Constructive Synthesis of Optimized Cryptographic Primitives

by

Robert M. Sloan, Jr.

S.B., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by .....
Adam Chlipala
Associate Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Constructive Synthesis of Optimized Cryptographic Primitives

by

Robert M. Sloan, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, I designed and implemented a formally verified compiler for programs over arbitrary-width integers in Parametric Higher-Order Abstract Syntax. This was part of a larger project, *Fiat-crypto*, which seeks to produce formally verified machine-code implementations of Elliptic Curve Cryptography.

My implementation outputs *Qhasm*, a high-level assembly language developed for the implementation of highly optimised cryptography, and maintains platform independence by being totally parametric over the width of system words.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor

Acknowledgments

I would like to thank my parents for their continual encouragement, and I would like to thank my advisor, Adam Chlipala, for putting up with me.

Contents

Chapter 1

Introduction

In order for the internet to function as a social institution, people need to *trust* it, to believe that when computer scientists say that their data is safe, well, it is. But in recent years, we have seen many high-profile bugs in OpenSSL, the real bedrock of web security for most major browsers [26]. That doesn't only happen – as one might expect – with highly sophisticated cryptanalytic attacks [1]; but, concerningly, we also see many ordinary software exploits involving very basic mistakes [4]. That is really not condusive to belief in the quality of software as a whole [17], and so our question is an essential one: how do we design our most important software so that this does not (and ideally, cannot) happen?

From an engineering perspective, though, it is weird that this is happening: one would expect to see some new open-source project that re-implements the difficult functionality in a new and more sophisticated way. However, attempts at a more organized solution are fragmentary [2, 3], mostly because none of them provide a strong reason for trust outside their respective organizations, and even though there are some attempts at validating some pieces of cryptographic code more generally, it isn't widespread [6].

So let us ask the obvious question: why is this hard? Why do errors in cryptography not lend themselves to the software-engineering solutions computer scientists are taught in every university in the world?

Well, the first problem is *speed*: any cryptographic implementation competing

with OpenSSL will be asked to do quite a lot of computation for every packet in every HTTPS session. Because of the way optimization works, any less mature implementation will be orders of magnitude slower [19], which creates an extremely strong disincentive for organizations to switch.

And even if somebody tried to do a good job with optimization, it might not work: the ordinary way that we make software really fast is that someone goes down and implements the "hot" functionality in actual machine code [7]; but because of the extreme complexity of assembly code, nobody else will trust that they didn't make any mistakes. This is the case even in the higher-level components of the system, where the state machines are complex enough to make implementation difficult [2].

Compounding the trust problem is the fact that cryptography is extremely high-stakes: browsers like Chrome have many billions of users [27], and even extremely subtle mistakes can have amazingly large impact [15], which is why government entities will pay millions of dollars for clever exploits [21]. For this reason, it's unlikely that a highly optimized implementation would be adopted absent other qualifications (e.g. being written by D. J. Bernstein [7]).

The normal way that we would get around the trust problem is to have a standards agency write down a bunch of test cases that a "valid" implementation should satisfy, and this can get you a long way; but in the context of cryptographic implementation, it is often impossible to test for there being no possible exploit (though projects like *Wycheproof* do try [18]). So here, we have a crisis in verifiability, and we have a seeming Gordian Knot:

> Because cryptographic implementation is so complex and difficult to validate, no one will trust a new solution; and even if a trustworthy one presented itself, it's unlikely to ever be adopted for performance reasons.

But there is a way to cut through this seemingly intractable issue: *proof*. Proof assistants like *Coq* can produce certificates that programs fulfill arbitrary predicates, providing a clearly very strong reason to believe in the correctness of the implementation [13]; so if we can make our implementations *fast enough*, we've solved the whole

issue.

The subject of this paper, named Qhasm Pipeline for reasons that will soon become apparent, seeks to provide a provably valid compiler for arbitrary programs over bounded integers (i.e. machine words), a space large enough to encompass the relevant cryptographic functions, with the idea that it will connect to a formal specification of the algorithm. It had a unique feature set at the time:

1. The implementation is built atop Coq, a proof assistant that can prove that the resultant implementation always fulfills our desired contract, regardless of the complexities within the compiler.

2. It uses a computer algebra system embedded in the type system of Coq to formally specify the target algorithm in ordinary algebra [23], thereby eliminating the need (and ability) for programmers to manipulate the logic directly.

3. It provides a sequence of formally verified compilation steps – the *Pipeline* part – that elaborate the initial algorithm into more complex program logics, e.g. from ML-style functional programs, to a domain-specific language, to assembly.

4. At its conclusion, the pipeline generates machine code in a high-level assembly language called *Qhasm*, which is important because we can manipulate the code-generation procedure to optimize the resultant assembly while maintaining contractual invariants and subverting the need to trust another compiler.

In this document, I will outline the whole project, trying my best to elide the low-level details of proofs and provide a practical understanding of both how the design worked and its flaws.

Extension to *Fiat-Crypto*

Toward the conclusion of (and following) my M. Eng., a larger group (including myself) used similar ideas to implement *Fiat-Crypto*, which performs end-to-end synthesis of assembly implementations of EdDSA, an Elliptic Curve signature algorithm

[8]. It is the first framework to synthesize formally verified code for Elliptic Curves, which can deal with the problems of efficiency, abstraction, and trust simultaneously. In the specific context of EdDSA, it presents a model for how the next generation of cryptographic implementation could work, and the approach readily generalizes to other algorithms.

The project, however, was much larger than just me, and is still ongoing, so I will not devote a huge amount of time to covering it, as that should be done adequately by our paper [5]. Suffice it to say that synthesis of these algorithms is very possible, even if an actually workable implementation took far more sophistication than my main M.Eng. project had.

Previous Work

There is a large literature about code generation, with some extremely well-developed and sophisticated methodologies [24, 25]; and I am even aware of one researcher who constructed a differently focused BIGNUM library [22]. However, few tools can simultaneously *generate* and *verify*, as the code-generation tools do not generally play well with the verification tools, and the verification tools rarely generate code. The most ready example of a tool that does both, albeit in a totally different context, is actually the *Fiat* from which *Fiat-Crypto* derives its name [16].

There have been several attempts at building verified libraries for finite-field arithmetic, with the specific target of cryptography:

1. *Verif25519* attempted to show correctness of the internal "Montgomery Ladder" implementation of Curve25519 for x64, or at least that the implementation correctly matched their higher-level model [11]. Their implementation mostly used an SMT solver, with minimal Coq code for goals the solver could not deal with.

2. *GFVerif*, which was written by many of the same authors, compiles C code to a Sage program by replacing the machine integers with symbolic variables, and

solves out the resultant equations via a Gröbner-basis representation [10]. The paper described the tool as an "early experiment" and an "alpha test".

3. *ECC-star* synthesizes the scalar-multiplication procedure for three Elliptic Curves in Microsoft Research's F*, which can only be run as managed code [28]. This is a relatively simpler implementation than GFVerif, and is slower by a factor of 290.

We can make many complaints of the situation here: the first two have fragile implementation strategies that likely do not generalize to more complex situations (e.g. P256), and the third is too simplistic to support sophisticated optimization, not to mention its lack of interaction with real assembly.

This makes a reasonable case that the Qhasm Pipeline – if not something totally new – is a positive step toward a general, sustainable method for producing efficient implementations of the finite-field code used for cryptography.

Chapter 2

Coq, Elliptic Curves, and other Preliminaries

But before we can get into the details of my pipeline, we have to slog through the mathematical and algorithmic preliminaries:

2.1 The Shape of our Input: Elliptic Curve Cryptography

I will not attempt to give a full description of Elliptic Curve Cryptography (ECC), or even a derivation of its primitives: for the purposes of developing the *compilation* pipeline, only need to understand the general function space; so the following sections describe the relevant operations only insofar as someone simply implementing the algorithm is concerned.

The general story I wish to justify is this: my input syntax trees will be extremely large, but will only consist of ordinary arithmetic operations and conditionals.

2.1.1 Finite Fields

Most public-key cryptography – and ECC is no exception – is based on *number-theoretic* problems, i.e. problems grounded in the shape and symmetries of mathe-

matical objects with a fixed number of elements. The fundamental species of symmetry is, well, a cycle, and the way number theory considers it is via the familiar *modulo* operation, in sets formalized as:

$$\mathbb{Z}_N \equiv \{x | x = x \mod N\}$$

Most of the numbers in the following sections will be members of such a finite field – in the case of *Curve25519*, for example, the relevant space is, as seems appropriate, $\mathbb{Z}_{2^{255}-19}$ [7]. Indeed, that is exactly what makes the exact implementation of Elliptic Curve operations possible.

