DistSingle_G c1] - Pr[DistSingle_G c2] |.

Definition ListHybrid_G (c : A -> Comp B) :=
  [lsA, s_A] <-$2 A1;
  lsB <-$ foreach (x in lsA) (c x);
  A2 s_A lsB.

Definition ListHybrid_Adv :=
  | Pr[ListHybrid_G c1] - Pr[ListHybrid_G c2] |.

Theorem Single_impl_ListHybrid_sum :
  ListHybrid_Adv <=
    sumList (forNats maxA)
      (fun i => DistSingle_Adv c1 c2 (B1 i) B2).

Hypothesis maxAdvantage_correct :
  forall i,
    DistSingle_Adv c1 c2 (B1 i) B2 <= maxAdvantage.

Theorem Single_impl_ListHybrid :
  ListHybrid_Adv <= maxA  * maxAdvantage.
```

Listing 6. A Hybrid Argument on Lists

IV. Searchable Symmetric Encryption

This section informally introduces Searchable Symmetric Encryption and describes the strategy used in the proof of security. An SSE scheme provides a mechanism to encrypt a database and a list of queries. These encryptions are given to an untrusted party who is able to produce encryptions of the result of executing the queries on the database while learning very little about the database or queries. We call the party that knows the unencrypted database the *client*, and the untrusted party that executes queries on the encrypted database is the *server*. A database is simply a list of identifiers and a set of keywords associated with each identifier. Each identifier can be used to retrieve some other object in an encrypted database, but this operation is beyond the scope of the SSE definitions.

The SSE scheme of [15] is constructed from an abstraction called a Tuple Set (or T-Set) that behaves like a secure associative array. In this paper, we limit ourselves to single-keyword SSE (SKS-SSE), in which each query is a single keyword. Roughly speaking, this scheme works by encrypting each value using a key derived from the appropriate keyword, and then storing the ciphertexts in a T-Set. The server can perform a query by looking up the specified keyword in the T-Set and giving the resulting ciphertexts to the client.

Figure IV describes the structure of the security proof. Each node in the diagram is an object that is conjectured (in the case of PRF) or proved to exist, and each arrow is a reduction that proves the existence of some construction. Many of these reductions are complex arguments involving a large sequence of games. In particular, the T-Set construction and the proofs related to this construction are quite complex, and the T-Set abstraction hides the complexity of this construction in order to make the SSE proof simpler. This is a standard technique in cryptography that is even more important when developing mechanized proofs. The abstraction and modular construction features of Coq, which are inherited by FCF, are very useful for developing these sorts of proofs.

The left side of the diagrams shows the proof that the T-Set construction in [15] is secure and correct, and the right side is the proof of security of the SKS-SSE scheme. In the T-Set proof, we begin by showing that if some function f is a PRF, then is is an *iterated* PRF as described in Section III. From a PRF and an iterated PRF, we show that a simplified "single-trial" form of the T-Set construction is correct and secure. Then we use some reusable arguments to obtain the correctness and security of the "full" T-Set construction. More information about the T-Set security and correctness proofs can be found in Sections VI-A and VI-B, respectively

The proof of security for SKS-SSE, requires an IND-CPA encryption scheme, which can be formally derived from a PRF as shown in [23]. We then show that this encryption scheme is an *iterated* encryption scheme in a manner similar to the iterated PRF reduction. This fact also follows from the hybrid argument described in Section III. Then we show that the SKS-SSE scheme is secure as long as the T-Set is correct and secure, the encryption scheme used is an iterated IND-CPA encryption scheme, and the function used to derive encryption keys is a PRF. We expand on this part of the proof in Section V.

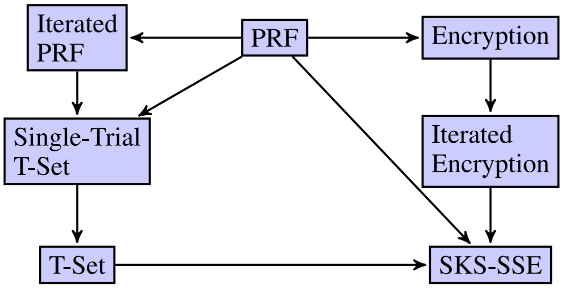

Fig. 1. SSE Security Proof Structure

V. Single Keyword Searchable Symmetric Encryption from Tuple Sets

In this section, we present the formal definitions related to SSE and Tuple Sets, and prove the security of the SKS-SSE scheme of [15]. An SSE scheme consists of an `EDBSetup` function that takes a database and produces an encrypted database and a key, and a `SearchProtocol` that uses a key and a query known to the client and an encrypted database known to the server to produce a list of identifiers and a transcript.

A. Non-Adaptively Secure SSE

We use a non-adaptively secure definition for SSE (Listing 7), in which an adversary produces a database and the entire list of queries up front. The definition is given as an indistinguishability between a pair of games parameterized by a leakage function `L` that describes the information that is allowed to leak to the adversary. The real game uses the actual `EDBSetup` and `SearchProtocol` while the ideal game uses a simulator that is only given the result of the leakage function applied to the unencrypted database and list of queries. The SSE scheme is non-adaptively secure if the distance between these two games, `SSE_NA_Advantage`, is small.

In this definition, the adversary is divided into two separate procedures, `A1`, and `A2` which are allowed to share state. In the equivalent definition in [15], the second adversary procedure is also given the list of identifiers resulting from the queries in order to model the assumption that the client will immediately give the identifiers to the server to retrieve the required objects. For simplicity, we remove this assumption and only give the search transcript to the adversary.

```
Definition SSE_Sec_NA_Real :=
  [db, q, s_A] <-$3 A1;
  [k, edb] <-$2 EDBSetup db;
  ls <-$ foreach (x in q) (SearchProtocol edb k x);
  A2 s_A edb (snd (split ls)).

