Mechanizing the Proof of Adaptive, Information-theoretic Security of Cryptographic Protocols in the Random Oracle Model

Alley Stoughton
Boston University
alley.stoughton@icloud.com

Mayank Varia
Boston University
varia@bu.edu

***Abstract*—We report on our research on proving the security of multi-party cryptographic protocols using the EASYCRYPT proof assistant. We work in the computational model using the sequence of games approach, and define honest-but-curious (semi-honest) security using a variation of the real/ideal paradigm in which, for each protocol party, an adversary chooses protocol inputs in an attempt to distinguish the party's real and ideal games. Our proofs are information-theoretic, instead of being based on complexity theory and computational assumptions. We employ oracles (e.g., random oracles for hashing) whose encapsulated states depend on dynamically-made, nonprogrammable random choices. By limiting an adversary's oracle use, one may obtain concrete upper bounds on the distances between a party's real and ideal games that are expressed in terms of game parameters. Furthermore, our proofs work for adaptive adversaries, ones that, when choosing the value of a protocol input, may condition this choice on their current protocol view and oracle knowledge. We provide an analysis in EASYCRYPT of a three party private count retrieval protocol. We emphasize the lessons learned from completing this proof.**

I. INTRODUCTION

In this paper, we report on our research at mechanizing the security proofs of multi-party cryptographic protocols in the computational model. We limit ourselves to *honest-but-curious* (semi-honest) security: each party of the protocol follows the rules of the protocol, but may try to learn as much as it can from the information coming its way—i.e., from its *protocol view*. We define security using a variation of the simulation-based real/ideal paradigm [1], [2] in which, for each protocol party, an adversary chooses protocol inputs in an attempt to distinguish the party's real and ideal games. Intuitively, the adversary is trying to learn more from a party's view in the real game than it should be able to—i.e., more than it can learn from the view produced by the ideal game. If it cannot do this, we say that the protocol is *secure against* the protocol party. More formally, the real and ideal games for a protocol party return the boolean judgments made by the adversary, and a security theorem upper-bounds the absolute value of the difference between the probabilities that the real and ideal games return true.

There are several ways to bound the distance between the real and ideal games. One may work with upper bounds that explicitly involve the concrete adversaries constructed during a sequence of games proof [3]–[5]. For instance, part of the upper bound might be the advantage of a concrete adversary A in differentiating between the games defining security of a pseudorandom function F (the first game involves use of the application of F to a randomly generated key unknown to A, whereas the second game involves use of a uniformly random function). Another approach is to make use of complexity theoretic assumptions about the adversaries attempting to distinguish the real and ideal games, assuming, e.g., that they run in probabilistic polynomial time (PPT), in terms of a security parameter λ. One must then prove that constructed adversaries (like A, above) are also in the same complexity classes. E.g., this allows us to define and use in security theorems' upper bounds the advantage of a PPT adversary in differentiating between the pseudorandom function games—i.e., the least upper bound, across all PPT adversaries A, of the advantage of A in differentiating between the games. Finally, one may work with oracles (e.g., random oracles for hashing [6]) whose encapsulated states depend on dynamically-made, random choices, and to limit an adversary's oracle use. This *information-theoretic* approach allows one to upper-bound the distance between real and ideal games using bounds involving game parameters, like sizes of hash tags or limits on adversary oracle use. Combinations of the above approaches are also possible; e.g., one may both work with PPT adversaries and limit their usage of oracles.

An adversary's generation of protocol inputs may be done *adaptively* or *non-adaptively*. In the non-adaptive case, all the inputs are generated up front, before the protocol's execution begins, whereas in the adaptive case, when choosing the value of a protocol input, the adversary may condition this choice on its current protocol view and oracle knowledge. Non-adaptive protocols are both easier to define and prove secure, but adaptive protocols provide a more realistic abstraction of the behavior of adversaries in practice.

A. Our Contributions

In our work, we are developing and employing techniques for proving the adaptive, information-theoretic, honest-but-curious security of cryptographic protocols in the nonprogrammable random oracle model [6]. We formalize our proofs using EasyCrypt [7], [8], a framework for interactively finding security proofs for cryptographic constructions and protocols using the sequence of games approach [3]–[5]. In EasyCrypt's logics, one may specify that an adversary is "lossless," i.e., always terminating, but there is no more precise way of bounding its execution time, either asymptotically or concretely. On the other hand, limiting an adversary's oracle access is straightforward in EasyCrypt, making it fruitful to work information-theoretically.

In this paper, we present as a case study the security proof of a simple secure database protocol that we call PCR, for "Private Count Retrieval" (see Section III). This protocol raises many of the issues that will arise in more complex protected database search protocols [9], including:

- working with multiple protocol parties, each with their own security guarantees;
- expressing a protocol in such a way that it can be specialized to the real games for the different protocol parties;
- expressing ideal games parameterized by simulators, whose job is to construct parties' views given the limited information provided by the ideal games;
- coping with adaptive adversaries;
- working with oracles having encapsulated random state, and constraining oracle use by adversaries;
- calculating concrete upper bounds on the distance between pairs of cryptographic games;
- carrying out cryptographic reductions;
- reasoning up to failure (up to bad reasoning);
- working with complex relational invariants; and
- removing redundant hashing.

B. Paper Outline

The remainder of the paper is organized as follows. We begin by surveying the relevant literature (Section II). Next we define the PCR Protocol (Section III), say what it means for PCR to be secure (Section IV), and state the theorems expressing security against the protocol parties (Section V). This is followed by a brief survey of EasyCrypt (Section VI). Section VII describes the EasyCrypt formalization of the PCR Protocol, along with the definitions on which it is based. Sections VIII–X consider the EasyCrypt formalizations of the proofs of security against the Client, Server and Third Party. In Section XI, we summarize the results of our PCR case study, taking stock of what was learned. And in Section XII, we look forward to the next steps of our research program.

II. Related Work

Numerous cryptographic constructions and protocols have been proved secure using EasyCrypt, including OAEP [10], Merkle-Damgård [11], a core part of the TLS Handshake Protocol [12], RSA-PSS [13], one-round key exchange protocols [14] and padding-based encryption [15]. In contrast to these protocols, our PCR protocol involves three parties and multiple rounds of interaction.

CryptoVerif [16] is another tool for finding sequence of games proofs in the computational model. This highly automated tool attempts to synthesize intermediate games. CryptoVerif has been successfully applied to an aspect of SSH's Transport Layer Protocol [17], as well as to the Kerberos network authentication system [18]. But CryptoVerif's automated nature is a two-edged sword: it makes some proofs very easy, but complex proofs of multi-party cryptographic protocols are outside its scope.

Cryptographic algorithms and protocols may also be proved secure in the computation model using Petcher and Morrisett's Foundational Cryptography Framework (FCF) [19], which is shallowly embedded in the Coq proof assistant [20]. Petcher and Morrisett reported in [21] on using FCF to prove the security of a two-party protected database search protocol from [22]. In this protocol, databases are finite maps sending document indices (integers) to sets of keywords. A query q is a keyword, and is a request for the set of indices i such that q is an element of i's keyword set. In the protocol, the Client holds a database, but sends it to the Server in encrypted form (as what is called a TSet). When the Client wants the answer to a database query, it sends an encrypted form of the query (an stag) to the Server, which is able to return to the Client the query's answer, also in encrypted form. The goal of the protocol is for the Server to learn almost nothing about the database and queries through its interaction with the Client.

Petcher and Morrisett define the security of this protocol using the real/ideal paradigm, but they only prove security against the Server, because the Client owns the database and proposes the queries. They work with a non-adaptive adversary, one that proposes both the database and all queries, up front. And they employ pseudorandom functions, rather than working in the random oracle model. The upper bound of their security theorem involves the concrete adversaries constructed during their sequence of games proof. They note that, were they to tackle the adaptive version of their protocol, they would work in the random oracle model.

There is a large literature on protected database search protocols with non-mechanized proofs. We refer interested readers to the following surveys for more information: [9], [23].

III. PCR Protocol

In this section, we define the PCR protocol. It involves three parties: a *Server*, which holds a database, a *Client*, which makes queries about the database, and an untrusted *Third Party* (TP), which mediates between the Server and Client. A *database* is one-dimensional: it consists of a list of *elements* (which may be anything). Each *query* is also

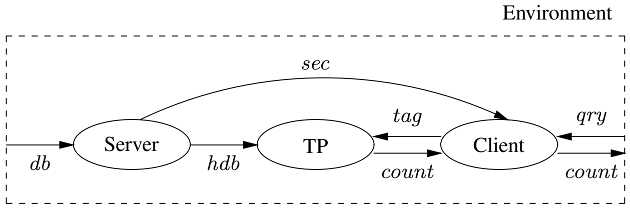

Figure 1. PCR Protocol Operation

an element, and is a request for the count of the number of times it occurs in the database.

We assume the parties do not collude with each other. The goal is for:

- the Client to learn only the counts for its queries, not anything else about the database;
- the Server to learn only the number of queries made by the Client, not which queries are made or what their counts are; and
- the TP to learn nothing about the database and queries other than certain element *patterns* (see below for what this means).

The Protocol's operation involves interaction with an Environment, as illustrated in Figure 1. Because we are working with honest-but-curious security, we find it more convenient for control flow to be driven by the Protocol, instead of by the Environment.

