Poster: Foundational Approaches to Formalizing Cryptography can be both Powerful and Usable

Devon Tuma
Dept. of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
tumax040@umn.edu

Nicholas Hopper
Dept. of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
hoppernj@umn.edu

Introduction

As cryptographic protocols and their proofs continue to become more complex and specialized, methods for verifying the correctness of security proofs have become more and more important. Formal verification via proof assistants is one promising solution to this problem, and a number of systems have been developed to reason about security in this way.

Our work presents a new framework for verifying security proofs, taking a foundational approach to representing different protocols and computations. We implement our framework in the Lean programming language, and give a number of security proofs to demonstrate that our system is both powerful and usable, with comparable automation to other systems. One particularly powerful result is an implementation of the general forking lemma proposed by Bellare and Neven [4], which to our knowledge is more powerful than any previous rewinding mechanisms in other systems, and that many would be unable to even represent in a non-axiomatic way.

Motivation

A wide variety of frameworks have been developed to reason about cryptographic proofs in formally verified ways, which all have different pros and cons depending on specific use cases. Some like EasyCrypt [6], SSProve [7], and CryptoVerif [8] take a "top-down" approach that focuses mainly on high-level functionality and behavior, often using a custom language and proof system to represent and reason about computations. Others like FCF [4], CertiCrypt [5], and CryptHOL [9] take a more "foundational" approach, building all constructions from a small computing base with limited axiomatization. Usually computations in this approach are represented by a shallow embedding into a more general proof assistant like Coq and Isabelle. The foundational approach has significant benefits in terms of flexibility and extensibility, but has some difficulties in terms of automation and usability.

Additionally, existing foundational systems have a number of limitations in how they reason about the oracles available to computations. In FCF for example oracle access is specified by a single oracle with fixed input and output types. This makes it difficult to reason about situations where multiple oracles are available, e.g. both a coin-flipping oracle and a random oracle. Considering multiple oracles then requires making use of dependent product types to index the possible input and output types, which means that input and output types of queries aren't known at compile time and type checking failure cases must be handled by the end-user. Many systems overcome this by making random sampling and other oracles into different objects in their representation, but this both adds additional cases to proofs and weakens the ability to reason about both types of oracles at once.

We define a generalized notion of oracle access, where all the oracles available are parameterized by some indexing set. This allows for inputs and types to be checked at compile type, greatly simplifying the process of working with oracle queries. One especially powerful effect of our approach is that it allows us to avoid separating probabilistic computation and other types of oracle access, and this unification allows us to give stronger semantics for simulating the oracles in a computation, which is key to defining our generalized forking mechanism.

However our generalization also presents a number of significant usability issues, especially in composability, modularity, and proof complexity. We develop a number of mechanisms to handle these issues, including the use of automatic type coercions, tactic programming, custom induction principles. In general our system still requires more developer work in order to construct back-end foundations, but achieves comparable proof complexity to other systems in actually verifying cryptographic protocols, which we demonstrate by proving the security of multiple different constructions.

Defining the Framework

Our framework represents computations and protocols via a shallow embedding into the underlying proof assistant, with our basic approach being similar to the one used by FCF. This means that functions and types in our model of computations are exactly the same as functions and types in the underlying language, allowing all of the constructions and definitions in the underlying language to be used directly in this representation as well. In particular any lemmas and theorems proven in the underlying system can be applied

[1]This work was supported by an NSF Graduate Research Fellowship and NSF grant 1814753

directly without change, avoiding the need to reimplement functionality in our representation of computations.

We do this by representing computations using free monads, i.e. an inductively defined type where the monadic `return` and `bind` operations are defined as explicit type constructors. This is augmented by a single `query` constructor for querying one of the available oracles. The set of oracles is specified by a structure that contains an indexing set for the oracles as well as the input and output types for each index. Choosing which oracle to call is then equivalent to providing the corresponding element of the indexing set. Further constructions like try-fail operations and while loops are constructed as abstractions on top of this.

We also define two types of semantics for this system, generalizing the semantics used in FCF. The first is a denotational semantics that associates a probability mass function to computations. The probability of an event holding after a computation is then just the total probability mass of the event.