Definition SSE_Sec_NA_Ideal :=
  [db, q, s_A] <-$3 A1;
  leak <-$ L db q;
  [edb, t] <-$2 Sim leak;
  A2 s_A edb t.

Definition SSE_NA_Advantage :=
  | Pr[SSE_Sec_NA_Real] - Pr[SSE_Sec_NA_Ideal] |.
```

Listing 7. SSE Non-Adaptive Security

B. T-Sets

A T-Set is a primitive that associates values with keywords, and allows retrieval of all the values associated with some keyword. A T-Set differs from a standard associative array in that the T-Set scheme attempts to hide as much as possible about the values in the T-Set and the relationship between keywords and values. A server that possesses a T-Set structure but not the key for that structure should learn very little about the contents of the structure. The server can also query the structure on behalf of a client that knows the T-Set key, and in the process the server should learn very little other than the set of values returned by the query.

A T-Set scheme is composed of three procedures: `TSetSetup`, `TSetGetTag`, and `TSetRetrieve`. `TSetSetup` takes a database and returns a T-Set and a secret key. Database keywords are elements of $\{0,1\}^*$ and identifiers are elements of $\{0,1\}^\lambda$. `TSetGetTag` takes a keyword and a secret key and outputs a tag. `TSetRetrieve` takes a T-Set and a tag and returns a list of identifiers.

The security of the SSE scheme relies on both the security and the correctness of the T-Set scheme. The formal correctness definition (Listing 8) is computational and non-adaptive. In this definition, the adversary chooses the database and list of keywords, and the correct answers are compared to the answers produced using the T-Set. If the T-Set is correct, then the probability that these answers differ (`AdvCor`) is small.

The non-adaptive security of a T-Set scheme is defined as a real/ideal indistinguishability parameterized by a leakage function `L` as shown in Listing 9. If the T-Set is secure, then `TSetAdvantage` should be small. Note that the correct answers are given to the simulator in the ideal game, implying that this information is allowed to leak to the adversary. The

```
Definition AdvCor_G :=
  [t, q] <-$2 A;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q) (TSetGetTag k_T x);
  t_w <- foreach (x in tags) (TSetRetrieve tSet x);
  t_w_correct <- foreach (x in q)
    (arrayLookupList t x);
  ret (t_w != t_w_correct).

Definition AdvCor := Pr[AdvCor_G].
```

Listing 8. T-Set Non-Adaptive Computational Correctness

T-Set only hides information about the queries and the non-queried portions of the database.

```
Definition TSetReal :=
  [t, q, s_A] <-$3 A1;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q) (TSetGetTag k_T x);
  A2 s_A (tSet, tags).

Definition TSetIdeal :=
  [t, q, s_A] <-$3 A1;
  T_qs <- foreach (x in q) (lookupAnswers t x);
  [tSet, tags] <-$2 Sim (L t q) T_qs;
  A2 s_A (tSet, tags).

Definition TSetAdvantage :=
  | Pr[TSetReal] - Pr[TSetIdeal] |.
```

Listing 9. T-Set Security Definition

C. IND-CPA Encryption and PRFs

The final elements required to construct the SSE scheme are an IND-CPA encryption scheme and a pseudorandom function. The T-Set is allowed to leak information about values returned by queries, so the SSE scheme stores ciphertexts in the T-Set instead of indices. Because the encryption is IND-CPA, the only information leaked is the number of values returned by each query. The encryption key is determined by a pseudorandom function applied to the appropriate keyword. We use adaptively-secure encryption and PRFs in this proof merely for convenience, and it would be possible to complete this proof using non-adaptive forms of these assumptions. The adaptive security definitions use the `OracleComp` type described in [23].

The particular IND-CPA definition that is used as an assumption is shown in Listing 10. In this definition, `EncryptOracle` is an oracle that returns an encryption of any plaintext it receives, and `EncryptNothingOracle` takes a plaintext and returns an encryption of some default value (e.g. 0^{λ}). The scheme encrypts each entry using a key derived from the keyword, so we actually need an iterated form of IND-CPA in which the adversary is allowed to interact with several encryption oracles, each with a different key. We can show that any IND-CPA encryption scheme is also an iterated IND-CPA encryption scheme (security definition omitted) using the hybrid argument described in Section III.

The SSE construction also requires a standard pseudorandom function. The definition (omitted) is the adaptive form of the PRF definition provided in Section III.

```
Definition IND_CPA_SK_O_G0 :=
  key <-$ KeyGen;
  [b, _] <-$2 A (EncryptOracle key) tt;
  ret b.

Definition IND_CPA_SK_O_G1 :=
  key <-$ KeyGen;
  [b, _] <-$2 A (EncryptNothingOracle key) tt;
  ret b.

Definition IND_CPA_SK_O_Advantage :=
  | Pr[IND_CPA_SK_O_G0] - Pr[IND_CPA_SK_O_G1]  |.
```

Listing 10. Iterated IND-CPA Encryption

D. SKS-SSE Construction

The formalization of the SKS-SSE construction is shown in Figure 11. In this figure, `Enc` and `Dec` are the encryption and decryption procedures for an IND-CPA encryption scheme, and `F` is a PRF. The `EDBSetup` routine iterates over all keywords in the database (obtained using the `toW` function) and encrypts the indices associated with each keyword under a key derived from that keyword. Then `TSetSetup` is used to construct a T-Set from this encrypted database. In this procedure, `lookupInds` returns all the indices associated with a keyword. The search protocol uses `TSetGetTag` and `TSetRetrieve` to get the encrypted indices, and then decrypts them.

```
Definition SKS_EDBSetup_wLoop db k_S w :=
  k_e <- F k_S w;
  inds <- lookupInds db w;
  t <-$ foreach (x in inds) (Enc k_e x);
  ret (w, t).