The Server randomly generates a secret, sec, and shares it with the Client, but not the TP. The Server requests a database from the Environment. If the request is refused, the Protocol terminates. Otherwise the Server randomly shuffles the database db, and turns the result into a hashed database hdb, which it sends to the TP. Each element, $elem$, of db is turned into the hash of $(elem, sec)$. Then the Client enters its query processing loop. At each iteration, it requests a query, qry, from the Environment. If the request is refused, the Protocol terminates. Otherwise the Client hashes (qry, sec), and asks the TP for the number of occurrences, $count$, of this hash tag, tag, in hdb. The Client then supplies $count$ to the Environment. Before the Protocol terminates, it asks the Environment what value the Protocol should return, and then returns this value as its overall result.

Secrets and hash tags are bit strings of length sec_len and tag_len, respectively, which are tunable parameters of the protocol. Hashing is done using a random oracle [6], consisting of a finite map to which new input/output pairs are added, dynamically. Element/secret pairs are mapped to hash tags, which are chosen uniformly randomly. The oracle's state is encapsulated: the Protocol (and Adversary and Simulator—see Section IV)) may only interact with it via the act of hashing. There is no way to check whether a given element/secret pair is already in the domain of the oracle's map; consequently, it is irrelevant to users of the oracle whether the pair has already been assigned a hash tag.

Normally, the set of all elements will be much bigger than the set of all hash tags; in fact, the former set will typically be infinite, whereas the latter has size $2^{\mathsf{tag\_len}}$. If a hash collision occurs, the Client's results may be inflated. E.g., if the database consists of distinct elements x and y, but (x, sec) and (y, sec) hash to the same hash tag, then the count for query x will be 2 not 1. But—assuming tag_len is large enough and the numbers of unique elements in the database and unique queries processed are small enough—hash collisions will be very unlikely. Thus, the Client should learn the correct counts of all the queries it processes.

Because the database is shuffled before being turned into a hashed database, and since hashing is not efficiently invertible and the TP will be unlikely to guess sec (assuming sec_len is big enough), the TP should only learn element *patterns*, not actual elements, from its interactions with the Server and Client. In particular, it will not learn anything about the order of the database (e.g., if the database begins with two occurrences of an element, the TP will not learn this fact).

IV. Definition of Security for PCR

We formalize security of the PCR protocol using the real/ideal paradigm. For each protocol party (Server, Client and Third Party), we have a pair of cryptographic games: a "real" game and an "ideal" game. The real games are based on the protocol as described above, where everything the party sees is recorded in its "view." The ideal game for a given party is designed so as to make it obvious that the party does not learn anything it should not, but where the party's view and random oracle state—as viewed through the lens of its hashing procedure—may still be simulated from the available information.

The real game for a given protocol party is parameterized by an Adversary with access to the random oracle. The Adversary plays the role of the Environment mentioned above when explaining the operation of the protocol. When the Adversary is called, the current value of the party's protocol view is passed to it. Upon the Protocol's final call to the Adversary, asking it for a final value to return, the Adversary returns a final boolean judgment, which the Protocol then returns as its final result. The Adversary is allowed to maintain state between the calls to it. When the Client processes a query provided by the Adversary, the Adversary is not informed of the result of the query, but is simply told that the query was processed (and what the party's view is at this point). Of course the Client's view does include the counts for all queries processed.

The ideal game for a protocol party is parameterized by both the party's Adversary and a Simulator. The job of the simulator is to try to make the Adversary think it is interacting with the real game: the Simulator constructs

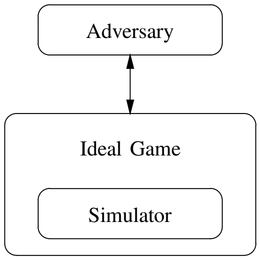

Figure 2. Structure of Ideal Games

the party's view and random oracle state given the limited information provided by the ideal game. This architecture is illustrated in Figure 2. We can usefully think of the Simulator as being inside the ideal game, which acts as an intermediary between it and the Adversary.

When we talk about what a given party *learns* about the database and queries supplied by the Adversary in the real or ideal game, we are referring to what the Adversary can learn from the values of the party's view that are passed back to it, as well as from its interactions with the random oracle. One may think of the party as being "woken up" upon each call of the Adversary. E.g., even after the Server has completed its construction of the hashed database, at each iteration of the Client's query processing loop, the Server is woken up, reminded of its current view, and allowed to interact with the random oracle. Consequently, the Server learns the number of queries that are processed by this loop.

Because we work information-theoretically, when assessing the information leakage of a party's ideal game, we do not have to scrutinize the party's Simulator (e.g., it cannot learn more about the database or queries by brute force computation). Consequently, the Simulator will be part of the *proof*—rather than the *specification*—of security.

A. Server's Ideal Game and Simulator

In the Server's ideal game, the Simulator must construct the Server's view without being given any information by the ideal game. For the real and ideal games to be indistinguishable, the ideal game must still have a Client loop in which the Adversary is allowed to propose a sequence of queries, which are ignored by the ideal game. Consequently, the Server learns nothing about the Client's queries, other than their number.

The Simulator for the Server that we use in our proof works as follows. When the ideal game tells it to initialize itself, it generates the secret, sec, and then initializes its view to record not only the generation of sec, but also its sharing with the Client (which happens in the real game, but not in the ideal one). The Simulator's processing of a database db proceeds just as in the real game, including the shuffling of the database, the hashing of database elements (paired with sec), and the updating of the view.

B. Third Party's Ideal Game and Simulator

In the TP's ideal game, the protocol operates normally—with the Simulator playing the role of the TP—except for a crucial difference:

> there is no shared Server/Client secret, and the Server and Client do their element hashing in a private random oracle, one the TP and Adversary do not have access to.

Because the database is shuffled before being turned into a hashed database, the TP learns nothing about the order of the Server's database, but does learn[1] the database's size. And it learns how many queries are processed by the Client. But otherwise it only learns element *patterns*. If no hash collisions occur in the private random oracle, it can tell how many distinct elements occur in the database, and how many times each one occurs in the database, as well as when the Client repeats a query, and how many times a query is in the database—all without knowing anything about the actual identities of the database elements and queries. But, because hash collisions in the private random oracle are possible—and become more probable as the numbers of unique database elements and distinct queries increase—there may be false positives. E.g., it may think the database has an element occurring 5 times, but it may actually have one element occurring 3 times, and another occurring 2 times.

The Simulator for the TP that we use in our proof behaves like the TP of the real game, recording in its view its receipt of the hashed database, hdb, as well as the result of its processing of each request from the ideal game for it to count the number of occurrence of a hash tag, tag, in hdb.

C. Client's Ideal Game and Simulator

In the Client's ideal game, the database (which does not need to be shuffled first) is turned into an "elements' counts" map detailing the number of times each element of the database occurs in the database. The Client's Simulator may ask the ideal game (passing the current Client view along with the request) for the next query along with its count. The ideal game responds to such a request by asking the Adversary for its next query. If the Adversary refuses (it will propose no more queries), this refusal is passed on to the Simulator. Otherwise, the proposed query is looked up in the elements' counts map, and the resulting count—or 0, if the query is not in the map's domain—is returned, along with the query, to the Simulator. Consequently, the Client learns nothing about the database other than the counts of the queries it proposes.

The Simulator for the Client that we use in our proof works as follows. When told by the ideal game to initialize

[1]When talking about what a party *can* learn in the ideal game, we are assuming the Simulator faithfully records what it learns in the party's view. Otherwise, the party may learn even less.

itself, it generates the secret, sec, and initializes its view to record that this secret was received. Its query processing loop requests a query, $elem$, and its count, $count$, from the ideal game, recording $elem$'s receipt or the refusal of the request in its view, and terminating if the request is refused. Otherwise, it hashes $(elem, sec)$, producing a hash tag, tag, and adds $(elem, tag, count)$ to the view.

D. Definitions of Security Against the Protocol Parties

The protocol is said to be *secure against* a given party iff the Adversary cannot distinguish the party's real and ideal games, i.e., the probabilities of the games returning true differ by a *negligible* amount. The idea is that, because the ideal protocol for a given party is secure by construction, and the Adversary is incapable of differentiating the games, the real protocol should also be considered secure. It is important to note that the Adversary must make its boolean judgments one protocol at a time—i.e., it is not given the results of runs of both protocols and allowed to try to tell them apart.

An Adversary's strategy for distinguishing a party's real and ideal games cannot in general be turned into a way for the party to learn more than it should be able to; for one thing, the party is not able to unilaterally choose both the database and sequence of queries (indeed the TP chooses neither). But any strategy for such over-learning should translate into an Adversarial strategy for distinguishing the real/ideal games, and so the lack of a viable Adversarial strategy will imply the lack of a way for the party to over-learn.

Because the Adversary chooses the database and queries, our definition of security against a given party is at least as strict as a definition saying that for all databases and sequences of queries, the results of running the real and ideal games are indistinguishable. But our Adversaries may adaptively condition their protocol input choices based on their interactions with the random oracle, and—in the Server and Client cases—on the dynamically generated Server/-Client secret, and so our security definition is strictly more restrictive than a universally quantified one. This models the fact that the Server and Client are capable of letting the shared Server/Client secret influence—intentionally or not—their choices of database and queries.

V. Security Theorems for PCR

In this section, we informally state the theorems expressing security against the Server, Third Party and Client. To obtain strong security against the TP and Client, we must limit the Adversary.

A. Security Against Server

For the Server, we are able to prove perfect security—i.e., that the real and ideal games are equally likely to return true. And we can do this without imposing any restrictions on the Adversary, not even that its procedures are lossless, i.e., always terminate.