Now, my first key point: every finite field involved here is *large*, generally on the order of 2^{256}, when the maximum value representable by most computers is about 2^{64}. For this reason, we cannot simply use the assembly operations, but rather have to *implement* those operations in terms of a radix representation in the smaller field. I did not write the code to perform that translation; however, it is important to note that that translation step means that there are a lot of bitwise operations, as well as right and left shifts, in my input.

Perhaps confusingly, except for this section, when I say "Integer", I am usually referring to the unbounded-size integers inherent in the language I use for implementation.

2.1.2 Elliptic Curves

Regardless of how hard mathematicians try to draw the intuition, elliptic curves are just extremely abstract objects, generally described as the space of solutions (x, y) to equations of the form:

$$y^2 = x^3 + ax + b$$

It turns out that these are about the hardest equations that are analytically solvable in a finite field, and so their manipulation makes for some very knotty operations cryptographers can use to make functions that are difficult to invert [20].

The form that those functions take is grounded in the *geometry* of these solutions: we can prove that for any two points, we can find exactly one point that lies on the same line as them. That is called point *addition*, and is the "group operation" of Elliptic Curves. For the edge cases, such as adding a point to itself, or adding a point to the null "point at infinity", we generally just find another point on the *tangent* line, which does drop out during the derivation of the addition equations, although ECC libraries do generally implement an optimized "doubling" operation [9].

A relatively simple geometric proof can show us that the point-addition operation is commutative and associative, and so the curve forms a *group*; and also if we define scalar multiplication as the simple repetition of that operation, we find that the curve is also a *field*, the core mathematical structure of number theory [20].

It is worth noting that there are a cornucopia of different mathematically equivalent *representations* of these points. For example, the algorithm my pipeline targets represents curves in *Twisted Edwards Form*:

$$ax^2 + y^2 = 1 + dx^2y^2$$

As an example, the point addition formula for these curves is [7]:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

This is one of the functions we would like to synthesize operations for, so we need to be able to represent basically all of the elementary finite field operations: addition, multiplication, and modular inversion (i.e. division on a finite field). The first two are relatively simple, but the third is less so, and – even with a highly sophisticated implementation strategy – will require me to also support conditionals (though the code will be much larger than if we did not).

2.1.3 Discrete Logarithms

One of the core ideas in public-key cryptography is the hardness of computing *discrete logarithms*, or finding a function that (with nontrivial probability) performs the

transformation:

$$kG \to k$$

Or, alternatively, a function that can perform scalar multiplication by k, even if k itself is unknown. There are a variety of formal definitions, but they all have that general flavor, and are usually described under the heading of the Discrete Logarithm Problem (DLP) [20].

The intuition behind the hardness of the problem is essentially that the geometry of the curves as described above is about as complex as is possible for the solution of diophantine equations; and solving equations involving exponentiations of those operations should be, well, exponentially harder.

However, the crux of the argument here is that basically, in addition to ordinary point addition (and subtraction for the purposes of function inversion), we care a lot about supporting scalar multiplication. It may seem that that means that we need *loops*, but we can actually simplify that away if we manually test each bit in k:

```
current = G
if k & 1 > 0: result += current
current *= 2
if k & 2 > 0: result += current
...
```

Again, this means that our sequence of operations will be very long, but should nonetheless only need conditional tests and elementary operations.

2.1.4 *EdDSA*

This has been an extremely high-level description; however, it is enough to mostly grok the idea of *EdDSA*, which is that given a hash function H, a group generator G, a private key k and a public key $K \equiv kG$ (as above, it should be hard to get the private key from the public key), we can sign a message m of fixed length by generating a random value r and returning the pair [20]:

$$(rG, r + H(r, K, m)k)$$

The idea is that we multiply the second value by G to get a value

$$(r + mk)G \equiv rG + mkG \equiv rG + mK$$

And anybody with the public key can clearly verify that this identity holds.

I explain this not to simply fill up space, but rather to emphasize the fact that *this really is just math*, and implementing the elementary curve operations will actually yield usable high-level cryptography.

2.2 Functional Primitives

Note: unless otherwise noted, this section is mostly information that can be found in *Certified Programming with Dependent Types* [13], though this explanation is interlaced with my own commentary.

Due to the extreme mathematicization of this problem, we will need to deal with a so-called *functional* language, which represents programs in a way that should feel very familiar to the mathematically inclined. For example, the core "option" type in OCaml:

```
type 'a option = None | Some of 'a
```

This means that the resulting type is either the placeholder "None" or a "Some" wrapper around the Option's argument type "a"; so this effectively represents the *union* of the "None" space and the "Some" space, which is effectively that of the type argument a.

Walking through this may seem overly pedantic; but the idea that we can represent types as combinations of disjunctions (i.e. "None | Some") and conjunctions (i.e. "Some a"), along with the functionalization of the type (via the type parameter a) will allow us to perform the traditional mathematical operation of taking this to its logical conclusion:

```
Inductive exp : Set :=
| Var : string → exp
| App : exp → exp → exp
| Abs : string → exp → exp.
```

This is actually a complete specification of untyped lambda calculus [13], described as a disjunction of the "constructors" of the language: any term is either a variable ("Var"), named by a string; a function application ("App"); or a lambda, which defines a new variable.

The language there is Coq, an extraordinarily literal language where "Inductive" means that the type is specified *inductively*, or in terms of itself, just as inductive problems reduce to their own instances; and we do explicitly say that the result is a "Set".

So there is some indication that we can represent programs in this framework, but lambda calculus does not lend itself to our ordinary understanding of programming as a sequence of operations. For that, we need a *let binder*:

```
| Let : string → exp → exp → exp
```

The idea being that it "Lets" the variable named by its first argument be equal to the evaluation of its second argument in the evaluation of its third argument, and so effectively represents *variable assignment*.

2.2.1 Dependent Types

The major problem with this tidy story is that we have not yet dealt with *types*, and so can represent all sorts of invalid programs; but there is no way to *constrain* those constructors to only allow valid programs.

The way traditional programmers deal with this hole is to write a *type-checker* after the fact, but formal verification generally takes a different tack, that we should be able to represent types *in the language*.

We do this by using *dependent types*, which can represent a dependency between the type and its argument.

```
Inductive type : Set :=
| Value : type
| Fun : type → type.

Inductive exp : type → Set :=
| Var : forall t, string → exp t
| App : forall a b, exp (Fun a b) → exp a → exp b
| Abs : forall a, string → exp b → exp (Fun a b).
```

There, the "type" of an expression is either a "Value" (like "5"), or a lambda from an input to an output, and we can label the constructors with these types *directly*, removing the possibility of type inconsistency.

These "forall" dependencies – which work like the ordinary mathematical $\forall$ – and their attendant variable dependencies are the *dependent types* most languages cannot represent, and, as we shall see, are the extremely powerful feature of languages like Coq that allows us to think about proofs.

2.2.2 Higher-Order Abstract Syntax

But for now we should deal with the major hole in even that, much nicer explanation: *the strings*. How can we make sure that there are no undefined variables? Well, we simply make the variables a function of our types:

```
Parameter var : type → Set.

Inductive exp : type → Set :=
| Var : forall t, var t → exp t
| App : forall a b, exp (Lam a b) → exp (Lam a) → exp (Lam b)
| Abs : (var a → exp b) → exp (Lam a b).
```

This is a typed variant of *weak Higher-Order Abstract Syntax* (weak HOAS), which will be refined into *parametric HOAS* [12], the foundation of the language types in my pipeline.

2.3 Formal Verification

Now that we have gone through a lightning tour of the functional underpinnings of the representation of languages, we can talk about how we end up with actual *verification*.

2.3.1 Representations of Proofs

Dependent types are very powerful things, and it turns out that with that innocent-looking "forall" constructor described above, we can actually think about types for expressions like:

$$\forall x, x + x = 2 * x$$

Using just the tools described above, we can write this as an *equivalence relation*:

```
Inductive Equals : forall {t}, t → t → Prop :=
  | Refl : forall x, Equals x x.
```

This does not seem particularly useful, though, because we could only represent "proofs" that $x = x$. The way that Coq deals with that fact is by enabling *replacement*, in the sense that we can *replace* $x + x$ with $2 * x$ if the expressions are equivalent.

