Explorations at the Intersection of Cryptography and Verification

Jared Roesch
jroesch@cs.ucsb.edu

Abstract— **We demonstrate a prototype framework for light weight reasoning about the correctness and performance of cryptographic algorithms. Our framework is informed by the state of the art in machine checked verfication, but goes beyond it in the pursuit of building a framework for the specification, verification, optimization, and execution of algorithms.**

I. Introduction

Cryptographic algorithms and protocols are quickly becoming more prevelant, more special purpose, and more complex. In the face of this reasoning about their correctness is still not easy, and requires a combination of proof, testing, and peer review. Many of these algorithms have also become essential to operation of services we rely on in our day to day lives, and we require their implementations to not only be correct, but fast to scale to real world use.

The question becomes: How do give cryptographers powerful tools to specify, verify, test and optimize algorithms? Traditionally each acitivity has required a different sets of tools that worked in isolation. Our goal in this paper is to introduce a framework that allows us to perform each activity with a single tool chain.

We allow the user of our tool to write down executable specifications of their algorithms with boths proofs and examples as well as exposing machinery for automatically solving common proof obligations.

We can leverage the power of this approach in order to build our specification language. By embedding our language into the theorem prover Lean we gain all of its theorem proving power and capbilities. Lean is a good choice here because its foundational system is type theory, enabling us directly execute programs written in the framework. This is a well known technique for verifying computational domains, it is discussed in depth in section II.

We implement a embedded domain specific language inside of Lean built around the manipulation of depedently typed bit vectors, allowing us to verify the correctness of algorithms and extract an implementation.

II. Related Work

Embedded domain specific languages are a core idea in programming languages and one that both our work and all prior work are inspired by. Embedded domain specific languages come in two primary flavors: shallow or deep. In a shallow embedding the language is directly translated from the domain specific language into the host language operations. In a deep embedding the language is used to construct an abstract syntax tree that represents the desired program that can then be interpreted in multiple ways, or compiled to another format.

It is a common approach to build DSL in theorem proving environments as it allows one to encode a domain's semantics and verif various properties about the programs written in the DSL. There have been many different approaches to this from hardware description[1][2], reactive systems[3], distributed systems, and cryptography.

There have been many other approaches to verifying the correctness of cryptographic systems. The first attempt to build an embedded verification language is CertiCrypt [4][5][6][7].

CertiCrypt is a machine-checked framework built on top of Coq. Coq is an industry standard proof assistant that implements dependent type theory. CertiCrypt is a deep embedding of an extensible imperative probablistic programming language that allows one to describe both security goals and algorithms as probablistic programs. The framework also allows the characterization of adversaries as probablistic polynominal-time programs.

EasyCrypt is an evolution of CertiCrypt in which they leverage tools such as SMT solvers in order to improve proof automation. After the release of EasyCrypt a team of cryptographers and programming language researchers attempted to prove security properties of a private information retreival system and were unable to write a complete proof due to limitations of the system. These fundamental limitations lead to the newest effort in this area: The Foundational Cryptography Framework[8].

The primary difference is that they use a shallow embeding of the language instead of a deep emebedding. The shallow embedding means their constructs are interpreted into normal Coq expressions, which make the entire system more extensible and flexible.

Our approach is less focused on formalization but providing a operational semantics to the langauge that allows for the translation and execution of real programs.

In this way we are influenced by Cryptol[9] which is a domain specific language for specifying cryptographic algorithms, that can be extracted to either a code or hardware implementation. The langauge consists of bit vectors and functional style operations for transforming them, the goal is to have the Cryptol specifications closely resemble what cryptographers write on paper.

III. Lean

Lean is a new theorem proving environment from Microsoft Research. It has been designed to be a modern,

highly performant implementation by leveraging state of the art ideas in constraint solving and synthesis, as well as providing the flexibility to explore new ideas in type theory such as Homotopy Type Theory[10].

It key features are:

- powerful elaboration engine
- small trusted kernel
- flexible kernel
- incremental compilation
- universe polymorphism
- mixed declarative and tactic proof style
- powerful automation

A. Theorem Proving

Theorem Proving has been an area of active research for many decades. It is apparent that the verification of semantic properties are desirable for work in high assurance software systems, computer science and mathematics. There have been various approaches to Automated Theorem Proving in the past ACL2, NuPRL, LF, LCF, Coq, Agda, and Lean. The ones based on dependent type theories seem to be the most promising for various reasons:

- dependent types correspond to higher-order logic
- tools for automated proof search and generation
- specification and implementation happen in the same language
- executable "proofs" (i.e program extraction)

B. Type Theory

The programming language community has a long history with type theory, a computational proof theory that allows us to equate programs with proofs, and types with propositions.

Type theory is at its core a foundational framework for mathematics. It was a school of thought that evolved out of Russel's work on foundational systems. One of the key insights in its developement and adoption was the Curry-Howard Isomorphism, which states that types can be associated with propositions and proofs with programs. This allows one to write down a specification (a type) satisfy it with a program (a proof) and then execute the program for a result. Because of this type theory should be interesting to computer scientists, it means our foundations can simply be a programming language.

Traditional programming languages have often treated types as a means of enforcing memory layout, like in C, but they can be much richer. Traditional type systems are very rigid and maintain a strict phase distinction; types and values live in separate universes. If we ease this restriction and freely allow programmers to intersperse types and values we end up dependent types, which happen to correspond to higher order logic. This is of course a very brief overview and more information can be found in [11] [12] and [10].