The only challenging aspect of this proof is dealing with the hashing done in the Client's query processing loop of the real game (but not in the ideal game). The hash tags resulting from this hashing are only put in the Client's view, and so from the Server's perspective they are redundant. But at each iteration of the query processing loop, and also at protocol termination, the Adversary has black box access to the random oracle. Thus we must prove that the Adversary cannot tell whether the redundant hashing was actually done.

B. Security Against Third Party

For the Third Party, the Adversary can differentiate the real and ideal games with high probability[2] if it succeeds in guessing the Server/Client shared secret, sec, of the real game. More precisely, because it proposes a sequence $elem_1, \ldots, elem_n$ of queries to the games, it can work through the bit strings of length $\mathsf{sec\_len}$, looking for a secret sec' such that the hash tags produced by hashing the $(elem_i, sec')$ in the random oracle match the corresponding hash tags appearing in the TP's view. This will succeed in the real game, but will be unlikely to succeed in the ideal game (where the elements were hashed using the private random oracle on elements).

Consequently, to obtain strong security against the TP, we must impose some limit on the amount of hashing done by the Adversary. We have opted to impose a limit, limit, on the number of distinct inputs the Adversary may hash before being given a dummy result when new hashing inputs. Then we are able to upper-bound the absolute value of the difference in the probabilities of the real and ideal games returning true by

$$\mathsf{limit}/2^{\mathsf{sec\_len}},$$

a fairly tight upper bound on the probability that no more than limit random choices of bit strings of length $\mathsf{sec\_len}$ will successfully guess the Server/Client shared secret. The parameters limit and $\mathsf{sec\_len}$ may be tuned so as to make our upper bound arbitrarily small.

For a reason that will be made apparent in Section X, we must also assume the Adversary's procedures are always terminating, and we must limit the number of iterations of the Client's query processing loop to a constant $\mathsf{qrys\_max}$, ensuring termination of that loop. Consequently, the TP knows there will never be more queries than $\mathsf{qrys\_max}$.

C. Security Against Client

The Client receives the Server/Client shared secret, sec, at the beginning of the real game, just before the Adversary is asked to produce a database. Thus the shared secret is part of the Client view that is passed to the Adversary when

[2]We are assuming the TP's Simulator is the one described in Subsection IV-B, which is the one we use in our proof.

the Adversary is asked to produce a database. The Adversary can distinguish the real and ideal games[3] by finding a certain kind of hash collision, or arranging for the Server or Client in the real game to cause that kind of hash collision:

- Suppose it can find distinct elements $elem$ and $elem'$ such that $(elem, sec)$ and $(elem', sec)$ hash to the same hash tag, tag. Then it can choose $[elem]$ as the database, so that $[tag]$ becomes the Server's hashed database. And it can choose $elem'$ as the only query, which will have a count of 1 in the real game, but a count of 0 in the ideal game.
- It can choose a database consisting of a list of distinct elements whose size is more than $2^{\mathsf{tag\_len}}$, the number of distinct hash tags. This will force the hashed database produced by the real game's Server to have one or more duplicate elements, so that using the elements of the database as queries, one by one, will result in at least one query with a count of more than one in the real game, but a count of exactly one in the ideal game.
- If the Adversary chooses $[elem]$ as the database, it can choose n distinct elements other than $elem$ as queries. In the ideal game, all these queries will have counts of 0, but if n is big enough, the chance of the real game giving one of the queries a non-zero count will be non-negligible.

Consequently, we must not only impose a limit on the hashing done by the Adversary, but also limit both the number of distinct elements in a database chosen by the Adversary, and the number of distinct queries it may propose. For a reason that will be explained below, we will actually limit the number of times the Adversary may propose any query, making the Adversary avoid proposing duplicate queries if it does not want to incur the cost.

We impose a hashing "budget,"

$$\mathsf{budget} = \mathsf{adv\_budget} + \mathsf{db\_uniqs\_max} + \mathsf{qrys\_max},$$

on the Adversary, where budget is no more than the number $2^{\mathsf{tag\_len}}$ of distinct hash tags, and:

- adv_budget is the number of distinct elements the Adversary may hash itself without being considered "over budget" (except when asked to deliver its final boolean judgment, when it is not subject to budgeting as collisions caused at that point are harmless);
- db_uniqs_max is the maximum number of unique elements allowed in a database proposed by the Adversary; and
- qrys_max is a limit on the number of times the Adversary may ask to have a query processed.

If the Adversary exceeds its own hashing budget (adv_budget) before proposing its database, or if it proposes a database with more unique elements than db_uniqs_max, then the real and ideal games will skip the Client's query processing loop (which would be carried out by the Simulator, in the ideal game). And if, during the query processing loop, the Adversary (cumulatively) exceeds its own hashing budget, or if the Adversary asks more than qrys_max times to have a query processed, then the query processing loop will be terminated early (in the case of the ideal game, this is done by refusing the Simulator's request for another query/count pair).

Then we are able to upper-bound the absolute value of the difference in the probabilities of the real and ideal games returning true by

$$(\mathsf{budget} * (\mathsf{budget} - 1))/2^{\mathsf{tag\_len}}.$$

This is two times a fairly tight upper bound on the probability that no more than budget random choices of hash tags will result in a duplication. The reason for the factor of two will be explained in Section VIII. The parameters adv_budget, db_uniqs_max, qrys_max and tag_len may be tuned so as to make our upper bound arbitrarily small.

For a reason that will be made apparent in Section VIII, we must also assume the Adversary's procedure's are always terminating. We also need that the Client's query processing loop always terminates, but this is guaranteed by our use of qrys_max. (If we had only counted unique queries toward the qrys_max part of the hashing budget, termination would not have been ensured.)

Note that some of hashing done by the Adversary, the Server (in the real game), and the Client (in the real game) or Client's Simulator (in the ideal game) may overlap (i.e., an element/secret pair may be queried that is already in the oracle's map). Furthermore, in the ideal game, the db_uniqs_max part of the hashing budget is unused—in a sense wasted. But by keeping the different parts of the budget separate, we ensure the Adversary uses its budget at the same rate in both games, as well as that the Client's query loop runs the same number of times in both games. Because databases with more than db_uniqs_max unique elements are rejected, the Client knows that the database has no more than db_uniqs_max elements.

VI. Introduction to EasyCrypt

In EasyCrypt, cryptographic games (probabilistic programs) are modeled as *modules*, which consist of procedures and global variables. Procedures are written in a simple imperative language featuring while loops and random assignments.

EasyCrypt has four logics: a probabilistic, relational Hoare logic, relating pairs of procedures; a probabilistic Hoare logic allowing one to prove facts about the probability of a procedure's execution resulting in a postcondition holding; an ordinary Hoare logic; and an ambient higher-order logic for proving general mathematical facts, as well as for connecting judgments from the other logics. For instance,

[3] We are assuming the Client's Simulator is the one described in Subsection IV-C, which is the one we use in our proof.

one may use the probabilistic, relational Hoare logic to prove an equivalence between the boolean-returning main procedures of two modules whose postcondition says the procedures' results are equal, and then use the ambient logic to prove that the two procedures are equally likely to return true. One may prove facts involving abstract modules, e.g., ones representing adversaries.

Proofs are carried out using *tactics*, which transform the current proof goal into zero or more subgoals. Simple ambient logic goals may be automatically proved using SMT solvers. Once found, an EASYCRYPT proof script can be replayed step-by-step, or checked in batch-mode. Proofs may be structured as sequences of lemmas, and EASYCRYPT's *theories* may be used to group definitions, modules and lemmas together. Theories may be specialized using a process called cloning. Abstract theories must be cloned before they can be used. Requiring (**require**) a theory makes it available for use; but it must also be imported (**import**) for its definitions and lemmas to be usable without being qualified by the theory name.

EASYCRYPT has a fairly small trusted computing base (TCB). Its core proof engine is comprised of about 5,000 lines of OCaml code, implementing well-studied logics proven correct [24] using the Coq proof assistant [20]. Almost all of EASYCRYPT's library of mathematical and cryptographic theories is outside the TCB. When solving goals using SMT solvers, one may specify the list of previously proven EASYCRYPT lemmas the solvers may use.

The remainder of this section details the EASYCRYPT definitions used in the rest of the paper. EASYCRYPT provides the types bool, int and real with the expected constants and operations. If exp denotes a natural number, then exp%r denotes the corresponding real number. The unit type, unit, has only one element, (). EASYCRYPT has tuple (product) types written with $*$ and function types written with $\rightarrow$, so that, e.g., int $*$ bool $\rightarrow$ int $\rightarrow$ real is the type of functions from integer/boolean pairs to functions from integers to reals.

EASYCRYPT provides an option type, 'a option, where 'a is a type variable, which may be instantiated with any type. This type is defined as a concrete datatype:

```
type 'a option = [None | Some of 'a].
```

None and Some are its constructors, and its values are None and the results of applying Some to the values of type 'a. The operator oget : 'a option $\rightarrow$ 'a transforms an input of the form Some x to x; when given None, it returns an unknown, but fixed, value of type 'a. Types in EASYCRYPT are always nonempty.

EASYCRYPT provides list types, 'a list. E.g., [0; 1; 2] is the int list consisting of the first three natural numbers. ++ is list concatenation. The operator size : 'a list $\rightarrow$ int computes the number of elements in a list. The operator nth : 'a $\rightarrow$ 'a list $\rightarrow$ int $\rightarrow$ 'a option selects the ith element of a list (counting from 0); it returns the first (default) argument when i is out of range. The operator trim : 'a list $\rightarrow$ int $\rightarrow$ 'a list deletes the ith element of a list (leaving the list as is if i is out of range).