We can think of how such expressions could be equivalent – the multiplication algorithm would simply repeatedly add x – but how would Coq know that?

2.3.2 Reduction

Well, it turns out that the high-level language in Coq (called "Gallina") is itself *formally specified*, and has symmetries that we can use. In order to convert $2 * x$ into $x + x$, we could perform a sequence of simplifications:

$$\begin{aligned}
2 * x &\to \text{mul } (S(SO))\ x \\
&\to x + \text{mul } (SO)\ x \\
&\to x + x + \text{mul } O\ x \\
&\to x + x + O \\
&\to x + x && \text{if x is instantiated}
\end{aligned}$$

All of these are at the language level, in the sense that neither the integer type nor the multiplication or addition procedures need to know about what simplifications are happening. These operations are called *reductions* and are some of the closest things Coq has to primitives. Actually, each of the simple steps above was a combination of *two*, the ι and β reduction steps.

When I say "reduction", though, I am usually not referring to the individual operations. Coq provides a variety of "reductions" that can simplify expressions like $2 * x$ by repeatedly applying these simplifications, such as *cbv* and *simpl*.

It is important to mention here that these reductions are far more complex than the above explanation makes it seem: they can automatically expand definitions, choose the specific types of reduction steps to take, etc. My pipeline makes a lot of use of some very customized reductions, which do an excellent job of regularizing my expressions.

2.3.3 Tactics

Tactics arise from the observation that many proofs look about the same, such as the proof of $2 * x$ and $3 * x$, or indeed, showing equivalence of any polynomial expression. Abstractly, both are some combination of the application of lemmas, "rewrites" as mentioned above, and the elemental reductions.

Coq provides *Ltac*, an embedded language specifically designed for manipulating Gallina, and, by extension, these proofs. I will not go into the details here, but suffice to say that we can *programmatically* construct proofs, which, as with other forms of computer automation, makes proof-construction immensely more efficient.

2.4 Domain-Specific Techniques

That is all the background I need to be able to explain the techniques used in my pipeline at a high level, save for some implementation details:

2.4.1 Sigma Types

Some of the objects I will be dealing with are too complex to be able to specify in terms of a grammar; so I use pervasive *sigma types*, which store a value *along with* a proof about it:

$$\Sigma(T, P) \equiv \{x : T | P(x)\}$$

For example, I use such a type to store a value along with a proof that it is above and below certain numbers, thereby creating a type that *requires* that its value be in a certain range.

2.4.2 *Bedrock* Word Representation

I will use a representation of machine words from a library called *Bedrock*, which provides words of a fixed size (i.e. the type "word 32" represents 32-bit words, another dependent type) and implements many annoying low-level operations, such as bitwise operations [14].

The rub here is the enormous amount of time I spent trying to prove compatibility between the Bedrock words and the Coq integers, which differ in enough subtle ways to make the conversion extremely tricky.

2.4.3 Reification

One of the key points of my pipeline is that its input is *implemented in Coq*. This is not a domain-specific language; my input actually uses the Coq integer types, making the system optimally easy to integrate with other tools.

The way that works is via a *reification tactic*: essentially, I have a function whose result is the sigma type $\{x : AST \mid \text{interp}(x) = input\}$, and then use an *existential quantifier* to prove $\text{interp}(x) = input$ for an unknown x.

The details of how that works are a bit difficult to explain, but essentially we start off with a proof with a wildcard in the field for x, and a tactic "fills it in" by traversing the Gallina syntax tree for the input and manually constructing a corresponding AST. This is another sophisticated technique only possible in formally specified languages such as Coq, though TemplateHaskell provides some similar, proof-less facilities [25].

2.4.4 Extraction

Because code written in Coq is often intended to interact with the outside world, but there is no way to call Coq code from external tools or vice-versa, we use a technique called *Extraction*, in which the Coq compiler traverses whatever expression you give it, and tries its best to convert it to OCaml or Haskell, removing any unrepresentable proof terms. In my case, at the end of the pipeline, I was dealing with strings, which are easily extractable.

Chapter 3

Overarching Proof Strategy

With our general understanding of Coq, we can now go on a tour through the general strategy of my Qhasm pipeline: I assume that an input program takes the form of a sequence of arithmetic expressions in *Let binders*, like so:

```
Definition example (a b: Z) :=
  let x0 := a + b in
  let x1 := someFunction (x0 + a) in
  (x0 + x1, x1 + b)
```

That looks kind of like an imperative program we might be able to work with, but it would be much more convenient to have the functions come one at a time, to unwrap any function calls, and also to convert the intrinsic Lets to our our own function (so that Coq does not automatically simplify it). That is done by our first *canonicalization* stage, which outputs code like this:

```
Definition example (a b: Z) :=
  LetIn (a + b) (\x0 →
  LetIn (x0 + a) (\x1 →
  LetIn (x1 + c) (\x2 →
  (x0 + x2, x2 + b)))
```

This is regular enough that we can convert it into a syntax tree via *reification* tactics, our second step, and we will end up with a *high-level syntax tree* like this:

```
Definition example (a b: Z) :=
  Let (Binop OPadd (Const a) (Const b)) (\x0 →
  Let (Binop OPadd (Var x0) (Const a)) (\x1 →
  Let (Binop OPadd (Var x1) (Const c)) (\x2 →
  Pair (Binop OPadd x0 x2) (Binop OPadd x2 b))))
```

Note that this implementation can only reify a fixed set of operations (addition, subtraction, multiplication, bitwise AND, and right shifts) (though *Fiat-crypto* can do much more).

The interesting thing about this high-level syntax is that (via some magic we will explore in the coming sections) we can transparently switch out the argument type of the AST from Z to machine words, with a syntax tree that looks the same. Doing that directly is not particularly useful; however if we use a custom *Bounded Word* type, we can show that none of the operations ever overflow, and therefore the AST over words is always equivalent to the AST over integers. This sounds like simple abstract interpretation; however, as I will explain in the coming sections, it turned out to be much more complex.

Even this high-level syntax, however, does not map cleanly onto instructions, so we need to compile it into a simpler *low-level* syntax type, which looks like:

```
Definition example (a b: word 64) :=
  LetBinop OPadd (Const a) (Const b) (\x0 →
  LetBinop OPadd (Var x0) (Const a) (\x1 →
  LetBinop OPadd (Var x1) (Const c) (\x2 →
  LetBinop OPadd (Var x0) (Var x2) (\x3 →
  LetBinop OPadd (Var x2) (Const b) (\x4 →
  Pair x3 x4)))))
```

At this point, we have something that we could compile using C or directly via LLVM, and so that was one of the first-order deliverables of this project. For that reason, I actually have a sequence of *reductions* that let us perform correctness proofs here, which is important because showing correctness at the assembly level was computationally intensive enough that I could not do it for the full target syntax trees.

From here, we can compile directly into an embedded assembly language:

```
Definition example := [
  QOp (IOpReg IAdd (Reg W64 2) (Reg W64 0) (Reg W64 1));
  QOp (IOpReg IAdd (Reg W64 3) (Reg W64 0) (Reg W64 2));
  QOp (IOpStack IAdd (Reg W64 4) (Reg W64 3) (Stack W64 0));
  QOp (IOpReg IAdd (Reg W64 5) (Reg W64 2) (Reg W64 4));
  QOp (IOpStack IAdd (Reg W64 6) (Reg W64 4) (Stack W64 1));
  QAssign (ARegReg (Reg W64 0) (Reg W64 5));
  QAssign (ARegReg (Reg W64 1) (Reg W64 6)) ]
```

Note that this language is a nearly complete formal semantics of Qhasm, even if the present case is relatively simple, and the approach is flexible with respect to preference for stack allocation, the number of registers available, etc. From there, we simply convert to a string and extract the implementation to OCaml, which yields a file which we pass to the Qhasm binary.

Now that the basic steps are clear, we can dig into the details:

3.1 Canonicalization of Input Expressions

The first pipeline segment is a series of *reductions* that simplify input expressions into a form consistent enough for me to traverse during the subsequent reification procedure. This will likely seem arbitrary, as the steps needed to get the expressions into a usable form is highly dependent on my input, which has changed over time, and the form of which is totally dependent on code outside the scope of this paper (of the elliptic-curve variety). In any case, the algorithm is generally:

1. Convert all Let-binders and intrinsic integer operations into opaque functions.

2. *cbv* the resulting expression, unfolding any definitions I do not intend to reify.