Definition SKS_EDBSetup(db : DB) :=
  k_S <-$ {0, 1}^lambda;
  t <-$ foreach (x in (toW db))
    (SKS_EDBSetup_wLoop db k_S x);
  [tSet, k_T] <-$2 TSetSetup t;
  ret ((k_S, k_T), tSet).

Definition SKS_Search tSet k w :=
  [k_S, k_T] <-2 k;
  (* client *) tag <-$ TSetGetTag k_T w;
  (* server *) t <- TSetRetrieve tSet tag;
  (* client *) inds <- map (Dec (F k_S w)) t;
  ret (inds, (tag, t)).
```

Listing 11. SKS-SSE Construction

E. Proof of Security for SKS-SSE

Listing 12 contains the leakage function and simulator used in the proof of security. Note that `L_T` is the leakage function for the T-Set. Informally, this scheme leaks the number of indices associated with each queried keyword, as well as the result of the T-Set leakage function applied to the *structure* of the database (the number of indices associated with keyword) and the list of queries. The simulator for this proof uses `Sim_T`, which is the T-Set simulator. In this listing, `zeroVector lambda` is a vector of length `lambda` containing all zeroes, and `combine` is the Coq function that converts a pair of lists to the corresponding list of pairs.

The security proof is completed using a sequence of games (which is omitted for brevity). The exact security result is provided in Listing 13. The result refers to procedures `TSetCor_A`, `TSetSec_A1`, `TSetSec_A2`, `PRF_A`,

```
Definition SKS_resultsStruct db w :=
  k_e <-$ {0, 1}^lambda;
  inds <- lookupInds db w;
  foreach (_ in inds)
    (Enc k_e (zeroVector lambda)).

Definition L (db : DB) (qs : list Query) :=
  t_s<-$ foreach (x in (toW db))
    (SKS_resultsStruct db x);
  t <- combine (toW db) t_s;
  leak_T <- L_T t qs;
  ret (leak_T, map (arrayLookupList t) qs).

Definition SKS_Sim leak :=
  [leak_T, struct] <-2 leak;
  [tSet, tags] <-$2 Sim_T leak_T struct;
  ret (tSet, (combine tags struct)).
```

Listing 12. Leakage Function and Simulator for SKS-SSE Proof

Enc_A1, and Enc_A2 (all omitted), which form the constructed adversaries against T-Set correctness and security, the PRF, and the IND-CPA encryption scheme. Enc_A1 is a family of procedures, and the hypothesis states that IND_CPA_Adv is an upper bound on the advantage of all procedures in this family. The term maxKeywords represents the maximum number of keywords that may be contained in the database and queries produced by A1, and this term appears in the bounds due to the application of the hybrid argument as described in Section V-C.

```
Theorem SKS_Secure :
  (forall i, IND_CPA_SK_O_Adv ({0, 1}^lambda) Enc
    (Enc_A1 i) Enc_A2 <= IND_CPA_Adv) ->
  SSE_NA_Advantage SKS_EDBSetup
    SKS_Search A1 A2 L SKS_Sim <=
  AdvCor TSetSetup TSetGetTag TSetRetrieve
    TSetCor_A +
  TSetAdvantage TSetSetup TSetGetTag L_T
    TSetSec_A1 TSetSec_A2 Sim_T +
  PRF_Advantage (Rnd lambda) (Rnd lambda) F PRF_A +
  maxKeywords * IND_CPA_Adv.
```

Listing 13. Exact Security of SKS-SSE Scheme

VI. T-Set Instantiation

This section describes the efficient T-Set instantiation described in [15] as well as the formal proof of security and correctness of this construction. We slightly simplify the model of the T-Set construction because we only prove non-adaptive security of the scheme. Instead of using two PRFs and a random oracle, we model the scheme using only two PRFs, since this is sufficient for non-adaptive security.

The T-Set is a hash table with B buckets, each with at most S entries. The parameters B and S are selected based on the size of the input structure T in a way that the probability of constructing the T-Set without running out of space in any bucket is non-negligible. A PRF F is used to determine the bucket into which each value will be placed, as well as a label that can be used to determine the keyword associated with the value, and a key used to encrypt the value when it is placed in the T-Set. Another PRF $\bar{F}$ is used to map keywords to tags. The security of the T-Set scheme is derived from the assumed indistinguishability of F and $\bar{F}$ from random functions.

In order to organize the presentation and proof, we separate the TSetSetup routine into a number of subroutines. This routine is composed of a nested loop, so we construct a procedure for each loop body. Each loop body is a function that takes an accumulator and the next input value and returns the resulting value for the accumulator. The loop_over operator is simply notation for folding the procedure over some input list. The setup routine may fail if some bucket in the hash table is filled, so the setup is repeated in independent trials until a trial succeeds. In this listing, nth is a Coq function that returns the value at a certain position in a list, remove removes the first occurrence of some value in a list, replace replaces the value in a list at a specified position with another value, tSetUpdate sets the value in the T-Set at the specified location to the provided value, lookupAnswers returns the indices associated with some keyword in the T-Set, allNatsLt returns all the natural numbers (in increasing order) less than a specified number, and initFree initializes a "free list" that is used to keep track of which locations in each bucket are unoccupied. The ($ free_b) expression in the TSetSetup_tLoop construction denotes sampling from the distribution corresponding to the list free_b. This sampling routine and notation are provided by the FCF standard library. Because this sampling may fail if the list is empty, we perform the sampling inside a *Maybe* monad as indicated by the arrow <-?, and the TSetSetup_tLoop returns a value in an option type.

```
Definition TSetSetup_tLoop stag length acc e :=
  [tSet, free] <- acc;
  [i, s_i] <- e; [b, L, K] <- F stag i;
  free_b <- nth b free nil;
  j <-? ($ free_b) ;
  free <- replace free b (remove free_b j);
  bet <- (S i) != length;
  newRecord <- (L, (bet :: s_i) xor K);
  tSet <- tSetUpdate tSet b j newRecord;
  ret (tSet, free).