EASYCRYPT provides finite set types, 'a fset. There are the expected operations on finite sets, including mem : 'a fset $\rightarrow$ 'a $\rightarrow$ bool: mem $xs\,y$ tests whether y is an element of xs.

EASYCRYPT provides finite map types, ('a, 'b) fmap. E.g., (int, bool) fmap is the type of finite maps from integers to booleans. map0 is the empty map. To look up the value of an element x in a map m whose range has type t, one uses the notation $m.[x]$, which results in a value of type t option, giving None when x is not in m's domain. To update a map m so that it sends x to y but is otherwise unchanged, one uses the notation $m.[x \rightarrow y]$. The operators dom and rng transform a map to its domain and range (finite sets).

EASYCRYPT provides a type 'a distr of probability distributions of type 'a. A distribution is *lossless* iff the sum of the weights of all element of its support is 1%r. A distribution is *uniform* iff every element of the type has an equal weight in the distribution. E.g., if $n \leq m$, then $[n\,..\,m]$ is the uniform and lossless distribution on the set of all integers that are at least n and no more than m.

In EASYCRYPT's programming language, ordinary variable assignments are written with $\leftarrow$ and procedure call assignments are written with <@:

```
x ← x + 1;
x <@ M.f(x * 2);
```

There is a shorthand notation for updating maps via assignments:

```
mp ← mp.[x ← y];
```

may be abbreviated to

```
mp.[x] ← y;
```

If d is a distribution, then

```
x <$ d;
```

means to assign to x a value from d, respecting the weights of elements in d. Choosing a value from a distribution that is not lossless may fail, terminating the program.

VII. EASYCRYPT FORMALIZATION OF PCR PROTOCOL

In this section, we present the formalization in EASYCRYPT of the PCR Protocol. We also give the definitions supporting this formalization and the statements of security against the protocol's parties.

A. Supporting Definitions

The operator num_uniqs : 'a list → int returns the number of unique elements in a list. We have a type elem of elements—a tunable parameter to our games, which may be instantiated with any type. elem_default : elem is some element. We have a type sec of secrets whose elements are bit strings of length sec_len—a tunable parameter to our games. The uniform and lossless probability distribution on secrets is called sec_distr. And we have a type tag of hash tags whose elements are bit strings of length tag_len—a tunable parameter to our games. The tag consisting of all zeros is called zeros_tag. The uniform and lossless probability distribution on tags is called tag_distr.

The type elems_counts consists of finite "elements' counts" maps from elements to integers (thought of as counts):

```
type elems_counts = (elem, int) fmap.
op empty_ec : elems_counts = map0.
op get_count (cnts : elems_counts) (elem : elem) : int =
  if mem (dom cnts) elem then oget cnts.[elem] else 0.
op incr_count (cnts : elems_counts) (elem : elem) : elems_counts =
  if mem (dom cnts) elem
  then cnts.[elem ← oget(cnts.[elem]) + 1]
  else cnts.[elem ← 1].
```

Thus: empty_ec is the empty elements' counts map; get_count is a function for looking up an element's count in an elements' counts map, getting 0 when the element is not in the map; and incr_count increments an element's count in an elements' counts map, setting its value to 1 when it was not already in the map.

We have a module with a procedure for randomly shuffling lists:

```
module Shuffle = {
  proc shuffle(xs : elem list) : elem list = {
    var ys : elem list; var i : int;
    ys ← [];
    while (0 < size xs) {
      i <$ [0 .. size xs − 1]; (* pick random index into xs *)
      ys ← ys ++ [nth elem_default xs i];
      xs ← trim xs i;
    }
    return ys;
  }
}.
```

B. Random Oracles

We provide an abstract theory RandomOracle defining random oracles. To use an abstract theory, one must first *clone* it, instantiating (some) of its types, operators and predicates in the process, and yielding a (non-abstract, and so usable) theory. RandomOracle is parameterized by: a type input; an operator (constant) output_len : int that is constrained to be a natural number; a type output with exactly $2^{\mathsf{output\_len}}$ elements; an operator output_default : output; and an operator output_distr : output distr that is the uniform and lossless distribution on output. RandomOracle defines a module type (interface) OR:

```
module type OR = {
  proc init() : unit
  proc hash(inp : input) : output }.
```

An implementation of OR provides procedures init and hash, with the specified types, and the standard implementation is

```
module Or : OR = {
  var mp : (input, output) fmap

  proc init() : unit = { mp ← map0; }

  proc hash(inp : input) : output = {
    if (! mem (dom mp) inp) {
      mp.[inp] <$ output_distr;
    }
    return oget mp.[inp];
  }
}.
```

Or has a global variable mp, consisting of a finite map from values of type input to values of type output. The procedure init initializes mp to the empty map. The procedure hash first tests whether its input inp is not in mp's domain. If the answer is "no," it simply returns inp's value in mp. Otherwise, it updates mp, associating with inp a random value of type output, and then returns that random value.

RandomOracle also defines two wrappers for random oracles, each in its own abstract theory. The Limited abstract theory is parameterized by limit : int, which is constrained to be a natural number. It implements the *limited random oracle*, which is parameterized by an implementation O of OR, and has the form:

```
module LOr(O : OR) : OR = { ... }.
```

Its hash procedure uses O to do hashing, but keeps track of the inputs it has previously hashed. When the set of previously hashed inputs reaches size limit, it continues to use O to hash elements of the set, but returns output_default on fresh inputs (without changing the set or calling O). Its init function does not call O.init.

The Counted abstract theory is parameterized by budget : int, which is constrained to be a natural number. It provides a new module type of *counted random oracles*:

```
module type COR = {
  proc init() : unit
  proc chash(inp : input) : output
  proc hash(inp : input) : output
  proc within_budget() : bool }.
```

Here chash stands for "counted" hashing, whereas hash stands for ordinary hashing. Its implementation has the form:

```
module COr(O : OR) : COR = { ... }.
```

The procedure hash simply calls O.hash. The procedure chash keeps track of the elements it has seen while within budget, only counting inputs not previously seen toward the budget. It also notes when it goes over budget, i.e., a new input was presented when the budget was already exhausted. But unlike LOr's hash, even when it is over budget, it keeps using O to do hashing. The within_budget procedure tests whether the oracle is within its budget.

For use in the PCR Protocol definition, and in the Server, Third Party and Client proofs, we clone RandomOracle, making substitutions for the parameters of the abstract theory, proving that the substitutions have the required properties (the "realization" part), and calling the resulting theory RO:

```
clone RandomOracle as RO with
  type input ← elem * sec, op output_len ← tag_len,
  type output ← tag, op output_default ← zeros_tag,
  op output_distr ← tag_distr
proof *. (* realization *) ··· (* end *)
```

Now RO.Or is our random oracle. It hashes element/secret pairs to hash tags, and its limited random oracle wrapper returns the all zeros tag when a fresh input is hashed but the hashing limit was already reached.

C. Protocol Definition

We define the types of databases and hashed databases:

```
type db = elem list. type hdb = tag list.
```

The protocol views for the three parties have types

```
type server_view = server_view_elem list.
type tp_view = tp_view_elem list.
type client_view = client_view_elem list.
```

where the elements of server_view_elem, tp_view_elem and client_view_elem document events "seen" by the parties—e.g., that the Server received the database, or that it shuffled the database.

The PCR Protocol is defined as a module parameterized by an Environment Env with module type:

```
module type ENV = {
  proc * init_and_get_db() : db option
  proc get_qry() : elem option
  proc put_qry_count(cnt : int) : unit
  proc final() : bool }.
```

The procedure init_and_get_db initializes the Environment (that is what the asterisk mandates), and tries to get a database from it; None means refusal. The procedure get_qry tries to get a query from the Environment; None means the Environment has refused—by convention, it is done providing queries. The procedure put_qry_count tells the Environment the count corresponding to last query processed. And the procedure final finalizes the environment, and returns the Environment's boolean judgment.

At the top-level, Protocol looks like

```
module Protocol (Env : ENV) = {
  module Or = RO.Or
  var sv : server_view   var tpv : tp_view   var cv : client_view
  var server_sec : sec   var server_hdb : hdb
  var tp_hdb : hdb   var client_sec : sec   ···

  proc main() : bool = {
    var db_opt : db option; var b : bool;
    init_views(); Or.init(); server_gen_sec(); client_receive_sec();
    db_opt <@ Env.init_and_get_db();
    if (db_opt ≠ None) {
      server_hash_db(oget db_opt);
      tp_receive_hdb();
      client_loop();
    }
    b <@ Env.final();
    return b;
  }
}.
```

The module has an abbreviation for the random oracle, as well as global variables for: the three parties' views; the secret generated by the Server and the Client's copy of it; and the hashed database produced by the Server and the TP's copy of it. The main procedure initializes all three views to be empty lists, initializes the random oracle, asks the Server to generate the secret (storing it in server_sec and updating its view), asks the Client to receive that secret (storing it in client_sec and updating its view; it gets the secret by asking the Server for it, which updates the Server's view), and then asks the Environment to initialize itself and produce a database. If the Environment complies, the database is passed to the Server, which shuffles it (using Shuffle.shuffle), and turns it into a hashed database, server_hdb, all the while updating its view. Back in main, the TP receives the hashed database (storing it in tp_hdb, and updating its view; it obtains it by asking the Server for it), and then the Client query processing loop runs. After that loop terminates, the Environment is asked for a final boolean judgment, which main returns as its result. If the Environment refuses to produce a database, main skips to the finalization step.