3. Curry any tupled parameters to make the function of an *NAry* type.

Step 1 intends to make every primitive operation in the resulting syntax tree opaque, so that when we do reify it, the steps are clear. Step 2 is the actual simplification, which performs all of the reductions intrinsic to Coq, unfolding external definitions. Step 3 is a bit more complex: in order to make the syntax-tree types simpler, I defined a simple wrapper for curried types:

```
Fixpoint NAry (n: nat) (A: Type) (B: Type): Type :=
  match n with
  | (S m) ⇒ A → NAry m A B
  | O ⇒ B
  end.
```

The idea here is just to make code manageable when we have functions with upwards of 30 parameters (as primitive Elliptic Curve operations do). A small library of tactics performs the necessary conversion.

On a side-note which is possibly relevant for other integrations, toward the beginning of the project, this step was quite a lot more complex, because the input was in terms of a custom Galois Field sigma type:

$$GF\ n := \{x : Z | x \equiv x (mod\ n)\}$$

for which I had implemented a set of algebras in parallel to the intrinsic integer type. That was a major mistake with regard to performance, because the sigma types will carry around a *proof* of the second term, which can get extremely complex when the syntax trees are large or involve bitwise operations; so the job of the canonicalization step is to project those out. That is the real power of this kind of automatic, verifiable reduction.

3.2 Reification into High-Level Syntax

Now that we have a syntax tree in the correct format, we can use a *reification tactic* to produce our syntax tree. The logic here is simple:

- If we see a integer value x, and that was produced by one of our let-binders, then make it a *Var*, and otherwise make it a *Const*.
- If we see an expression of the form $a + b$ (or any other of the primitive operations), then convert it to a *Binop*.
- If we see an expression of the form (a, b), then convert it to a *Pair*.
- If we see a *LetIn*, then convert it to a *Let*.

The tactical implementation is admittedly opaque and abstruse, but the concept is just that simple. The real complexity here actually comes from the high-level syntax type, which has one small wrinkle:

```
| Const : forall {_ : Evaluable T}, ·interp_type T TT → expr TT
```

In Coq syntax, this means that constants carry around an *Evaluable* instance, which is a type-class I use to derive the primitive operations of the argument type of the syntax tree. This fact will allow me to switch out the Evaluable instances in each constant to change the type of the whole tree.

Looking back on it, a much better approach here would have been, instead of muddying up the language type with a trick, to make a smarter reification procedure that could have produced syntax trees with arbitrary argument types, along with a proof that the interpretation of the syntax tree is equivalent to the input (and likely a well-formedness predicate as well). Admittedly, at the time, I did not have the understanding or proof-engineering sophistication to do that effectively.

3.2.1 Boundedness Proofs

So we can convert these syntax trees between integer types, and therefore directly to machine words; however, it would be very easy to lose correctness there because whenever operations overflow, we would violate the desired integer semantics.

The solution here is to cleverly design your input so that you *think* none of the operations will overflow, then to prove it with a *boundedness proof*. The way my implementation works is to prove boundedness by converting a word-valued AST to a BoundedWord-valued AST, then simply evaluating it:

```
Record BoundedWord {n} := bounded {
  bw_low: N;
  bw_value: word n;
  bw_high: N;

  ge_low: (bw_low <= wordToN bw_value)%N;
  le_high: (wordToN bw_value <= bw_high)%N;
  high_bound: (bw_high < Npow2 n)%N
}.
```

The implementations of primitive operations over this type require that we have proofs that both the arguments and the results are always in-bounds; so if we evaluate the BoundedWord syntax tree and get *any result at all*, we will have a proof that no operations overflow.

It may seem that this is an overly pedantic consideration, because most of the code people write does not come close to integer bounds; however, cryptographic implementations tend to operate *very close to the wire*, so as to use as few registers as possible, and perform as few carry steps as possible. For this reason, it would be well-nigh impossible to get within a factor of 100 of solid, human-written assembly implementations of finite-field cryptography without dealing with bounds-checking.

The only problem with this approach is the relative complexity of evaluating the syntax tree over BoundedWords. To deal with this, I attempted to prove correct a parallel set of lemmas over a simpler type:

```
Record RangeWithValue := rwv {
  rwv_low: N;
  rwv_value: N;
  rwv_high: N
}.
```

And indeed, the relevant proofs are much faster; however I was unable to prove that using RangeWithValue was equivalent to using BoundedWords; although I maintain that it should be possible, the proof scripts quickly became infeasibly complex.

3.3 Compilation into Low-Level PHOAS Syntax

Having proven everything we care about in terms of the high-level syntax type, we can compile it to the lower-level syntax type that looks like a real sequence of operations, and will be easy to send to any C compiler.

3.3.1 Side-Channel Equivalence Proof

In terms of the low-level integer-valued syntax tree, we can make an equivalence proof that it performs the same function as our initial AST. The strategy is relatively simple:

1. Interpret both expressions, unfolding every definition.

2. Make all let-binders opaque.

3. *cbv* both expressions.

And at the end, both sides will be the same. The reason we have to deal with let-binders again is because it is possible to have exponential blowup during the final reduction. Though it is possible to work through this with a powerful computer, because this compilation procedure is not doing anything too complicated, there will not be differences in Let-structure. This exact complexity is why I did not make end-to-end proofs with interpretations of my Assembly.

The hole in the correctness proof here is that it operates in terms of *integer-valued* syntax trees: it would be much better to use the final machine-word syntax trees. I never integrated it into this implementation, but during the development of *Fiat-Crypto*, I proved compatibility of the primitive operations with conversion to/from machine words (given some bounds); and so it is definitely possible to make this a true end-to-end proof.

3.4 Compilation into *Qhasm*

Now that we have a real list of operations in terms of machine words, we can perform the relatively uninteresting task of assigning registers and stack locations to every variable. I will deal with this in a later section, and the general algorithm is simple:

> For each instruction we see, keep track of a list of every variable that is used in any *lower* line. When a register-allocated variable is removed from this list, free its register for use later.

There are many possible optimizations here, some of which I made attempts at reasoning about, including searching through the possible implementation space by swapping instructions, although I did not get the opportunity to explore the topic deeply. The simple approach was good enough to allow me to work on other things.

3.5 Extraction and OCaml Driver

Now my code performs the unenviable task of using the intrinsic string manipulation library in Coq to yield machine code as a large string. Because Coq does not have the facility to materialize files, the pipeline generates an OCaml implementation of the string, which an external OCaml "driver" passes directly to Qhasm.

Chapter 4

Automatic Bounds-checking

Clearly, much of the pipeline as described is not so interesting, so I will not spend a huge amount of time on it – the first place where the implementation starts getting nuanced is in the verifiable bounds-checking procedure.

Here, again, the idea is that we have a syntax tree over integers, and we would like to turn that into a syntax tree over machine words. In practice, converting the two is easy – you simply interchange the types – however, part of the point is *proof*, and proving that the result does not misbehave is more complex.

4.1 Evaluable Model

First, we should get clear about exactly the interface that we expect each integer type to implement, i.e. the set of operations we may have to propagate bounds through. In the Qhasm pipeline, this is represented as the *Evaluable* type-class.

```
Class Evaluable T := evaluator {
  ezero: T;

  (* Conversions *)
  toT: option RangeWithValue → T;
  fromT: T → option RangeWithValue;

  (* Operations *)
  eadd: T → T → T;
  esub: T → T → T;
  emul: T → T → T;
  eshiftr: T → T → T;
```

```
  eand: T → T → T;

  (* Comparisons *)
  eltb: T → T → bool;
  eeqb: T → T → bool
}.
```

There are class instances for integers, words of arbitrary width, bounded integers, etc. Most of the operations are clear, with the only hiccup being the conversions. Those let us convert *between* Evaluable types through the medium of *RangeWithValue*, defined as:

```
Record RangeWithValue := rwv {
  rwv_low: N;
  rwv_value: N;
  rwv_high: N
}.
```

The idea being that for each type we use, the conversions to and from RangeWithValue are simple and easy to prove things about. Indeed, we will need to programmatically remove a lot of them.

4.2 Bounded-Integer Types

The most important Evaluable is the *BoundedWord* type, defined as such:

```
Record BoundedWord {n} := bounded {
  bw_low: N;
  bw_value: word n;
  bw_high: N;

  ge_low: (bw_low <= wordToN bw_value)%N;
  le_high: (wordToN bw_value <= bw_high)%N;
  high_bound: (bw_high < Npow2 n)%N
}.
```

Here, the BoundedWord not only carries around bounds, but also, confusingly, a value and proofs that the value is between the desired bounds. Some important things to note:

- We do not always have to have a value: during many of our computations, we will leave `bw_value` as a *free variable*, or rather a variable that Coq does not know anything about. Even with no knowledge of `bw_value`, we can still get

a lot of information about the bounds because they are computed separately (and lazily).