Definition TSetSetup_wLoop T k_T acc w :=
  [tSet, free] <- acc;
  stag <- F_bar k_T w;
  t <- lookupAnswers T w;
  ls <- combine (allNatsLt (length t)) t;
  loop_over ((tSet, free), ls)
    (TSetSetup_tLoop stag (length t)).

Definition TSetSetup_trial T  :=
  k_T <-$ {0, 1} ^ lambda;
  loopRes <-$ loop_over ((nil, initFree), (toW T))
    (TSetSetup_wLoop T k_T) ;
  ret (loopRes, k_T).

Definition TSetSetup t :=
  [res, k_T] <-$ Repeat (TSetSetup_trial t)
    (fun p => isSome (fst p));
  ret (getTSet res, k_T).
```

Listing 14. T-Set Setup Routine

The TSetGetTag procedure (Listing 15) simply produces a tag for a keyword using the $\bar{F}$ PRF and the key for the T-Set.

```
Definition TSetGetTag (k_T : Bvector lambda) w :=
    ret (F_bar k_T w).
```

Listing 15. T-Set Get Tag Routine

The TSetRetrieve procedure (omitted) searches through the T-Set to find all the entries matching a keyword. Because Coq requires us to model this procedure as a total function, we assume that there is a maximum number of entries for any keyword, and we use this number as "fuel". The loop body searches for the i_{th} value matching the tag, and returns an optional value and an indication of whether there are additional entries matching the tag. This loop body is iterated until it indicates that there are no more values, or it runs out of fuel.

A. *T-Set Security*

The simulator used in the security proof is shown in Listing 16. The leakage function (not shown) returns the total size of the database, and this value is given to the simulator as the parameter leak.

```
Definition randomTSetEntry acc :=
  label <-$ {0, 1} ^ lambda;
  value <-$ {0, 1} ^ (S lambda);
  [tSet, free] <- acc;
  b <-$ [0 .. B);
  free_b <- nth b free nil;
  j <-? ($ free_b);
  free <- replace free b (remove free_b j) nil;
  tSet <- tSetUpdate tSet b j (label, value);
  ret (tSet, free).

Definition TSetSetup_Sim_wLoop tSet_free e :=
  [tSet, free] <- tSet_free;
  [stag, t] <- e;
  ls <- combine (allNatsLt (length t)) t;
  loop_over ((tSet, free), ls)
    (TSetSetup_tLoop stag (length t)).

Definition TSet_Sim_trial n ts :=
  tags <-$ foreach (_ in ts) ({0, 1} ^ lambda);
  loopRes <-$ loop_over
    ((nil, initFree), (combine tags ts))
    TSetSetup_Sim_wLoop;
  loopRes <-$ loop_over
    (loopRes, allNatsLt (n - length (flatten ts)))
    (fun acc i => randomTSetEntry acc);
  ret (loopRes, tags).

Definition TSet_Sim leak ls :=
  [_, ts] <- split ls;
  [trialRes, tags] <-$
    Repeat (TSet_Sim_trial leak ts)
    (fun p => isSome (fst p));
  ret (getTSet trialRes, tags).
```

Listing 16. T-Set Simulator

This proof is complicated by the fact that the real setup routine and the simulator perform multiple trials in an attempt to create the T-Set. So we begin by proving the security of a modified form of the scheme in which only one attempt is made to construct the T-Set. Then we combine this result with some additional arguments in order to obtain the proof of security for the full T-Set scheme.

1) Single-Trial T-Set Security: The Single-Trial T-Set security proof is a straightforward, though complicated, sequence of games in which we replace PRFs with random values and use the resulting randomness to show that the output is independent from the input. The first complication relates to applying the PRF definition to F in that some of the PRF keys are the same as the tags that are given to the adversary at the end of the computation. The PRF definition only applies when the PRF key is not given to the adversary, so we must split the T-Set initialization procedure into two parts: first it adds entries related to the keywords that are queried by the adversary, then it adds the rest of the entries. The first part of this procedure already matches the ideal functionality, and we only apply the PRF assumption to the entries created during the second part of the procedure. Another complication is that the initialization procedure places each record in a random location in the correct bucket. So it is necessary to perform game manipulations in the presence of sampling *without replacement*, and the games must keep track of the unused locations in each bucket.

The intermediate game code is omitted for brevity, but a diagram of the sequence is provided in Figure 2. The box around the top half marks a portion of the proof that is reused as an argument in the correctness proof described in Section VI-B1. Each equivalence in the diagram is labeled to indicate the argument or assumption used. Equivalences labeled S are simple transformations such as unfolding definitions, inlining statements, and removing unused values or statements. F indicates a loop fission transformation such as the one described in Section II-C. A describes an information augmentation transformation in which additional information is added to a data structure without changing the results of the game. Such a transformation enables "ghost state" reasoning in which this additional information can be used in program logic judgments. For example, a list of ciphertexts could be augmented with a list of plaintexts and keys used in the encryption. Then a program logic judgment could state that the plaintext is equal to the value obtained by decrypting the ciphertext with the key. D is a dimension reduction where a data structure of dimension n is represented using a data structure of dimension $n - 1$. A dimension reduction may be performed to replace a 2-dimensional data structure with a list in order to apply a theorem related to list processing. O is a non-trivial change to the order in which statements are executed in the game. The T-Set construction stores entries in a random location in each bucket, requiring sampling without replacement to determine the location of each entry. In some transformations, we change the order that entries are added to the T-Set in the presence of this sampling without replacement. R equivalences replace random function outputs with independent random values by showing that there are no duplicates in the input to the function. In L transformations, we show that folding the function f over a list is equivalent to folding f over the first n elements of the list, and then folding f over the rest of the list. I equivalences show that certain values are independent of each other by applying a one-time pad argument as seen Section III.

The statement of security for single-trial T-Sets is shown in Listing 17. In this listing, TSetSetup_once and TSet_Sim_once are procedures that try to create a T-Set in a single attempt using the corresponding trial routines. These routines produce an empty T-Set if the trail fails. The procedures TSet_PRF_A1, TSet_PRF_A2, TSet_IPRF_A1, and TSet_IPRF_A2, are efficient adversaries against the PRFs constructed from A1 and A2. The proof uses an iterated PRF as described in Section III, and TSet_IPRF_A1 and TSet_IPRF_A2 form a family of adversaries constructed using different distributions from the appropriate hybrid distri-

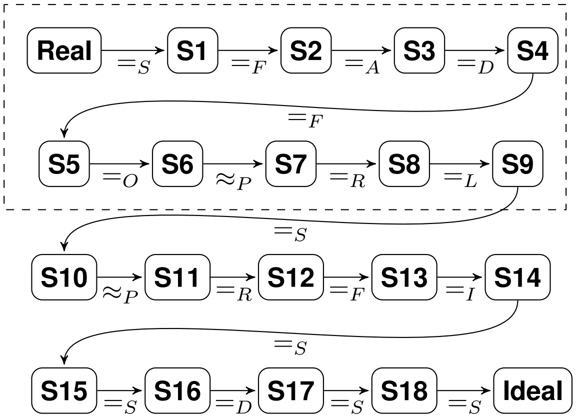

Fig. 2. Single-Trial T-Set Security Games

bution family. This theorem assumes that F_Adv is an upper bound on the advantage of all of these adversaries against the PRF F. The theorem also assumes that F_bar_Adv is an upper bound on the advantage of a particular constructed adversary against the PRF F_bar. Similar to the proof in Section V-E, the database and queries provided by the adversary contain at most maxKeywords keywords, and this term appears in the bounds due to the application of the hybrid argument.