The Client's query processing loop is:

```
proc client_loop() : unit = {
  var cnt : int; var tag : tag; var qry_opt : elem option;
  var not_done : bool ← true;
  while (not_done) {
    qry_opt <@ Env.get_qry();
    cv ← cv ++ [cv_got_qry qry_opt];
    if (qry_opt = None) { not_done ← false; }
    else {
      tag <@ Or.hash((oget qry_opt, client_sec));
      cnt <@ tp_count_tag(tag);
      cv ← cv ++ [cv_query_count(oget qry_opt, tag, cnt)];
      Env.put_qry_count(cnt);
    }
  }
}
```

The code should be self-explanatory, and it is worth comparing it with the description of the query processing loop from Section III. Note how the Environment is asked for a query, and informed of the query's count. Also note how the Client's view, cv, is updated. The TP's tp_count_tag procedure is what you would expect: it simply counts the number of times its argument hash tag appears in its copy of the hashed database, tp_hdb, returning that count, and updating its view.

VIII. Proof of Security Against Client

In this section, we consider the proof of security against the Client. As explained in Subsection V-C, the Client's Adversary is subjected to a hashing budget with three parts: the hashing it can do directly (adv_budget), the hashing it can make the Server do (db_uniqs_max), and the hashing it can make the Client do (qrys_max):

```
op adv_budget : int.
axiom adv_budget_ge0 : 0 ≤ adv_budget.
op db_uniqs_max : int.
axiom db_uniqs_max_ge0 : 0 ≤ db_uniqs_max.
op qrys_max : int.
axiom qrys_max_ge0 : 0 ≤ qrys_max.
op budget : int = adv_budget + db_uniqs_max + qrys_max.
axiom budget_ub : budget ≤ 2 ^ tag_len.

clone RO.Counted as CRO with op budget ← adv_budget
proof *. (* realization *) · · · (* end *)
```

The counted random oracle abstract theory (see Subsection VII-B) is cloned with adv_budget as its budget, with the resulting theory being called CRO. It uses RO.Or to do its hashing.

The Adversary's module type for the Client is defined by:

```
module type ADV(O : CRO.COR) = {
  proc * init_and_get_db(cv : client_view) : db option {O.chash}
  proc get_qry(cv : client_view) : elem option {O.chash}
  proc qry_done(cv : client_view) : unit {O.chash}
  proc final(cv : client_view) : bool {O.hash} }.
```

This should look similar to the Environment (Env) module type of Subsection VII-C, but there are important differences. First, an Adversary is parameterized by a counted random oracle, O. The annotations in set braces at the end of the procedure specifications say that the Adversary's first three procedures may only do counted hashing (O.chash), whereas its final procedure may only do ordinary hashing (O.hash). Second, put_qry_count has been replaced by qry_done, which simply tells the Adversary that the processing of the most recent query has finished. Third, all procedures pass the Client's view, and nothing more, to the Adversary.

The Client's real game, GReal, is listed in Figure 3. First, three module abbreviations are given: Or is the random oracle, COr is the counted random oracle derived from Or, and A is the resulting of connecting the Adversary to COr (so the Adversary's calls to the procedures of its parameter O will be translated into calls to COr's procedures). It then declares an Environment, Env, whose procedures call the corresponding procedures of A, passing them the current Client view. Finally, GReal's main procedure simply calls Protocol(Env)'s main procedure (so Protocol's calls to its argument's procedures go to those of Env), returning what it returns.

```
module GReal(Adv : ADV) : GAME = {
  module Or = RO.Or   module COr = CRO.COr(Or)
  module A = Adv(COr)

  module Env : ENV = {
    var qrys_ctr : int

    proc init_and_get_db() : db option = {
      var db_opt : db option; var adv_within_budg : bool;
      qrys_ctr ← 0; COr.init();
      db_opt <@ A.init_and_get_db(Protocol.cv);
      if (db_opt ≠ None) {
        adv_within_budg <@ COr.within_budget();
        if (db_uniqs_max < num_uniqs(oget db_opt) ∨
            !adv_within_budg) { db_opt ← None; }
      }
      return db_opt;
    }

    proc get_qry() : elem option = {
      var qry_opt : elem option; var adv_within_budg : bool;
      qry_opt <@ A.get_qry(Protocol.cv);
      if (qry_opt ≠ None) {
        adv_within_budg <@ COr.within_budget();
        if (qrys_ctr < qrys_max ∧ adv_within_budg) {
          qrys_ctr ← qrys_ctr + 1;
        }
        else { qry_opt ← None; }
      }
      return qry_opt;
    }

    proc put_qry_count(cnt : int) : unit = { A.qry_done(Protocol.cv); }

    proc final() : bool = {
      var b : bool; b <@ A.final(Protocol.cv); return b;
    }
  }

  proc main() : bool = {
    var b : bool; b <@ Protocol(Env).main(); return b;
  }
}.
```

Figure 3. Client's Real Game

Env has a global variable qrys_ctr that keeps track of the number of queries that have been processed. The procedure init_and_get_db initializes qrys_ctr and COr. (Protocol initializes Or.) Note how it returns None if the Adversary proposes a database with too many distinct elements, or if it exceeds its hashing budget. The procedure get_qry increments qrys_ctr each time a query is processed. Note how it returns None when the query processing limit has been exceeded or the Adversary has exceeded its budget.

The Client's ideal game is parameterized by a Simulator that keeps track of the Client's view, and communicates with the ideal game via the following interface:

```
module type SIG = {
  proc get_qry_count(cv : client_view) : (elem * int) option
  proc qry_done(cv : client_view) : unit }.
```

SIG stands for "Simulator's interface to Ideal Game". The Simulator calls get_qry_count to request the next query along with its count. And it calls qry_done to tell the ideal game it is done processing the most recently received query. The Simulator itself has the following interface:

```
module type SIM(O : RO.OR, SIG : SIG) = {
  proc * init() : unit { }
  proc get_view() : client_view { }
  proc client_loop() : unit {O.hash SIG.get_qry_count SIG.qry_done} }.
```

It is parameterized by both the random oracle O and the interface SIG to the ideal game. It has procedures for initialization and obtaining the current view—neither of which are allowed to access either O or SIG. But its client_loop procedure has access to both O and SIG. We do not need to limit the Simulator's access to O—doing more hashing will not help it learn more about the database.

The Client's ideal game, GIdeal, is listed in Figure 4. It is parameterized by both the Adversary Adv and Simulator Sim. It has a procedure count_db for turning the database into an elements' counts map, stored in the global variable db_elems_cnts. Its submodule SIG implements the Simulator's interface to the ideal game, and S is an abbreviation for the connection of Sim to the random oracle and SIG. The main procedure of GIdeal initializes the queries counter, Simulator, random oracle and counted random oracle, before asking the Adversary for a database, passing it the view provided the the Simulator. As in the real game, if the Adversary refuses to provide a database, or proposes a database with too many distinct elements, or exceeds its budget, the game proceeds on to calling the Adversary's final procedure. Otherwise it first uses count_db to turn the database into the elements' counts map, and then invokes the Simulator's Client loop. The get_qry_count procedure of SIG is much like the procedure get_qry of the submodule Env of the real game. But instead of returning (Some of) a query, it returns the query along with its count in the elements' counts map (0, if it is not in the map's domain).

The lemma expressing security against the Client is:

```
lemma GReal_GIdeal :
  exists (Sim <: SIM{GReal, GIdeal}),
  forall (Adv <: ADV{GReal, GIdeal, Sim}) &m,
  (forall (O <: CRO.COR{Adv}),
   islossless O.chash ⇒ islossless Adv(O).init_and_get_db) ⇒
  (forall (O <: CRO.COR{Adv}),
   islossless O.chash ⇒ islossless Adv(O).get_qry) ⇒
  (forall (O <: CRO.COR{Adv}),
   islossless O.chash ⇒ islossless Adv(O).qry_done) ⇒
  (forall (O <: CRO.COR{Adv}),
   islossless O.hash ⇒ islossless Adv(O).final) ⇒
  `|Pr[GReal(Adv).main() @ &m : res] −
   Pr[GIdeal(Adv, Sim).main() @ &m : res]| ≤
  (budget * (budget − 1))%r / (2 ^ tag_len)%r.
```