From a theoretical perspective the majority of interesting mathematics and and verification can be formalized using a small set of features from dependent type theory:

- type universes
- dependent function spaces
- inductive types

The key concept is that of a dependent function space or Π type. A Π type represents a function where the codomain is dependent on the value of the domain. Logically this means our propositions can be predicated on the value of their parameters. We can extend a lambda calculs with dependent functions with inductive types to have something powerful enough to implement a theorem proving environment and formalize large chunks of mathematics.

Inductive types makes this possible; they define a type, a set of type formers (constructors) and an elimnator (or computation rule). We can use these to build up data types that represent data structures or proofs and compute based on them via eliminaton rules.

Lean implements many other nice features by elaborating them to these simple core constructs. Lean's focus on engineering quality has made it an attractive choice for this project.

IV. Vrypto

In this section we describe the current state of our prototype framework Vrypto a portmanteau of Verified Crypto. Our framework is designed as an embedded domain specific language of bits and sequences. We use a a technique called parametric higher order abstract syntax in order to represent variable binding, and we paramertrize our term language by both the varible binder and type of expression, this means our programs are type correct by construction, so any term we build is guarenteed to be correctly typed.

Our type system for the language currently supporting bits, sized sequences, arbitrary sized integers and functions.

```
inductive ctype : Type :=
    | Unit     : ctype
    | Bit      : ctype
    | Sequence : forall (n : nat) (a : ctype), ctype
    | Fn       : ctype -> ctype -> ctype
```

This type language seems simplistic but actually captures most of the interesting behavior we need. We can easily define more complex types for example numbers just become sequences of bits.

```
definition Number (n : nat) : ctype :=
    Sequence (divide n 2) a
```

Our main language supports the definition and use of functions, streams, and bits. We can use these simple pieces to write relatively complex programs, as well as reason about program behavior. The encoding we use makes it possible to define transforms simply as recursive functions over the tree structure. We can also utilize proofs

of equality to prove the equivalence of programs, and thus algorithms.

```
variable {var : ctype -> Type}

inductive crypto : ctype -> Type :=
    | One  : crypto Bit
    | Zero : crypto Bit
    | Num  : forall {n}, crypto (Number n)
    | Empty : forall {c : ctype},
                crypto (Sequence zero c)
    | Cons : forall {n} {c : ctype},
                crypto c ->
                crypto (Sequence n c) ->
                crypto (Sequence (succ n) c)
    | Var  : forall {t}, var t -> crypto t
    | Abs  : forall {domain range},
                (var domain -> crypto range) ->
                crypto (Fn domain range)
    | App  : forall{domain range},
                crypto (Fn domain range) ->
                crypto domain ->
                crypto range
    | Let  : forall {t1 t2},
                crypto t1 ->
                (var t1 -> crypto t2) ->
                crypto t2
```

Currently there are some unfinished parts of the proofs and we can not demonstrate a complex example. When finished it would be possible to write statements like so:

```
example : (complexity algo1) = (complexity algo2)
```

Where complexity is simply a recursive function that computes the complexity class of an algorithm.

V. Future Work

Our current framework is very bare bones and we would like to pursure a couple directions in the future:

- hardware compiliation
- proof automation
- reasoning about complexity
- optimization

I think that providing a tool kit for reasoning about optimization of algorithms and hardware generation will be the most interesting future directions. The current language is designed to be amendable to extraction to a hardware specification in a hardware description language, and the main challenge would just be getting the extraction implemented and proving correspondence between the high level algorithm and hardware.

The other direction I see being useful is taking more inspiration from previous verification efforts and reasoning about the computational complexity of algorithms and being able to prove equivalence between optimized an unoptimized algorithmic approaches. This would most likely require a redesign of the core language in order to support this kind of reasoning.

Constructing a verification framework requires a lot of experimentation and refinement to find the right tradeoff between complexity and simplicity in order to end up with a final product that is both ergnomic and expressive.

References

[1] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st Symposium on Principles of Programming Languages*, Jan. 2014, POPL.

[2] Thomas Braibant and Adam Chlipala, "Formal verification of hardware synthesis," *CoRR*, vol. abs/1301.4779, 2013.

[3] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner, "Automating formal proofs for reactive systems," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2014, PLDI '14, pp. 452–462, ACM.

[4] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella-Béguelin, "Verified indifferentiable hashing into elliptic curves," in *1st International Conference on Principles of Security and Trust, POST 2012*. 2012, vol. 7215 of *Lecture Notes in Computer Science*, pp. 209–228, Springer.

[5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin, "Probabilistic relational reasoning for differential privacy," in *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. 2012, pp. 97–110, ACM.

[6] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. 2009, pp. 90–101, ACM.

[7] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella-Béguelin, "Verified indifferentiable hashing into elliptic curves," *Journal of Computer Security*, 2013, To appear.

[8] Adam Petcher and Greg Morrisett, "The foundational cryptography framework," *CoRR*, vol. abs/1410.3735, 2014.

[9] Sally Browning and Philip Weaver, "Designing tunable, verifiable cryptographic hardware using cryptol," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin, Ed., pp. 89–143. Springer US, 2010.

[10] The Univalent Foundations Program, "Homotopy type theory: Univalent foundations of mathematics," Tech. Rep., Institute for Advanced Study, 2013.

[11] Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.

[12] B. Nordstrm, K. Petersson, and J. M. Smith, "Martin-lfs type theory," 2000.