```
Theorem TSet_once_secure :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) (RndF_Range) F
    (TSet_IPRF_A1 i) TSet_IPRF_A2 <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    TSet_PRF_A1 TSet_PRF_A2 <= F_bar_Adv ->
  TSetAdvantage TSetSetup_once TSetGetTag
    L_T TSet_Sim_once A1 A2 <=
  <= F_bar_Adv + maxKeywords * F_Adv.
```

Listing 17. Single-Trial T-Set Security

2) The "One to Many" Argument: We employ a couple of non-trivial reusable arguments in order to derive security of the full T-Set scheme from the proof of security of the *Single Trial T-Set* scheme. The first of these arguments is the "One to Many" argument (Listing 18), which is a special case of the hybrid argument described in Section III in which the same argument is repeated a fixed number of times and the results are collected in a list.

```
Definition DistMult_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ foreach (x in (forNats n)) ((c a);
  A2 s_A b.

Definition DistMult_Adv :=
  | Pr[DistMult_G c1] - Pr[DistMult_G c2] |.

Theorem DistSingle_impl_Mult :
  DistMult_Adv c1 c2 A1 A2 n <=
    n * (DistSingle_Adv c1 c2 B1 B2).
```

Listing 18. The One to Many Theoreom

3) The "Many to Core" Argument: The next argument applies to any pair of probabilistic computations c_1 and c_2 that produce values of type B. There is also some predicate P on values of type B that defines the "core" of the distributions corresponding to c_1 and c_2. This argument shows that if any efficient adversary A can effectively distinguish c_1 from c_2 when given a single value from c_1 or c_2 such that $P(b) = true$, then there exists an efficient adversary A' that can effectively distinguish c_1 from c_2 when given (polynomially) many samples from one of the distributions. An additional condition required for this fact to hold is that the total probability mass of the core is not too small. The statement of this argument is shown in Listing 19, where k1 and k2 represent the probability mass of the core of c1 and c2, respectively.

```
Definition RepeatCore_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ Repeat (c a) P;
  A2 s_A b.

Definition RepeatCore_Adv :=
  | Pr[RepeatCore_G c1] - Pr[RepeatCore_G c2] |.

Theorem DistMult_impl_RepeatCore :
  RepeatCore_Adv P c1 c2 A1 A2 <=
    DistMult_Adv c1 c2 A1 DM_RC_B2 n +
    (1 - k1)^n + (1 - k2)^n.
```

Listing 19. The Many to Core Theoreom

The proof of this fact is intuitive, and is illustrated in Figure 3. If the core of the distribution is sufficiently large, and if enough samples are taken from the distribution, then it is likely that at least one of these samples will fall within the core of the distribution. The constructed adversary A' samples the distribution n times and gives the first "hit" in the core of the distribution to A which it uses to determine the source of the sample. When a hit is obtained, the distribution observed by A is identical to the distribution in which only the core is sampled. These distributions only differ when no hit is obtained after n attempts, but this event has negligible probability in n.

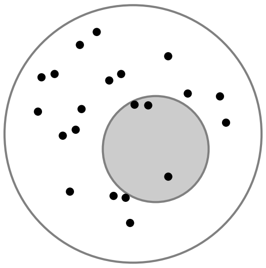

Fig. 3. Illustration of "Many to Core" Argument

4) Full T-Set Security: We obtain security of the full T-Set scheme by combining the arguments in the previous sections. In order to apply the "Many to Core" argument, it must be shown that there is some positive $k \in \mathbb{Q}$, and the probability of successfully creating a T-Set from a database supplied by the adversary is at least k. This argument also requires that the simulator succeeds in one trial with probability at least k. Because these facts depend on the choice of parameters B and S, we leave them as assumptions in the proof.

By combining the *Single-Trial T-Set* security proof with the assumptions related to k described in the previous paragraph, and with the arguments presented in Sections VI-A2 and VI-A3, we get the final security result in Listing 20. This theorem has the same assumptions as the "Single-Trial" security theorem in Listing 17, and the bounds of that theorem are present in this one.