It is existentially quantified by a Simulator Sim, and the restriction on SIM restricts Sim to be a module that cannot interact with GReal or GIdeal either directly or indirectly

```
module GIdeal (Adv : ADV, Sim : SIM) : GAME = {
  module Or = RO.Or    module COr = CRO.COr(Or)
  module A = Adv(COr)

  var db_elems_cnts : elems_counts    var qrys_ctr : int

  proc count_db(db : db) : unit = {
    var i : int; var elem : elem;
    db_elems_cnts ← empty_ec; i ← 0;
    while (i < size db) {
      elem ← nth elem_default db i;
      db_elems_cnts ← incr_count db_elems_cnts elem;
      i ← i + 1;
    }
  }

  module SIG : SIG = {
    proc get_qry_count(cv : client_view) : (elem * int) option = {
      var qry_opt : elem option;
      var qry_cnt_opt : (elem * int) option;
      var adv_within_budg : bool; var cnt : int;
      qry_opt <@ A.get_qry(cv);
      if (qry_opt = None) { qry_cnt_opt ← None; }
      else {
        adv_within_budg <@ COr.within_budget();
        if (qrys_ctr < qrys_max ∧ adv_within_budg) {
          qrys_ctr ← qrys_ctr + 1;
          cnt ← get_count db_elems_cnts (oget qry_opt);
          qry_cnt_opt ← Some (oget qry_opt, cnt);
        }
        else { qry_cnt_opt ← None; }
      }
      return qry_cnt_opt;
    }

    proc qry_done(cv : client_view) : unit = { A.qry_done(cv); }
  }

  module S = Sim(Or, SIG)

  proc main() : bool = {
    var db_opt : db option; var b : bool; var adv_within_budg : bool;
    var cv : client_view;
    qrys_ctr ← 0; S.init(); Or.init(); COr.init();
    cv <@ S.get_view(); db_opt <@ A.init_and_get_db(cv);
    if (db_opt ≠ None) {
      adv_within_budg <@ COr.within_budget();
      if (num_uniqs(oget db_opt) ≤ db_uniqs_max ∧
          adv_within_budg) {
        count_db(oget db_opt); S.client_loop();
      }
    }
    cv <@ S.get_view(); b <@ A.final(cv); return b;
  }
}.
```

Figure 4. Client's Ideal Game

(except, of course, through its arguments O and SIG). After the existential quantifier comes the universal quantification over all Adversaries Adv not interacting with GReal, GIdeal and Sim, and all initial memories &m. The rest of the lemma is conditioned on the procedures of Adv being lossless (always terminating). In the conclusion,

```
Pr[GReal(Adv).main() @ &m : res]
```

is the probability of GReal(Adv).main returning true when

```
module (Sim : SIM) (O : RO.OR, SIG : SIG) = {
  var cv : client_view   var sec : sec

  proc init() : unit = { sec <$ sec_distr; cv ← [cv_got_sec sec]; }

  proc get_view() : client_view = { return cv; }

  proc client_loop() : unit = {
    var tag : tag; var qry : elem; var cnt : int;
    var qry_cnt_opt : (elem * int) option;
    var not_done : bool ← true;
    while (not_done) {
      qry_cnt_opt <@ SIG.get_qry_count(cv);
      if (qry_cnt_opt = None) {
        not_done ← false; cv ← cv ++ [cv_got_qry None];
      }
      else {
        (qry, cnt) ← oget qry_cnt_opt;
        cv ← cv ++ [cv_got_qry (Some qry)];
        tag <@ O.hash((qry, sec));
        cv ← cv ++ [cv_query_count(qry, tag, cnt)];
        SIG.qry_done(cv);
      }
    }
  }
}.
```

Figure 5. Client's Simulator

started with memory &m, and

```
Pr[GIdeal(Adv, Sim).main() @ &m : res]
```

is the probability of GIdeal(Adv, Sim).main returning true when started with &m. As expected, the upper-bound on the distance between these two probabilities is expressed in terms of budget and tag_len.

When proving GReal_GIdeal, we implement the Simulator by the module given in Figure 5. Its initialization procedure generates the secret, in contrast to in the real game, where the Server is responsible for doing this.

The Client proof uses a BudgetedRandomOracle abstract subtheory of RandomOracle providing *budgeted random oracles*, which implement the interface

```
module type BOR = {
  proc init() : unit
  proc adv_bhash(inp : input) : output
  proc adv_within_budget() : bool
  proc server_bhash(inp : input) : output
  proc client_bhash(inp : input) : output
  proc hash(inp : input) : output }.
```

providing ordinary hashing, plus budgeted hashing procedures for the adversary, server and client, each subject to their own parts adv_budget, serv_budget and clnt_budget of a total hashing budget, budget, that is no more than the number of elements in output. adv_bhash and server_bhash only debit their parts of the budget when called with unseen inputs, but each call to client_bhash counts toward its part of the budget.

There are two implementations of this interface: a first, BOr, in which hashing collisions may occur, as usual; and a second, BOrInj, in which, as long as hash is not called and all three parts of the hashing budget are (individually) within budget, the oracle's map stays injective, i.e., collision-free. We work with switching adversaries

```
module type SWADV(O : BOR) = {
  proc * main() : bool {O.adv_bhash O.adv_within_budget
                        O.server_bhash O.client_bhash O.hash} }.
```

and have a game GSwitching(SWAdv, O) that takes in a switching adversary SWAdv and a budgeted random oracle O, and whose main function initializes O, and then returns SWAdv's boolean judgment on O. Our version of the usual switching lemma bounds the distance between games involving BOr and BOrInj. It is proved using reasoning up to failure, which requires the losslessness of the switching adversary. EASYCRYPT's failure event lemma is used to upper-bound the possibility of failure with

$$(\mathsf{budget} * (\mathsf{budget} - 1))/2^{\mathsf{output\_len}+1},$$

a fairly tight upper bound on the probability that no more than budget random choices of output values will result in a duplication. In the Client proof, we clone BudgetedRandomOracle, substituting adv_budget for itself, db_uniqs_max for serv_budget, and qrys_max for clnt_budget. (When we originally cloned RandomOracle, we handled the substitutions for input and output, etc.)

In our sequence of games, we transition from the real game (with the Environment inlined, and simplifications made), in which the Server and Client use the random oracle Or but the Adversary uses the counted random oracle COr derived from Or, to a game using the collision-possible budgeted random oracle, BOr. Then we transition to using the collision-free-while-under-budget budgeted random oracle, BOrInj, incurring the above upper bound (with tag_len substituted for output_len) as a penalty. In more detail, we define a concrete switching adversary SWAdv in such a way that the Client's game involving BOr can be connected with GSwitching(SWAdv, BOr), and GSwitching(SWAdv, BOrInj) can be connected with the game involving BOrInj. The requirement that the switching adversary be lossless explains why the security theorem for the Client requires the losslessness of the Adversary's procedures, and why the number of queries proposed by the Adversary must be limited. This is how reductions are carried out in EASYCRYPT.

This sets the stage for the hardest part of the Client proof, which involves switching from the Server and Third Party using a hashed database, to them using an elements' counts map produced by the Server from the database (whose elements the Server still hashes), and shared with the TP (which now accepts requests from the Client for queries to be looked up in its map). This step uses a complex relational invariant involving the secret, hashed database (in the first game), random oracle's map, and elements' counts map (in the second game). Knowing that the random oracle stays injective (subject to the budget being respected) made this step much easier.

After that, we transition back to the collision-possible budgeted random oracle, BOr, incurring the above penalty a second time, and then to a game in which the Server and Client use Or, but the Adversary uses COr.

At this point, the Server's hashing is seen to be redundant: the elements of the database are still hashed (paired with the secret), but nothing is done with the resulting hash tags. Happily, Grégoire [25] recently invented a general technique for removing redundant hashing, which we have adapted and reimplemented. In a RedundantHashing abstract subtheory of RandomOracle we have module types

```
module type HASHING = {
  proc init() : unit
  proc hash(inp : input) : output
  proc rhash(inp : input) : unit }.

module type HASHING_ADV(H : HASHING) = {
  proc * main() : bool {H.hash H.rhash} }.
```

We have two implementations of HASHING, both built from a random oracle O: NonOptHashing ("non optimized hashing"), in which rhash ("r" for redundant) hashes its input, but discards the result; and OptHashing ("optimized hashing"), where rhash does nothing. In both cases, hash works normally. Then we have the following games

```
module GNonOptHashing(HashAdv : HASHING_ADV) = {
  module H = NonOptHashing(Or)
  module HA = HashAdv(H)
  proc main() : bool = {
    var b : bool; Or.init(); b <@ HA.main(); return b;
  }
}.
module GOptHashing(HashAdv : HASHING_ADV) = {
  module H = OptHashing(Or)
  module HA = HashAdv(H)
  proc main() : bool = {
    var b : bool; Or.init(); b <@ HA.main(); return b;
  }
}.
```

and a lemma saying one may move from the first game to the second:

```
lemma GNonOptHashing_GOptHashing
      (HashAdv <: HASHING_ADV{Or}) &m :
  Pr[GNonOptHashing(HashAdv).main() @ &m : res] =
  Pr[GOptHashing(HashAdv).main() @ &m : res].
```

Proving the lemma involves advanced use of EASYCRYPT's eager tactics, but the intuition behind the proof is simple: redundant hashing can be postponed until the point where ordinary hashing makes it non-redundant, or when the end of the game is reached and there is no point in doing it.

Finally, all that separates us from the ideal game (apart from some inlining and bookkeeping) is that the Server is still randomly shuffling the database before producing the elements' counts map. But we can use the EASYCRYPT Library's loop iteration abstract theory to show that computing the elements' counts map is independent from the database's order.

IX. PROOF OF SECURITY AGAINST SERVER

In this section, we consider the proof of security against the Server. The Adversary's module type for the Server is defined by:

```
module type ADV(O : RO.OR) = {
  proc * init_and_get_db(sv : server_view) : db option {O.hash}
  proc get_qry(sv : server_view) : elem option {O.hash}
  proc qry_done(sv : server_view) : unit {O.hash}
  proc final(sv : server_view) : bool {O.hash} }.
```

Note that the Adversary is parameterized by an ordinary random oracle.

The Server's real game, GReal, is much simpler than that of the Client, because the Server's Adversary does not have to be limited in any way.