- Because you can perform operations on BoundedWords that cannot yield a result, the Evaluable instance is actually on `option BoundedWord`, which obfuscates the implementation significantly.

- Due to the non-negativity of N and `high_bound`, we implicitly have a guarantee that `bw_value` will always fit in a word, so why would we not keep it as a N for simplicity? Well, when we end up reducing a BoundedWord, this means that the value has a syntax tree involving *only words*, which can be compared directly with that of the strictly word AST.

That second point brings up a more important question: how do we square operations on words with operations on BoundedWords? The answer lies in the *validBinaryWordOp* predicate:

```
Definition validBinaryWordOp
     (rangeF: Range N → Range N → option (Range N))
     (wordF: word n → word n → word n): Prop :=
  forall (low0 high0 low1 high1: N) (x y: word n),
    (low0 <= wordToN x)%N →
    (wordToN x <= high0)%N →
    (high0 < Npow2 n)%N →
    (low1 <= wordToN y)%N →
    (wordToN y <= high1)%N →
    (high1 < Npow2 n)%N →
    match rangeF (range N low0 high0)
                 (range N low1 high1) with
    | Some (range low2 high2) ⇒
       (low2 <= ·wordToN n (wordF x y))%N
      ∧ (·wordToN n (wordF x y) <= high2)%N
      ∧ (high2 < Npow2 n)%N
    | _ ⇒ True
    end.
```

There, we correlate a function over words with a function over ranges; and a collection of my lemmas show that any `validBinaryWordOp` can be converted to a function over BoundedWords whose update on the value looks like `wordF` and whose update on the enclosing ranges looks like `rangeF`. This is `bapp`:

```
Definition bapp {rangeF wordF}
```

```
(op: ·validBinaryWordOp n rangeF wordF)
(X Y: BW): option BW.
```

This problem factorization allows us to painlessly project out different pieces of the BoundedWord, which is useful in conversions. Needless to say, all of the applicable Evaluable operations have corresponding `validBinaryWordOp` lemmas with relatively tight bounds, for example `range_and_valid`:

```
Lemma range_and_valid :
    validBinaryWordOp
      (fun range0 range1 ⇒
        match (range0, range1) with
        | (range low0 high0, range low1 high1) ⇒
          let upper :=
            (N.pred (Npow2
              (min (N.to_nat (getBits high0))
                   (N.to_nat (getBits high1))))))%N in
          Some (range N
            0%N
            (if (Nge_dec upper (Npow2 n))
             then (N.pred (Npow2 n))
             else upper))
         end)
      (·wand n).
```

And the Evaluable instance is mostly combining these with `bapp`.

Underlying Proofs About Words

Although it is not visible from a high level and also relatively uninformative, I would like to mention the literally hundreds of hours I spent working on seemingly simple lemmas about Bedrock words such as these:

```
Lemma wordize_shiftr: forall {n} (x: word n) (k: nat),
  (N.shiftr_nat (wordToN x) k) = wordToN (shiftr x k).

Lemma wordize_and: forall {n} (x y: word n),
  wordToN (wand x y) = N.land (wordToN x) (wordToN y).

Lemma wordize_or: forall {n} (x y: word n),
  wordToN (wor x y) = N.lor (wordToN x) (wordToN y)
```

If there is one thing that came out if this project, it is my now extremely solid understanding of Coq's infinite twos-complement integer representation.

4.3 Bounds-checking from BoundedWords

Now that we have a type with proofs, we would like to be able to show that no operations overflow: that fact is very intuitive on the level of individual operations; however, we do have to re-frame the problem a bit:

> *Our conversion to words is correct iff it computes exactly what the integer program would have computed.*

Put this way, the logical implication is clear: we have to show that converting the inner word AST of the BoundedWord program to integers is exactly equivalent to the integer program. This is the lemma:

```
Lemma RangeInterp_bounded_spec: forall {t} (E: ·expr Z Z t),
  bcheck (f := ZtoB) E = true →
  typeMap (fun x ⇒ NToWord n (Z.to_N x)) (zinterp E)
    = wordInterp (ZToWord _ E).
```

That is real code, and so is a bit convoluted, but what this means is that the word AST is equivalent to the integer AST if running `bcheck` on the AST yields true. But `bcheck` basically just runs the BoundedWord program and makes sure that there is a result in the end.

As I mentioned in passing before, it turns out that `bcheck` is fairly time-intensive to evaluate (or, at least, my strategy for its evaluation is unsophisticated). I have made a number of attempts at making a more optimized alternative, but I did not manage to prove that any of them correspond exactly.

Chapter 5

Intermediate PHOAS Compilers

Once we have a word-valued high-level syntax tree with valid elementary operations, we need to start the long slog of converting it into actually executable machine operations: as described previously, the pipeline does this via a sequence of verified compilers on Parametric Higher-Order Syntax (PHOAS) types, at varying levels of abstraction. I can start this process by re-introducing the previously mentioned high-level syntax type in more detail:

5.1 High-Level Assembly

5.1.1 PHOAS Model

The primary difference between PHOAS and the weak HOAS type mentioned in the technical introduction is the *parametric* part, i.e. the fact that the variable type mapping itself is a parameter of the expression type. In Coq, this difference is not very apparent because we can implicitly add arguments using *Section variables* and end up with a fairly simple type structure:

```
Context {T: Type}.

Section expr.
  Context {var : type → Type}.

  Inductive expr : type → Type :=
    | Const : forall {_ : Evaluable T}, ·interp_type T TT → expr TT
```

```
    | Var : forall {t}, var t → expr t
    | Let : forall {tx}, expr tx → forall {tC}, (var tx → expr tC) →
      expr tC
    | Binop : forall {t1 t2 t3}, binop t1 t2 t3 → expr t1 → expr t2 →
      expr t3
    | Pair : forall {t1}, expr t1 → forall {t2}, expr t2 → expr (Prod
      t1 t2)
    | MatchPair : forall {t1 t2}, expr (Prod t1 t2) → forall {tC}, (var
      t1 → var t2 → expr tC) → expr tC.
End expr.
```

This looks *kind of* like the HOAS type, except that it does not even attempt to model function types, rather preferring to add constructors for a fixed set of functions, specifically binary operators, tupling (referred to here as "Pair"), and pattern-matching (here, the inverse of tupling). We also add a "Let" constructor to allow for expression composition.

The other difference here is the "Const" constructor, which takes the interpretation of the single-value type (at the reification stage, just an integer), and wraps it as an expression. But it also has a seemingly unused Evaluable instance: this is a bit of a trick to sneak in bounds on our input variables *inside* the Const constructor, without having to materialize an AST of a different type. As mentioned, this strategy was a mistake, and added quite a lot of complexity: it would have been far easier to simply reify into differently typed syntax trees.

One of the things that PHOAS lets us do is to actually hold the variable type parameter *undefined*: because Coq operates mostly in terms of lemma application, rewrites, and reductions, we can do all of that without ever bothering to specify the form the variable takes, and only fill it in when necessary, although, as it turns out, it never ends up being really necessary. The added benefit there is that because there actually is no variable type, Coq never tries to prove anything about it, which makes the relevant proof terms quite a lot simpler [12].

This allows us to make some types with the variables abstracted that end up almost legible:

```
Definition Expr t : Type := forall var, ·expr var t.

Definition Interp {t} (e: Expr t) : interp_type t := interp (e
    interp_type).
```

```
Example example_Expr : Expr TT := fun var ⇒ (
  Let (Const 7) (fun a ⇒
    Let (Let (Binop OPadd (Var a) (Var a)) (fun b ⇒ Pair (Var b) (Var
        b))) (fun p ⇒
      MatchPair (Var p) (fun x y ⇒
        Binop OPadd (Var x) (Var y))))))%Z.
```

5.1.2 Reification Strategy

Having seen the underlying type we intend to reify into, we can deal with the reification strategy itself. This will involve a bunch of *Ltac*; but I will elide the details, and hopefully the code is clear enough that a thorough explanation is not necessary.

```
Ltac reify n var e :=
  lazymatch e with
  | let x := ?ex in ·?eC x ⇒ ...
  | match ?ep with (v1, v2) ⇒ ·?eC v1 v2 end ⇒ ...
  | pair ?a ?b ⇒ ...
  | ?op ?a ?b ⇒ ...
  | (fun x : ?T ⇒ ?C) ⇒ ...
  | ?x ⇒ ...
  end.

Ltac Reify_rhs n :=
    let rhs := rhs_of_goal in
    let RHS := Reify rhs in
    transitivity (ZInterp (RHS n));
    ...

Lemma test : {v | ZInterp v = (let x := 1 in let y := 2 in x * y)%Z}.
  eexists.
  Reify_rhs 32.
  reflexitivity.
Abort.
```

Here, we can see that the reification procedure operates by *directly matching* on Gallina, and the terms of that match are almost exactly the constructors of the high-level syntax (plus one which unwraps the variable functions described above).