```
Theorem TSet_secure :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) (RndF_Range) F
    (TSet_IPRF_A1 i) TSet_IPRF_A2 <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    TSet_PRF_A1 TSet_PRF_A2 <= F_bar_Adv ->
  TSetAdvantage TSetSetup TSetGetTag
    L_T TSet_Sim A1 A2
  <= lambda * (F_bar_Adv + maxKeywords * F_Adv)
    + 2 * (1 - k)^lambda
```

Listing 20. T-Set Security

B. T-Set Correctness

The T-Set correctness proof has very similar structure to the security proof. The primary difference is that the ultimate goal is an inequality, rather than a proof that two values are "close." The proof uses slightly different forms of the "One to Many" and "Many to Core" arguments, and there are some interesting differences in the "single-trial" proof, which we highlight in this section.

1) Single-Trial T-Set Correctness: The single-trial T-Set security proof was simplified by the fact that security is obvious when initialization fails. The empty T-Set resulting from an initialization failure clearly has no information that the adversary could use to distinguish it from the simulator. This argument is not so simple in the case of correctness, because an empty T-Set is obviously *not* correct. So we instead prove that the single-trial construction is conditionally correct. That is, a database and list of queries produced by the adversary is highly unlikely to result in a T-Set *on the first initialization attempt* that will produce an incorrect answer when queried. In the formalization of this definition (Listing 21), `good` is a predicate that indicates whether the TSetSetup routine produced a valid T-Set.

```
Definition AdvCor_C_G :=
  [t, q] <-$2 A;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q)(TSetGetTag k_T x);
  t_w <- foreach (x in tags) (TSetRetrieve tSet x);
  t_w_correct <- foreach (x in q)
    (arrayLookupList _ t x);
  ret (good tSet && (t_w != t_w_correct)).

Definition AdvCor_C := Pr[AdvCor_C_G].
```

Listing 21. T-Set Conditional Correctness

Notice that AdvCor_C_G unifies with the real game in the T-Set security definition (Listing 9). Since this definition is used in the single-trial T-Set security proof, we could use the result of this proof in the correctness proof to replace the game above with the ideal game from the security proof. Unfortunately, the simulator in the security proof eliminates some of the information required to show correctness. This security proof is a sequence of games, however, and we can use it to replace the game above with any game in that sequence. There is a game about halfway through in which many simplifications have been applied and the first PRF outputs are replaced with random values. So we save a significant amount of effort by reusing this result.

Next we perform a sequence of manipulations that simplify the T-Set and make it look more like the input database. For example, we put the values in the buckets in the same order as the input list rather than in a random order, we store and retrieve actual values instead of encryptions of values, and we make the structure one-dimensional. Then we replace the remaining PRF with a random function and replace the outputs with random values. Finally, we show that the T-Set is correct as long as there are no collisions in these random values, and we derive an expression for the probability of such a collision.

The sequence of games is diagrammed in Figure 4. The proof uses several of the same forms of equivalence from the security proof, and only the new labels are described in this paragraph. The equivalence labeled M uses the part of the security proof surrounded by a box in Figure 2 as an argument. In inequalities labeled C, we modify the game so that the adversary can also win by finding a collision during some operation. That is, the adversary can win by getting the game to produce a collision, or by satisfying the original "win" condition when there is no such collision. This allows a form of "identical until bad" reasoning for inequalities in which we can assume that there are no collisions going forward, and we will calculate the probability of collision and add it to the bounds in a later stage of the proof. E represents an equivalence by functional injection, in which we replace some operation on the outputs of an injective function with a related operation on the inputs of the function. These equivalences may use the assumptions provided by C steps, because if no collisions are encountered while interacting with a function, then that function behaves like an injection. In the final N equivalence of the correctness proof, we convert a simple collision-finding game into the corresponding probability expression B. The expression B is negligible in λ, and the bound on the advantage of the adversary in this theorem is the sum of B and the PRF advantage terms introduced by the $\approx$ equivalences.

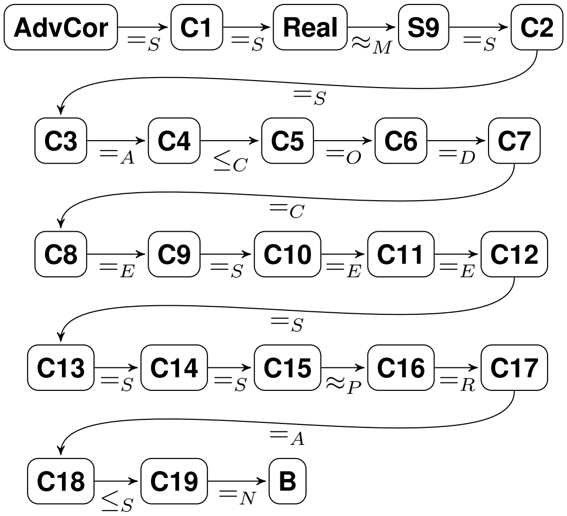

Fig. 4. Single-Trial T-Set Correctness Games

The single-trial conditional correctness result is in Listing 22. In this listing, `maxMatches` is the maximum number of records matching any query, and `maxKeywords` is the maximum number of keywords in the database and queries supplied by the adversary. This result is similar to the single-trial security result because both proofs assume the functions

F and F_bar are PRFs, and F is used as an *iterated* PRF in both proofs. The first term in the bounds of this theorem corresponds with B—the probability of a collision that would cause the result to be incorrect.