The Server's Simulator has this interface:

```
module type SIM(O : RO.OR) = {
  proc * init() : unit { }
  proc get_view() : server_view { }
  proc main(db : db) : unit {O.hash} }.
```

Its initialization procedure generates the secret and initializes its view to reflect not just the generation of the secret but also its sharing with the Client (which happens in the real game, and so must be simulated in the ideal game). Its main procedure takes in the database and constructs the Server's view, which involves shuffling the database and hashing its elements (paired with the secret).

The Server's ideal game, GIdeal, is parameterized by the Adversary Adv and Simulator Sim. Its main procedure initializes Sim and the random oracle, before asking the Adversary to propose its database. If the Adversary obliges, main runs Sim's main procedure on that database, and then executes the version of the Client's query loop in which the Adversary's queries are ignored. In any event, main finishes by finalizing the Adversary and returning its boolean judgment.

The lemma expressing security against the Server is:

```
lemma GReal_GIdeal :
  exists (Sim <: SIM{GReal, GIdeal}),
  forall (Adv <: ADV{GReal, GIdeal, Sim}) &m,
  Pr[GReal(Adv).main() @ &m : res] =
  Pr[GIdeal(Adv, Sim).main() @ &m : res].
```

When proving this theorem, the only challenge is dealing with the fact that, in the Client's query loop, the real game does hashing that is absent in the ideal game. That hashing is redundant: its results are only placed in the Client's view, where nothing is done with them. Consequently we can use our abstract theory for removing redundant hashing (see Section VIII) to complete the proof.

X. Proof of Security Against Third Party

In this section, we consider the proof of security against the Third Party. We start by creating a private random oracle for hashing elements, not element/secret pairs:

```
clone RandomOracle as Priv with
  type input ← elem, op output_len ← tag_len,
  type output ← tag, op output_default ← zeros_tag,
  op output_distr ← tag_distr
proof *. (* realization *) · · · (* end *)
```

Now Priv.Or is the private random oracle for elements.

The Adversary will have limited access to the random oracle RO.Or (see Subsection VII-B):

```
op limit : int.
axiom limit_ge0 : 0 ≤ limit.
clone RO.Limited as LRO with op limit ← limit
proof *. (* realization *) · · · (* end *)
```

Thus LRO.LOr is the limited random oracle wrapper. To ensure termination of the Client's query loop, the Adversary will be constrained to proposing at most qrys_max queries:

```
op qrys_max : int.
axiom qrys_max_ge0 : 0 ≤ qrys_max.
```

The Adversary's module type is

```
module type ADV(O : RO.OR) = {
  proc * init_and_get_db(tpv : tp_view) : db option {O.hash}
  proc get_qry(tpv : tp_view) : elem option {O.hash}
  proc qry_done(tpv : tp_view) : unit {O.hash}
  proc final(tpv : tp_view) : bool {O.hash} }.
```

The TP's real game, GReal, is what you would expect. The Adversary is given access to the limited random oracle. Its environment submodule, Env, keeps track of the number of queries that have been processed. The init_and_get_db procedure of Env initializes the query counter and the limited random oracle. Its get_qry procedure returns None when the query processing limit has already been reached.

The TP's Simulator has the following interface:

```
module type SIM = {
  proc * init() : unit
  proc get_view() : tp_view
  proc receive_hdb(hdb : hdb) : unit
  proc count_tag(tag : tag) : int }.
```

It does not need to have access to the random oracle, in contrast to the Simulators of the Client and Server. The receive_hdb procedure is used to give the Simulator the hashed database produced by the Server, and the count_tag procedure lets the Client request the counts of hash tags in that hashed database.

The TP's ideal game, GIdeal, is parameterized by the Adversary Adv and Simulator Sim. Adv is given access to the limited random oracle. The game's structure is similar to that of GReal (after inlining and simplification), with Sim playing the part of the TP, except that the Server and Client do their element hashing using the private random oracle, Priv.Or.

The lemma expressing security against the TP is:

```
lemma GReal_GIdeal :
  exists (Sim <: SIM{GReal, GIdeal}),
  forall (Adv <: ADV{GReal, GIdeal, Sim}) &m,
  (* losslessness of Adv's procedures *) ⇒
  `|Pr[GReal(Adv).main() @ &m : res] −
    Pr[GIdeal(Adv, Sim).main() @ &m : res]| ≤
  limit%r / (2 ^ sec_len)%r.
```

The restrictions on SIM and ADV are crucial—otherwise, e.g., the Adversary or Simulator could access Priv.Or. The lemma is conditioned on the Adversary's procedures being lossless, and it upper-bounds the distance between the real and ideal games in terms of limit and sec_len.

In the proof, we must transition across a gap: in the real game, the Server and Client hash elements paired with their shared secret in RO.Or, whereas in the ideal game, the Server and Client do their element hashing in Priv.Or. We bridge the gap by employing an abstract theory SecrecyRandomOracle, which implements two versions of secrecy random oracles:

```
module type SEC_OR = {
  proc init(sec : sec) : unit
  proc lhash(inp : elem * sec) : output
  proc hash(elem : elem) : output }.
```

Initializing a secrecy random oracle takes in a secret, sec. Two hashing procedures are provided: lhash for limited hashing (up to limit distinct inputs) of element/secret pairs, and hash for unlimited hashing of elements. The first implementation of this interface uses a single map, as in the TP's real game, where hash hashes the pair of its argument $elem$ and sec, whereas the second implementation uses two maps—one for lhash and one for hash, as in the TP's ideal game. The theory defines games using these oracles, and proves a lemma bounding the distance between them. The idea is that unless a secrecy random oracle adversary calls lhash with a pair whose second component is sec (and without first exceeding its hashing limit), it cannot tell the games apart. The lemma is proved using reasoning up to failure, which requires the losslessness of the secrecy random oracle adversary. This, in turn, is why the security theorem for the TP requires the losslessness of the Adversary's procedures, and why the number of queries proposed by the Adversary must be limited.

The lemma's proof must also bound the probability that the failure event occurs, i.e., that lhash is passed the secret. We do this bounding using a SecretGuessing abstract theory. The secret guessing oracle implements this interface:

```
module type SG_OR = {
  proc init(x : sec) : unit
```

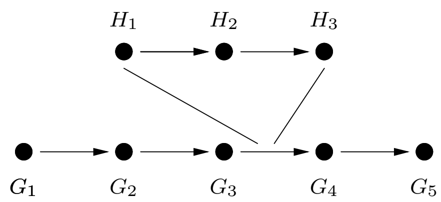

Figure 6. Two-dimensional Sequences of Games

```
proc guess(x : sec) : unit }.
```

The secret guessing game initializes the oracle, giving it a randomly generated secret. The secret guessing adversary then has a limited number (limit) of tries to guess the secret using the oracle's guess procedure. The proof uses EASYCRYPT's probabilistic Hoare logic to upper-bound the probability of the adversary guessing the secret by

$$\mathsf{limit}/2^{\mathsf{sec\_len}}.$$

XI. CASE STUDY RESULTS AND LESSONS LEARNED

In this section, we summarize the results of our case study and survey what we have learned from carrying it out.

A. On the Proof

The theorems expressing security against the Server, Third Party and Client, along with all the definitions needed to understand these theorems (including the definitions of the PCR Protocol and all the real and ideal games), total about 380 lines of EASYCRYPT code. It is only this code that must be carefully scrutinized so as to ensure the security theorems say what they should. EASYCRYPT can be trusted to faithfully check the approximately 5,275 lines of EASYCRYPT code comprising our proofs of these theorems plus supporting theories.

To minimize our reliance on particular SMT solvers, we have checked our proofs using two solvers: Alt-Ergo [26] and Z3 [27]. And in every use of an SMT solver, we have explicitly specified the previously proved EASYCRYPT lemmas that may be used by the solver when attacking the goal. This is good documentation, increases the speed of proof checking, and is very helpful when proofs need to be adapted.

The EASYCRYPT proof scripts for our case study are available on the web at:

https://github.com/alleystoughton/PCR

B. Two-dimensional Game Structure

In the sequence of games approach, to show a relationship between games G_1 and G_5 whose main procedures return booleans, one might make use of intermediate boolean-returning games G_2, G_3 and G_4, as in Figure 6. Some of these intermediate steps may show that source and target games are equally likely to return true, but for others we will have upper bounds on the absolute values of the differences between the probabilities that the games return true. One sums up these (hopefully small!) upper bounds (0 when there is no distance between the games), getting the distance between G_1 and G_5. But in EASYCRYPT, one can also make use of reductions, giving games a vertical as well as a horizontal structure. In the figure, we have used another sequence of games to establish the distance between H_1 and H_3. Let us suppose H_1 and H_3 are parameterized by an abstract adversary A of some type. We can package the proof connecting H_1 and H_3 into a theory. Then if we want to use this theory to establish the connection between G_3 and G_4, we clone this theory in the context of G_3 and G_4, and define a concrete adversary C of the same type as A so that G_3 can be connected with $H_1(C)$, and $H_3(C)$ can be connected with G_4.

We have made important use of this vertical approach in our security proof for PCR, using both our own theories and theories of the EASYCRYPT Library. The reduction of Third Party security to the security of secrecy random oracles, which was in turn reduced to the security of secret guessing oracles, was a prime example of this (see Section X).

C. Expressing Real and Ideal Games

An earlier version of our work suffered from the drawback that each party's real game had to be written out from scratch, even though it was largely the same as the other parties' real games. In addition to being tedious, this allowed for the possibility of the games being inconsistent. Thankfully, we now have a solution to this problem: the protocol is formalized once and for all, complete with code maintaining all parties' views (one must carefully scrutinize this code to ensure it faithfully records sufficient information for each party's execution to be reconstructable). The protocol is parameterized by an Environment, with which it interacts. The real game for a given party can then be obtained by instantiating the environment with code connecting the protocol to the party's Adversary. Limits on the Adversary can naturally be expressed in this code. Because the Protocol Environment is adaptive, this gave us a good start toward handling adaptive Adversaries.