5.1.3 Conversion of Argument Types

One of the declared features of the high-level type is that you can convert its argument type via a long and arduous *convertExpr* function, which I will elide for readability, but has the following type signature:

```
Context {A B: Type} {EB: Evaluable B}.

Fixpoint convertExpr {t v} (a: expr (T := A) (var := v) t): expr (T :=
    B) (var := v) t.
```

Essentially what it does is map a conversion function into the Const constructor, and juggles the dependent types to make the output expression well-formed.

The major issue with this design is that for every conversion it performs, it injects an instance of the *toT* and *fromT* functions, meaning that in order to show that the result is equal to what we expect, we need to have a *very good* logic over those operations. This was a continual sticking point for my development, and the resulting rewrites made my proofs about the function orders of magnitude more complex. I had been trying to avoid injecting more tactics into the pipeline for performance reasons; but really, this just shifted the complexity onto correctness proofs.

5.1.4 Boundedness Proofs

Once we have conversion operations, we can convert our integer syntax tree to a BoundedWord syntax tree as per the previous chapter. The difference now is that we can understand how each constant becomes a BoundedWord: by the same *toT* and *fromT* functions in the Evaluable instance in the Const constructor.

For the in-pipeline execution of the boundedness check, I actually do just apply the program on some list of bounds, check that all results exist, and yield bounds for each output, with the idea that we will be chaining operations together, and we can propagate them to the following operation:

```
Definition prog := Util.applyProgOn upper Input.inputBounds progR.
Definition valid := check (n := bits) (f := id) prog.
Definition output :=
    typeMap (option_map (fun x ⇒ range N (rwv_low x) (rwv_high x)))
           (LL.interp prog).
```

5.1.5 Compilation Procedure

With that out of the way, we can compile the high-level assembly type to a lower-level type via a simple function with an associated correctness lemma:

```
Fixpoint compile {T t} (e:·HL.expr T (·LL.arg T T) t) : ·LL.expr T T t.

Lemma compile_correct {_: Evaluable T} {t} e1 e2 G (_: wf G e1 e2) :
   List.Forall (fun v ⇒ let 'existT _ (x, a) := v in LL.interp_arg a =
      x) G →
      LL.interp (compile e2) = HL.interp e1 :> interp_type t.
```

There is a little bit of complexity here I tried to avoid in other sections around the application of a *well-formedness* "wf" predicate, which essentially exists to "line up" variables from the input expression to variables in the output expression; but at the level of input expressions, G will be empty, so the "List.Forall" term is always true and we know that the output expressions are functionally equivalent to the input expressions; we just cannot prove that analytically.

The actual compilation procedure just performs induction over the high-level type and produces an elaborated low-level syntax tree.

5.2 Low-Level Assembly

5.2.1 PHOAS Model

The low-level syntax model looks a lot like the high-level model, but is far more restrained in where it allows us to perform operations.

```
Context {T: Type}.
Context {E: Evaluable T}.

Section expr.
  Context {var: Type}.

  Inductive arg : type → Type :=
    | Const : ·interp_type T TT → arg TT
    | Var : var → arg TT
    | Pair : forall {t1 t2}, arg t1 → arg t2 → arg (Prod t1 t2).

  Inductive expr : type → Type :=
    | LetBinop : forall {t1 t2 t3}, binop t1 t2 t3 → arg t1 → arg t2 →
       forall {tC}, (arg t3 → expr tC) → expr tC
    | Return : forall {t}, arg t → expr t.
End expr.
```

First of all note that here *var* is not a function of type: *arg* takes that role now, and provides transparent constructors that users can match on. Where the high-level

expressions deal with pairing by “Pair” and “MatchPair” constructors in the syntax type, the low-level expressions do it by proxy by using this argument type and letting its consumers perform manual matching.

There is not a tremendous amount to say here, except that this type is isomorphic to a list of instructions: “LetBinop” is effectively the “cons” constructor and “Return” is effectively the “nil” constructor.

Because we now have a list of elemental operations, we can turn it into a list of instructions, and the conversion will be *about* one-to-one, modulo some register juggling.

Chapter 6

Compilation to Qhasm

Now that we have our desired instructions as a simple list, we can convert it to an assembly model, and then finally to a string that we will send to external compilation tooling.

6.1 Type Model

My assembly types are fairly simple, and the meat of the design looks like this:

```
Inductive Assignment : Type :=
  | ARegMem: forall {n m}, Reg n → Mem n m → Index m → Assignment
  | AMemReg: forall {n m}, Mem n m → Index m → Reg n → Assignment
  | AStackReg: forall {n}, Stack n → Reg n → Assignment
  | ARegStack: forall {n}, Reg n → Stack n → Assignment
  | ARegReg: forall {n}, Reg n → Reg n → Assignment
  | AConstInt: forall {n}, Reg n → Const n → Assignment.

Inductive Operation :=
  | IOpConst: forall {n}, IntOp → Reg n → Const n → Operation
  | IOpReg: forall {n}, IntOp → Reg n → Reg n → Operation
  | IOpMem: forall {n m}, IntOp → Reg n → Mem n m → Index m →
      Operation
  | IOpStack: forall {n}, IntOp → Reg n → Stack n → Operation
  | DOp: forall {n}, DualOp → Reg n → Reg n → option (Reg n) →
      Operation
  | ROp: forall {n}, RotOp → Reg n → Index n → Operation
  | COp: forall {n}, CarryOp → Reg n → Reg n → Operation.

Inductive QhasmStatement :=
  | QAssign: Assignment → QhasmStatement
  | QOp: Operation → QhasmStatement
  | QCond: Conditional → Label → QhasmStatement
```

```
  | QLabel: Label → QhasmStatement
  | QCall: Label → QhasmStatement
  | QRet: QhasmStatement.

Definition Program := list QhasmStatement.
```

The mapping to assembly operations is relatively clear: assignments are all `mov` instructions, and the operations are `add`, `sub`, `mul`, etc.

The wrinkle here is actually in the evaluation, for which I use the "Operational Semantics" method of defining evaluation as an instance of an inductive type:

```
Inductive QhasmEval: nat → Program → LabelMap → State → State → Prop
    :=
  | QEOver: forall p n m s, (n > (length p))%nat → QhasmEval n p m s s
  | QEZero: forall p s m, QhasmEval 0 p m s s
  | QEAssign: forall n p m a s s' s'',
      (nth_error p n) = Some (QAssign a)
    → evalAssignment a s = Some s'
    → QhasmEval (S n) p m s' s''
    → QhasmEval n p m s s''
  | QEOp: forall n p m a s s' s'',
      (nth_error p n) = Some (QOp a)
    → evalOperation a s = Some s'
    → QhasmEval (S n) p m s' s''
    → QhasmEval n p m s s''
  | QECondTrue: forall (n loc next: nat) p m c l s s',
      (nth_error p n) = Some (QCond c l)
    → evalCond c s = Some true
    → NatM.find l m = Some loc
    → QhasmEval loc p m s s'
    → QhasmEval n p m s s'
  | QECondFalse: forall (n loc next: nat) p m c l s s',
      (nth_error p n) = Some (QCond c l)
    → evalCond c s = Some false
    → QhasmEval (S n) p m s s'
    → QhasmEval n p m s s'
  | QERet: forall (n n': nat) s s' s'' p m,
      (nth_error p n) = Some QRet
    → popRet s = Some (s', n')
    → QhasmEval n' p m s' s''
    → QhasmEval n p m s s''
  | QECall: forall (w n n' lbl: nat) s s' s'' p m,
      (nth_error p n) = Some (QCall lbl)
    → NatM.find lbl m = Some n'
    → QhasmEval n' p m (pushRet (S n) s') s''
    → QhasmEval n p m s s''
  | QELabel: forall n p m l s s',
      (nth_error p n) = Some (QLabel l)
    → QhasmEval (S n) p m s s'
    → QhasmEval n p m s s'.
```

This actually did not work so poorly, because a fairly simple tactic can evaluate programs in this language with no trouble. It is, however, extremely slow, and evaluating syntax trees with free variables quickly became infeasible, so I could not get the same reduction-style correctness to work here.