```
Theorem TSet_Correct_once :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) RndF_R F
    (PRFI_A1 i) (PRFI_A2) <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    PRF_A1 PRF_A2 <= F_bar_Adv ->
  AdvCor_C TSetSetup_once TSetGetTag
    TSetRetrieve A1 A2 <=
  (maxKeywords * (S maxMatches))^2 / 2 ^ lambda
  + maxKeywords * F_Adv + F_bar_Adv.
```

Listing 22. Single-Trial T-Set Conditional Correctness

2) One to Many to Core Arguments: The "One to Many" and "Many to Core" arguments are slightly different from the ones used in the security proof. Rather than showing that the distance between two events is small, we only need to show that the probability of some event is small under the assumption that the probability of some other event is small. The required arguments are shown in Listing 23.

```
Definition TrueSingle_G :=
  a <-$ A1; b <-$ c a; ret (Q b).

Definition TrueMult_G :=
  a <-$ A1;
  bs <-$ foreach (x in (forNats n)) (c a);
  ret (fold_left (fun b x => b || (Q x)) bs false).

Definition TrueRepeat_G :=
  a <-$ A1; b <-$ Repeat (c a) P; ret (Q b).

Theorem TrueSingle_impl_Mult :
  Pr[TrueMult_G n] <= n * Pr[TrueSingle_G].

Theorem TrueMult_impl_Repeat :
  Pr[TrueRepeat_G] <=
  Pr[TrueMult_G n] + (k ^ n).
```

Listing 23. One to Many to Core Inequality Arguments

C. Full T-Set Correctness

The full T-Set correctness theorem is shown in Listing 24. This result is produced in a similar manner to the security result—the single-trial result is combined with the "One to Many" and "Many to Core" arguments along with some additional assumptions, and the single-trial bounds appears in the bound of the full T-Set result. This proof also assumes a value k representing the probability that the TSetSetup routine succeeds in any attempt.