As explained in Sections VIII—X, we parameterized each party's ideal game by both its Adversary and the party's Simulator, which constructs the party's view from the limited information given it by the ideal game. Because we are working information-theoretically, we were able to make the Simulators be part of the security *proofs*, as opposed to the security *specifications*. Consequently, in each of our security theorems, the Simulator is existentially quantified:

```
lemma GReal_GIdeal :
  exists (Sim <: SIM{GReal, GIdeal}),
  forall (Adv <: ADV{GReal, GIdeal, Sim}) &m,
  . . .
```

The restrictions on the module types SIM and ADV express that the Simulator, Sim, and Adversary, Adv, may not interact with each other or the real/ideal games (except via their module parameters). This is crucial, as otherwise we could prove such a theorem using a Simulator that, e.g., read variables of the ideal game—which would be unsound.

D. Limiting Adversaries

As explained in Section V, to obtain security theorems against the Client and Third Party with small upper bounds, we needed to limit the Adversary. For the Third Party proof (see Section X), it sufficed to limit the number of distinct inputs the Adversary may hash before being given a dummy result when hashing new inputs.

For the Client proof (see Section VIII), we developed a technique of budgeted random hashing allowing us to transition in and out of oracles whose maps remain collision-free as long as the budget is respected. Using this technique allowed us to attack the key step of the Client proof—moving from hashed databases to elements' counts maps—without the distraction of possible hashing collisions. We believe this kind of technique will be essential when working with more complex protocols.

The Client and Third Party security theorems are quantified over all lossless Adversaries (ones whose procedures always terminate). But when an Adversary runs up against a limit imposed on it, the real/ideal game is terminated early (Client proof) or the Adversary's hashing stops yielding true results (Third Party proof). Consequently, one may view these security theorems as being quantified over all lossless Adversaries that respect the limits that would otherwise be imposed on them.

By using reasoning up to failure and EASYCRYPT's probabilistic Hoare logic and failure event lemma, we were able to upper-bound the distances between real and ideal games using bounds built up from game parameters (sizes of hash tags and Server/Client secrets, limits on the Adversary).

E. Removing Redundant Hashing

Grégoire [25] recently invented a general approach to removing redundant hashing, and we employed our implementation of a variation of his technique in both the Client and Server proofs (see Sections VIII and IX). We believe variations of this technique will be essential when working with more complex protocols.

XII. NEXT STEPS

Having developed and tested techniques for proving the information-theoretic, adaptive security of multi-party cryptographic protocols involving random oracles, our next goal is to tackle a protocol—probably a protected database search protocol—involving encryption as well as hashing. Our idea is to model encryption in a way similar to the random oracle: as a construction whose encapsulated state depends upon dynamically made random choices.

A good candidate protocol appears to be the privacy-preserving sharing of sensitive information protocol of [28], which uses symmetric encryption as well as four random oracles. This protocol's adversary is non-adaptive, and its security proof doesn't follow a sequence of games style. But we are optimistic we can formulate and prove the security of an adaptive version of the protocol using our approach.

Additionally, we plan to explore the connections between our work and the Universal Composability (UC) model [29]. Adapting our security proofs of protocols to the UC model would have the consequence of preserving the protocols' security guarantees when they are combined with other protocols. In our architecture, the Adversary/Environment is already charged with both choosing protocol inputs and attempting to distinguish the real and ideal games; ergo, we believe that this extension will be feasible.

ACKNOWLEDGMENT

The reported work was partially completed while the authors were employed at MIT Lincoln Laboratory, funded by the Intelligence Advanced Research Projects Activity under Air Force Contract FA8721-05-C-0002. The second author's contributions were subsequently supported by the National Science Foundation under Grant No. 1414119.

It is a pleasure to acknowledge helpful discussions with Gilles Barthe, Ran Canetti, Robert Cunningham, François Dupressoir, Benjamin Grégoire, Jonathan Herzog, Aaron D. Jaggard, Jonathan Katz, Catherine Meadows, Adam Petcher, Emily Shen, Pierre-Yves Strub, Arkady Yerukhimovich and Santiago Zanella Béguelin. Special thanks to Zanella Béguelin for suggesting that security against the Third Party could be strengthened were the Server to begin by randomly shuffling its database.

REFERENCES

[1] R. Canetti, "Security and composition of multi-party cryptographic protocols," *J. Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[2] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004.

[3] M. Bellare and P. Rogaway, "Code-based game-playing proofs and the security of triple encryption," *IACR Cryptology ePrint Archive*, vol. 2004, no. 331, 2004.

[4] ——, "The security of triple encryption and a framework for code-based game-playing proofs," in *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT 2006. Saint Petersburg, Russia: Springer-Verlag, 2006, pp. 409–426.

[5] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004, http://eprint.iacr.org/2004/332.

[6] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS 1993. Fairfax, VA, USA: ACM, 1993, pp. 62–73.

[7] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of Security Analysis and Design VII*, ser. Lecture Notes in Computer Science, A. Aldini, J. Lopez, and F. Martinelli, Eds. Springer International Publishing, 2014, vol. 8604, pp. 146–166.

[8] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proceedings of the 31st Annual Conference on Advances in Cryptology*, ser. CRYPTO 2011. Springer-Verlag, 2011, pp. 71–90.

[9] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "SoK: Cryptographically protected database search," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, ser. SP 2017. San Jose, CA, USA: IEEE Computer Society, 2017, to appear.

[10] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin, "Beyond provable security: verifiable IND-CCA security of OAEP," in *Proceedings of the 11th International Conference on Topics in Cryptology*, ser. CT-RSA 2011. San Francisco, CA, USA: Springer-Verlag, 2011, pp. 180–196.

[11] M. Backes, G. Barthe, M. Berg, B. Grégoire, C. Kunz, M. Skoruppa, and S. Zanella Beguelin, "Verified security of Merkle-Damgård," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, ser. CSF 2012. Washington, DC, USA: IEEE Computer Society, 2012, pp. 354–368.

[12] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Zanella Béguelin, "Proving the TLS handshake secure (as it is)," in *Proceedings of the 34th Annual Conference on Advances in Cryptology*, ser. CRYPTO 2014. Santa Barbara, CA, USA: Springer-Verlag, 2014, pp. 235–255.

[13] G. Barthe, F. Dupressoir, P. Fouque, B. Grégoire, M. Tibouchi, and J. Zapalowicz, "Making RSA-PSS provably secure against non-random faults," in *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES 2014. Busan, Korea: Springer-Verlag, 2014, pp. 206–222.

[14] G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt, "Mind the gap: Modular machine-checked proofs of one-round key exchange protocols," in *Proceedings of the 34th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT 2015. Sofia, Bulgaria: Spring-Verlag, 2015, pp. 689–718.

[15] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Zanella Béguelin, "Fully automated analysis of padding-based encryption in the computational model," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 2013. Berlin, Germany: ACM, 2013, pp. 1247–1260.

[16] B. Blanchet, "Computationally sound mechanized proofs of correspondence assertions," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, ser. CSF 2007. Venice, Italy: IEEE Computer Society, 2007, pp. 97–111.

[17] D. Cadé and B. Blanchet, "From computationally-proved protocol specifications to implementations and application to SSH," *JoWUA*, vol. 4, no. 1, pp. 4–31, 2013.

[18] B. Blanchet, A. D. Jaggard, A. Scedrov, and J. Tsay, "Computationally sound mechanized proofs for basic and public-key Kerberos," in *Proceedings of the 3rd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS 2008, Tokyo, Japan, 2008, pp. 87–99.

[19] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *Procedings of the 4th International Conference on Principles of Security and Trust*, ser. POST 2015. London, UK: Springer-Verlag, 2015, pp. 53–72.

[20] Coq Development Team, "The Coq proof assistant," https://coq.inria.fr.

[21] A. Petcher and G. Morrisett, "A mechanized proof of security for searchable symmetric encryption," in *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, ser. CSF 2015. Verona, Italy: IEEE Computer Society, 2015, pp. 481–494.

[22] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proceedings of the 33th Annual Conference on Advances in Cryptology*, ser. CRYPTO 2013. Santa Barbara, CA, USA: Springer-Verlag, 2013, pp. 353–373.

[23] C. Bösch, P. H. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 18:1–18:51, 2014.

[24] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2009. Savannah, GA, USA: ACM, 2009, pp. 90–101.

[25] B. Grégoire, "Technique for removing redundant hashing in EasyCrypt," Personal communication with first author, January 2016.

[26] Alt-Ergo Development Team, "The Alt-Ergo SMT solver," http://alt-ergo.lri.fr.

[27] Z3 Development Team, "The Z3 SMT solver," https://github.com/Z3Prover/z3.

[28] E. De Cristofaro, Y. Lu, and G. Tsudik, "Efficient techniques for privacy-preserving sharing of sensitive information," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, ser. TRUST 2011. Pittsburgh, PA, USA: Springer-Verlag, 2011, pp. 239–253.

[29] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Procedings of the 42nd Annual Symposium on Foundations of Computer Science*, ser. FOCS 2001. Las Vegas, NV, USA: IEEE Computer Society, 2001, pp. 136–145.