The alternative approach, a correctness lemma like the one I used compiling high-level syntax, proved too difficult given the intricacies of interacting with my "mapping" type, as describe below, and so this formal specification remains largely unverified, although I was able to show that simple operations translate correctly and that the resulting assembly is valid. A better design would have skipped the register and stack mapping altogether and done the whole conversion in a single function.

6.2 Compiling Low-level Syntax to Qhasm Model

Now we can think about how we would convert our low-level syntax model into assembly, with the general approach:

1. "Fill in" the variable type of the low-level syntax with "mappings", either to registers or stack values.

2. Iterate through each operation and convert it into a Qhasm operation, as above.

3. If we see an operation that is allocated incorrectly (e.g. an attempt to add two stack values into a third stack value), then either reject it or add register or stack values to make it work.

The compilation procedure is parametric over the function that assigns variables to mappings, with the simple alternatives being "every variable is a register" and "every variable is a stack value". The real optimum should be somewhere in-between, and this is an area where the correctness of the algorithm is insensitive to the choice of this assignment, so humans can optimize it, but I did not spend much time trying, preferring rather to focus on the more difficult pipeline stages.

In the third section, we have some configurability. My code by default tries the second option of moving things around to make it work; however, for cleverly

designed input, we should not have to do that. This leaves the door open for a more sophisticated search procedure to find a register assignment that works, but I did not make a serious attempt at working on that.

The "mappings" were another sticking point where my code could certainly have been designed better: they basically choose between several places to obtain each constant or variable:

```
Inductive Mapping (n: nat) :=
  | regM: forall (r: Reg n), Mapping n
  | stackM: forall (s: Stack n), Mapping n
  | memM: forall {m} (x: Mem n m) (i: Index m), Mapping n
  | constM: forall (x: Const n), Mapping n.
```

In reality, this meant that I was constantly unwrapping and re-wrapping these mapping types, and the simple case-switching made my implementation larger and more knotted than it needed to be.

A smarter approach would probably have been to perform the whole transformation in one fell swoop, keeping an online mapping of what registers and stack entries were assigned to which variables, automatically freeing them as their respective variables became irrelevant. This may be less modular, but would have greatly simplified the relevant proofs.

6.3 String Conversion and Extraction

The code that converts the assembly type to a string involves a lot of manual labor and string-manipulation; but is generally unenlightening, so I will not dwell on it. The general procedure is simple:

1. Convert the list of QhasmStatements to a list of strings.
2. Find all input variables and add them as parameters.
3. Record the output register or stack indices so that we can add a handful of move instructions to put them in the right place.
4. Wrap the whole thing as a final Qhasm assembly script in a string.

5. Extract that string to OCaml.

6. In OCaml, materialize the string as a file, then run Qhasm on it.

Note that moving the output register and stack indices is possibly unnecessary, but I never came up with a good way to constrain the synthesis process to obviate the need for that step.

In any case, that about wraps it up: I have the complete pipeline implementation for the elementary operations of Curve25519 attached as a code listing.

Chapter 7

Conclusion

In this thesis, I presented a mechanism for verifiably compiling the integer-valued syntax trees relevant to synthesizing implementations of finite-field cryptography. It transparently interprets programs within the proof assistant Coq, and compiles them to assembly while making sure that no correctness errors can occur, especially concerning integer overflow.

This *Qhasm pipeline* is far too constrained and inefficient to be a viable long-term approach, and is flawed in many ways; however, it is a valuable illustration of the kind of clean, organized, legible proof strategy which reveals a glimmer of hope for the development of really trustable cryptographic implementation.

The fact that I was able to do this points not so much at the technical sophistication of my code – though, of course, my proof scripts were quite complicated – but rather at the extreme advantages of having formally specified algorithms. It is my strong contention that in the future, cryptographic design will happen exclusively via proof assistants like Coq.

Appendix A

Code

```
Module Type Expression.
  Parameter bits: nat.
  Parameter width: Width bits.
  Parameter inputs: nat.
  Parameter inputBounds: list Z.
  Parameter ResultType: type.
  Parameter prog: NAry inputs Z (·HL.Expr Z ResultType).
End Expression.

Module Pipeline (Input: Expression).
  Definition bits := Input.bits.
  Definition inputs := Input.inputs.
  Definition width: Width bits := Input.width.
  Definition ResultType := Input.ResultType.
  Hint Unfold bits inputs ResultType.

  Definition W: Type := word bits.
  Definition R: Type := option RangeWithValue.
  Definition B: Type := option (·BoundedWord bits).

  Instance ZEvaluable : Evaluable Z := ZEvaluable.
  Instance WEvaluable : Evaluable W := ·WordEvaluable bits.
  Instance REvaluable : Evaluable R := ·RWVEvaluable bits.
  Instance BEvaluable : Evaluable B := ·BoundedEvaluable bits.
  Existing Instances ZEvaluable WEvaluable REvaluable BEvaluable.

  Module HL.
    Definition progZ: NAry inputs Z (·HL.Expr Z ResultType) :=
        Input.prog.
    Definition progR: NAry inputs Z (·HL.Expr R ResultType) := liftN
        (fun x v ⇒ ·HLConversions.convertExpr Z R _ _ _ (x v))
        Input.prog.
    Definition progW: NAry inputs Z (·HL.Expr W ResultType) := liftN
        (fun x v ⇒ ·HLConversions.convertExpr Z W _ _ _ (x v))
        Input.prog.
```

```
  End HL.

  Module LL.
    Definition progZ: NAry inputs Z (·LL.expr Z Z ResultType) := liftN
        CompileHL.Compile HL.progZ.
    Definition progR: NAry inputs Z (·LL.expr R R ResultType) := liftN
        CompileHL.Compile HL.progR.
    Definition progW: NAry inputs Z (·LL.expr W W ResultType) := liftN
        CompileHL.Compile HL.progW.
  End LL.

  Module AST.
    Definition progZ: NAry inputs Z (·interp_type Z ResultType) := liftN
        LL.interp LL.progZ.
    Definition progR: NAry inputs Z (·interp_type R ResultType) := liftN
        LL.interp LL.progR.
    Definition progW: NAry inputs Z (·interp_type W ResultType) := liftN
        LL.interp LL.progW.
  End AST.

  Module Qhasm.
    Definition pair := ·CompileLL.compile bits width ResultType _
        LL.progW.
    Definition prog := option_map (·fst _ _) pair.
    Definition outputRegisters := option_map (·snd _ _) pair.
    Definition code := option_map StringConversion.convertProgram prog.
  End Qhasm.

  Module Bounds.
    Definition input := map (fun x ⇒ range N 0%N (Z.to_N x))
        Input.inputBounds.
    Definition upper := Z.of_N (wordToN (wones bits)).
    Definition prog :=
      Util.applyProgOn upper Input.inputBounds LL.progR.
    Definition valid := LLConversions.check (n := bits) (f := id) prog.
    Definition output :=
      typeMap (option_map (fun x ⇒ range N (rwv_low x) (rwv_high x)))
          (LL.interp prog).
  End Bounds.
End Pipeline.
```