```
Theorem TSet_Correct :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) RndF_R F
    (PRFI_A1 i) (PRFI_A2) <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    PRF_A1 PRF_A2 <= F_bar_Adv ->
  AdvCor TSetSetup TSetGetTag TSetRetrieve A1 A2 <=
  (1 - k)^lambda + lambda *
    ((maxKeywords * (S maxMatches))^2 / 2 ^ lambda
    + maxKeywords * F_Adv + F_bar_Adv).
```

Listing 24. T-Set Correctness

VII. Proof Engineering

This proof was completed in approximately 6 months by a person with expert-level knowledge of FCF and moderate knowledge of the SSE scheme in question. Most of this time was spent in the "single-trial" security and correctness proofs. Table I provides the number of lines of Coq code and the number of intermediate games for each proof. To determine the number of intermediate games, we count only those games that would be produced by a cryptographer when developing the structure of the proof. In many cases, a high-level transformation is divided into several smaller transformations, each with its own intermediate game. The games used in these smaller transformations are not counted in the total number of games or to the lines of definition, but they do contribute to the number of lines of proof. The "Supporting Arguments" line measures only the arguments described in Sections VI-A2, VI-A3, and VI-B2. This proof relies on a large amount of existing theory in the FCF library which comprises over 40,000 lines of Coq code, and this effort resulted in several thousand lines of additional reusable theory that was added to the standard library of FCF.

TABLE I. Proof Complexity

Proof	Lines of Definition	Lines of Proof	Games
Single-Trial T-Set Security	447	3515	19
Single-Trial T-Set Correctness	611	5510	19
Supporting Arguments	48	1041	12
T-Set Security	0	1033	0
T-Set Correctness	0	998	0
SSE Scheme Security	257	920	8
Total	1363	13017	58

The table provides separate columns for definition (security definitions, constructions, intermediate games, constructed adversaries, and simulators) and proof (everything else including proof scripts, program logic judgments, and minor intermediate games). This separation proposes a division between the essential, cryptographic portion of the proof and the portion required by the mechanization. The division suggests that the mechanization increased the complexity of the proof by (roughly) a factor of 10. This increase in effort is large, but it should be considered reasonable when viewed in the context of the larger engineering effort of developing an implementation of this scheme. The proof is composed of several arguments, and the more complex arguments are further decomposed into a sequence of games. This decomposition provides ample opportunity to divide the proof development effort among a team of programmers.

An important engineering concern is the extent to which artifacts developed for this proof could be reused in other proofs. Notably, the T-Set that was proved secure and correct in this proof is the same T-Set that is used in the more complex SSE schemes in [15]. By reusing the T-Set and its theory, we could greatly reduce the effort required to prove the security of any scheme that requires a correct or secure T-Set. Of course, the more general-purpose theory that was developed for this SSE proof could be directly reused by any proof.

Another concern is the difficulty of changing the proof artifact to respond to changes in the scheme itself. First consider a minor change, such as a change to the representation (but not the content) of the database. We could address this change by proving that some game using the new database representation is equivalent to an existing game using the old representation. This change adds a new intermediate game to the sequence and increases the size of the proof. Another solution is to use a reduction to prove the security of the modified scheme assuming the security of the original scheme. This is a very powerful and general approach, but it also increases the size of the proof. A third option is to refactor the proof and change the database into an abstraction that could be instantiated with either representation. This solution may require more effort to implement, but it does not increase the size of the proof, and it results in a proof that is more tolerant of these changes in the future.

For more significant changes, it may be very hard to modify the proof. For example, if we wanted to prove adaptive security of the SSE scheme, we would need to change the way the scheme and the adversaries are modeled, add a random oracle, and change many of the security definitions to the appropriate adaptive security forms. This is a completely different proof, and none of the artifacts from the non-adaptive proof would be reused. However, much of the general-purpose theory in FCF that was developed for the non-adaptive security proof would still be applicable in the adaptive security proof.

VIII. Related Work

There has been a large amount of work in the area of formalizing cryptographic proofs in the last decade, but much of this work only involves simple examples used to demonstrate a tool, framework, or proof technique. This section focuses on mechanized proofs in the computational model related to non-trivial or practical constructions.

Several complex proofs have been completed in EasyCrypt, CertiCrypt, and CertiPriv [9], a related system for reasoning about differential privacy. Stoughton [25] proved the security of a simplified version of a private information retrieval protocol. This is a fairly complex three-party protocol, but the simplified scheme only allows a query to retrieve the number of occurrences of a certain keyword in the database, and not the values associated with that keyword. Barthe et al. [4] demonstrate a formalization of differential privacy and a verification of a non-trivial smart metering system as an example. Almeida et al. [1] prove the security of a standardized public key encryption scheme. Barthe et al. [8] proved security of OAEP in CertiCrypt. Though this is a relatively simple construction, the proof of security is quite complex, comprising over 10,000 lines of Coq code. Barthe et al. [6] proved using CertiCrypt that a hash function into elliptic curves is indistinguishable from a random oracle. This is a non-trivial proof that incorporates a significant amount of Coq theory related to elliptic curves, and it uses an argument similar to the "Many to Core" arguments of this paper.

Bhargavan et al. [11] verify an implementation of TLS using the F7 refinement type system. This is a remarkably complex proof, but several steps of the proof must be verified by hand due to the fact that F7 does not support reasoning about non-zero statistical distance between distributions. Barthe et al. [5] show how a variant of F* (a successor to F7) can be used to verify implementations of cryptographic schemes. This work provides several non-trivial examples including a certified privacy-preserving system for smart metering.

A certified proof of SSH [14] was completed in CryptoVerif, though this proof is limited to the transport layer protocol, and to the secrecy and authenticity of the session key only. This security does not extend to the messages sent over the channel due to a vulnerability in SSH. CryptoVerif was also used to formally verify the Kerberos network authentication system [13].

Roy et al. [24] use Protocol Composition Logic to verify the security of Diffie-Hellman key exchange as used in Kerberos and IPSec key management. Both are standardized protocols, and the models and formal proofs are quite complex.

IX. Conclusion and Future Work

We have completed a mechanized proof of security for a complex efficient searchable symmetric encryption scheme, and shown that the effort required to complete such a proof is not prohibitive. In doing so, we have shown that complex proofs can be completed in FCF, and that the design of FCF supports decomposition into multiple reductions and sequences of games in order to manage the complexity of such proofs.

Future work includes the exploration of automation to reduce the effort required to develop proofs. A direct and powerful approach would be to use Ltac to develop custom automated tactics as suggested by [16]. EasyCrypt demonstrates the value of using SMT solvers to discharge low-level goals, and the same approach is possible in FCF by calling out to an SMT solver that produces proof witness [3]. Perhaps automation in the style of CryptoVerif [12] could also be used to replace entire equivalence proofs with a small amount of code describing the high-level argument and additional information such as loop invariants. An interesting challenge in implementing this level of automation is specifying the context where the argument should be applied, and automatically proving that the argument is valid in that context.

It would also be valuable to explore different sorts of cryptographic proofs and models in FCF. Because FCF is built on Coq, it is likely expressive enough to check any cryptographic proof, but perhaps certain elements of the proof must be modeled in unpleasant ways that make proof development difficult. For example, a non-black-box reduction may require an adversary to be modeled as a circuit or some other data structure. Another example is rewinding and coin-fixing arguments, which are likely supported in FCF without additional modeling due to the fact that adversary state is explicit, and FCF provides an operational semantics that gives control over coin flips.

Finally, FCF could be used to reason about the security of an *implementation* of a cryptographic scheme. Because FCF provides proofs in Coq, it can be combined with other Coq libraries for reasoning about code. For example, we could use FCF with the Verified Software Toolchain [2] to obtain a complete proof of security for the machine code implementing a cryptographic scheme.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from

high-level implementations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1217–1230, New York, NY, USA, 2013. ACM.

[2] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer Berlin Heidelberg, 2011.

[4] Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella-béguelin. Verified computational differential privacy with applications to smart metering. In *In 26th IEEE Computer Security Foundations Symposium, CSF 2013, Los Alamitos*, 2013.

[5] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*. ACM, 2014. To appear.

[6] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella Béguelin. Verified indifferentiable hashing into elliptic curves. In *Proceedings of the First International Conference on Principles of Security and Trust*, POST'12, pages 209–228, Berlin, Heidelberg, 2012. Springer-Verlag.

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

[8] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security verifiable IND-CCA security of OAEP. In Aggelos Kiayias, editor, *Topics in Cryptology CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer Berlin Heidelberg, 2011.

[9] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 97–110, New York, NY, USA, 2012. ACM.

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

[11] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459, May 2013.

[12] Bruno 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.

[13] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 87–99, Tokyo, Japan, March 2008. ACM.

[14] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):4–31, March 2013. Special issue ARES'12.

[15] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In Ran Canetti and Juan Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer Berlin Heidelberg, 2013.

[16] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[17] Emiliano De Cristofaro, Stanislaw Jarecki, Xiaomin Liu, Yanbin Lu, and Gene 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=32750071e5cf4afc3b7e973d608e657e, 2010.

[18] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

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

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

[21] Jonathan Herzog, Catherine Meadows, Aaron Jaggard, Alley Stoughton, and Jonathan Katz. MITLL-NRL panel: Easycrypt 0.2 feedback and opinions. http://web.archive.org/web/20140703170052/https://easycrypt.info/trac/wiki/SchoolUPen2013, 2013. Accessed: 2014-07-03.

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

[23] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In Riccardo Focardi and Andrew Myers, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015.

[24] Arnab Roy, Anupam Datta, and John Mitchell. Formal proofs of cryptographic security of diffie-hellman-based protocols. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 312–329. Springer Berlin Heidelberg, 2008.

[25] Alley Stoughton. Proving the security of the Mini-APP private information retrieval protocol in EasyCrypt. Dagstuhl Workshop on The Synergy Between Programming Languages and Cryptography,

2014. http://boemund.dagstuhl.de/mat/Files/14/14492/14492.StoughtonAlley.Slides.pdf.