The other is a small step semantics that allows oracles to be simulated with some simulation function, potentially maintaining some internal state as it goes. One example of this is query logging, which can be implemented by simulating the queries, using the internal state to track the input and output of each query. We allow the entire set of oracles to be simulated at once, but give ways to simulate only a partial subset of them as well.

Implementing the framework

We give an implementation of our system in the Lean programming language, with strong integration to the open source mathlib project [1]. We use mathlib to handle all of the mathematical foundations of our project, which provides a large source of existing theory and proofs that can be used directly in our system. Significant portions of our work were written as contributions to the mathlib library, separating only the portions of the project that are strictly cryptographic.

The unified nature of the mathlib library means that the different semantics and constructions underlying our system have high interoperability, which makes it easy to transition between different forms of reasoning.

We also make significant use of features of the Lean programming language in order to handle usability issues presented by our generalization. Tactic programming in particular (a form of meta-programming for proofs) allows us to hide most low-level complexity from end users. This significantly increases the complexity of some foundational aspects, but greatly reduces the complexity of proofs written by end-users.

Cryptographic Constructions

To demonstrate the usefulness of our framework we implement a number of different common cryptographic constructions. We focus on showing how different parts of the framework can be used to reduce proof complexity, and emphasize readability and familiarity for those without significant exposure to proof assistants.

Firstly we define symmetric encryption schemes and the information-theoretic notion of perfect secrecy, and give a concrete implementation of one-time pad as a specific example. We then prove Claude Shannon's theorem characterizing perfect secrecy in terms of independent random variables, and use this to show the security of one-time pad.

Next we construct a general forking lemma based on that of Bellare and Neven [4], which provides more powerful semantics than previous implementations in other systems, and avoids any axiomatization. Recent work [10] has implemented a rewinding mechanism in EasyCrypt, but their approach is limited to rewinding an adversary to some specific point and is limited to a subset of "rewindable" adversaries. Our approach allows an algorithm to be forked at an arbitrary query, decided post-hoc from the resulting output and state after the first execution, and can be applied to an arbitrary adversary in the system.

Finally we define the notion of Hard Homogenous Spaces as proposed by [3] and construct a Schnorr style signature scheme from them, and use our general forking lemma to give a simple proof of unforgeability via a reduction to the vectorization problem for the HHS.

Future Work

A very useful piece future work would be to develop a standardized approach to handling game-based security proofs in the system. One very promising route would be to take the approach of SSProve [7], which provides high level functionality for this. Their system uses a free monad to represent low level computation, and as they note in their paper it should be possible to implement a similar framework on top of our foundational system.

References

[1] The mathlib community, the lean mathematical library. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. January 2020.

[2] Bellare, Mihir and Neven, Gregory. 2005 New Multi-Signature Schemes and a General Forking Lemma.

[3] Couveignes, Jean-Marc. 2006. Hard Homogeneous Spaces.

[4] Petcher, Adam. 2015. A Foundational Proof Framework for Cryptography. Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences.

[5] Zanella-béguelin, Santiago. 2011. Formal certification of game-based cryptographic proofs, Computer Science and Game Theory

[6] Canetti, Ran, Stoughton, Alley, and Varia, Mayank. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. 32nd IEEE Computer Security Foundations Symposium (CSF 2019).

[7] Philipp G. Haselwarter and Exequiel Rivas and Antoine Van Muylder and Théo Winterhalter and Carmine Abate and Nikolaj Sidorenco and Catalin Hritcu and Kenji Maillard and Bas Spitters. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. 2021. Cryptology ePrint Archive, Paper 2021/397

[8] Blanchet, Bruno. CryptoVerif: A Computationally-Sound Security Protocol Verifier. November 2017.

[9] Basin, David, Lochbihler, Andreas, and Sefidgar, Reza. CryptHOL: Game-based Proofs in Higher-order Logic. 2019.

[10] Firsoz, Denis and Unruh, Dominique. Reflection, Rewinding, and Coin-Toss in EasyCrypt Denis. CPP 2022 - Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. 2022.