Listing A.1: Full pipeline instantiated with GF25519 operations.

```
Module GF25519.
  Definition bits: nat := 64.
  Definition width: Width bits := W64.

  Existing Instance ZEvaluable.

  Fixpoint makeBoundList {n} k (b: ·BoundedWord n) :=
    match k with
    | O ⇒ nil
    | S k' ⇒ cons b (makeBoundList k' b)
    end.

  Section DefaultBounds.
    Import ListNotations.

    Local Notation rr exp := (2^exp + 2^(exp-3))%Z.

    Definition feBound: list Z :=
      [rr 26; rr 27; rr 26; rr 27; rr 26;
       rr 27; rr 26; rr 27; rr 26; rr 27].
  End DefaultBounds.

  Definition FE: type.
  Proof.
    let T := eval vm_compute in fe25519 in
    let t := HL.reify_type T in
    exact t.
  Defined.

  Section Expressions.
    Definition flatten {T}: (·interp_type Z FE → T) → NAry 10 Z T.
      intro F; refine (fun (a b c d e f g h i j: Z) ⇒
        F (a, b, c, d, e, f, g, h, i, j)).
    Defined.

    Definition unflatten {T}:
      (forall a b c d e f g h i j : Z, T (a, b, c, d, e, f, g, h, i, j))
    → (forall x: ·interp_type Z FE, T x).
    Proof.
      intro F; refine (fun (x: ·interp_type Z FE) ⇒
        let '(a, b, c, d, e, f, g, h, i, j) := x in
        F a b c d e f g h i j).
    Defined.

    Ltac intro_vars_for R := revert R;
      match goal with
      | [ |- forall x, ·?T x ] ⇒ apply (·unflatten T); intros
      end.

    Definition ge25519_add_expr :=
      Eval cbv beta delta [fe25519 carry_add mul carry_sub opp Let_In]
          in carry_add.
    Definition ge25519_sub_expr :=
```

```
  Eval cbv beta delta [fe25519 carry_add mul carry_sub opp Let_In]
      in carry_sub.
Definition ge25519_mul_expr :=
  Eval cbv beta delta [fe25519 carry_add mul carry_sub opp Let_In]
      in mul.
Definition ge25519_opp_expr :=
  Eval cbv beta delta [fe25519 carry_add mul carry_sub opp Let_In]
      in opp.

Definition ge25519_add' (P Q: ·interp_type Z FE):
  { r: ·HL.Expr Z FE | HL.Interp r = ge25519_add_expr P Q }.
Proof.
  intro_vars_for P.
  intro_vars_for Q.
  eexists.

  cbv beta delta [ge25519_add_expr].
  etransitivity; [reflexivity|].

  let R := HL.rhs_of_goal in
  let X := HL.Reify R in
  transitivity (HL.Interp (X bits)); [reflexivity|].

  cbv iota beta delta [ HL.Interp
      interp_type interp_binop HL.interp
      Z.land ZEvaluable eadd esub emul eshiftr eand].

  reflexivity.
Defined.

Definition ge25519_sub' (P Q: ·interp_type Z FE):
  { r: ·HL.Expr Z FE | HL.Interp r = ge25519_sub_expr P Q }.
Proof.
  intro_vars_for P.
  intro_vars_for Q.
  eexists.

  cbv beta delta [ge25519_sub_expr].
  etransitivity; [reflexivity|].

  let R := HL.rhs_of_goal in
  let X := HL.Reify R in
  transitivity (HL.Interp (X bits)); [reflexivity|].

  cbv iota beta delta [ HL.Interp
      interp_type interp_binop HL.interp
      Z.land ZEvaluable eadd esub emul eshiftr eand].

  reflexivity.
Defined.

Definition ge25519_mul' (P Q: ·interp_type Z FE):
  { r: ·HL.Expr Z FE | HL.Interp r = ge25519_mul_expr P Q }.
```

```
  Proof.
    intro_vars_for P.
    intro_vars_for Q.
    eexists.

    cbv beta delta [ge25519_mul_expr].
    etransitivity; [reflexivity|].

    let R := HL.rhs_of_goal in
    let X := HL.Reify R in
    transitivity (HL.Interp (X bits)); [reflexivity|].

    cbv iota beta delta [ HL.Interp
        interp_type interp_binop HL.interp
        Z.land ZEvaluable eadd esub emul eshiftr eand].

    reflexivity.
  Defined.

  Definition ge25519_opp' (P: ·interp_type Z FE):
    { r: ·HL.Expr Z FE | HL.Interp r = ge25519_opp_expr P }.
  Proof.
    intro_vars_for P.
    eexists.

    cbv beta delta [ge25519_opp_expr zero_].
    etransitivity; [reflexivity|].

    let R := HL.rhs_of_goal in
    let X := HL.Reify R in
    transitivity (HL.Interp (X bits)); [reflexivity|].

    cbv iota beta delta [ HL.Interp
        interp_type interp_binop HL.interp
        Z.land ZEvaluable eadd esub emul eshiftr eand].

    reflexivity.
  Defined.

  Definition ge25519_add (P Q: ·interp_type Z FE) := proj1_sig
      (ge25519_add' P Q).
  Definition ge25519_sub (P Q: ·interp_type Z FE) := proj1_sig
      (ge25519_sub' P Q).
  Definition ge25519_mul (P Q: ·interp_type Z FE) := proj1_sig
      (ge25519_mul' P Q).
  Definition ge25519_opp (P: ·interp_type Z FE) := proj1_sig
      (ge25519_opp' P).
End Expressions.

Module AddExpr <: Expression.
  Definition bits: nat := bits.
  Definition inputs: nat := 20.
  Definition width: Width bits := width.
```

```
  Definition ResultType := FE.
  Definition inputBounds := feBound ++ feBound.
  Definition prog: NAry 20 Z (·HL.Expr Z ResultType) :=
     Eval cbv in (flatten (fun p ⇒ (flatten (fun q ⇒ ge25519_add p
        q)))).
End AddExpr.

Module SubExpr <: Expression.
  Definition bits: nat := bits.
  Definition inputs: nat := 20.
  Definition width: Width bits := width.
  Definition ResultType := FE.
  Definition inputBounds := feBound ++ feBound.
  Definition prog: NAry 20 Z (·HL.Expr Z ResultType) :=
     Eval cbv in (flatten (fun p ⇒ (flatten (fun q ⇒ ge25519_sub p
        q)))).
End SubExpr.

Module MulExpr <: Expression.
  Definition bits: nat := bits.
  Definition inputs: nat := 20.
  Definition width: Width bits := width.
  Definition ResultType := FE.
  Definition inputBounds := feBound ++ feBound.
  Definition prog: NAry 20 Z (·HL.Expr Z ResultType) :=
     Eval cbv in (flatten (fun p ⇒ (flatten (fun q ⇒ ge25519_mul p
        q)))).
End MulExpr.

Module OppExpr <: Expression.
  Definition bits: nat := bits.
  Definition inputs: nat := 10.
  Definition width: Width bits := width.
  Definition ResultType := FE.
  Definition inputBounds := feBound.

  Definition prog: NAry 10 Z (·HL.Expr Z ResultType) :=
     Eval cbv in (flatten ge25519_opp).
End OppExpr.

Module Add := Pipeline AddExpr.
Module Sub := Pipeline SubExpr.
Module Mul := Pipeline MulExpr.
Module Opp := Pipeline OppExpr.

Section Instantiation.
  Import InitialRing.

  Definition Binary : Type := NAry 20 (word bits) (·interp_type (word
     bits) FE).
  Definition Unary : Type := NAry 10 (word bits) (·interp_type (word
     bits) FE).
```

```
Ltac ast_simpl := cbv [
  Add.bits Add.inputs AddExpr.inputs Add.ResultType
      AddExpr.ResultType
  Add.W Add.R Add.ZEvaluable Add.WEvaluable Add.REvaluable
  Add.AST.progW Add.LL.progW Add.HL.progW AddExpr.prog

  Sub.bits Sub.inputs SubExpr.inputs Sub.ResultType
      SubExpr.ResultType
  Sub.W Sub.R Sub.ZEvaluable Sub.WEvaluable Sub.REvaluable
  Sub.AST.progW Sub.LL.progW Sub.HL.progW SubExpr.prog

  Mul.bits Mul.inputs MulExpr.inputs Mul.ResultType
      MulExpr.ResultType
  Mul.W Mul.R Mul.ZEvaluable Mul.WEvaluable Mul.REvaluable
  Mul.AST.progW Mul.LL.progW Mul.HL.progW MulExpr.prog

  Opp.bits Opp.inputs OppExpr.inputs Opp.ResultType
      OppExpr.ResultType
  Opp.W Opp.R Opp.ZEvaluable Opp.WEvaluable Opp.REvaluable
  Opp.AST.progW Opp.LL.progW Opp.HL.progW OppExpr.prog

  HLConversions.convertExpr CompileHL.Compile CompileHL.compile

  LL.interp LL.uninterp_arg LL.under_lets LL.interp_arg
  ZEvaluable WordEvaluable RWVEvaluable rwv_value

  eadd esub emul eand eshiftr toT fromT
  interp_binop interp_type FE liftN NArgMap id
  omap option_map orElse].

(* Tack this on to make a simpler AST, but it really slows us down
    *)
Ltac word_simpl := cbv [
  AddExpr.bits SubExpr.bits MulExpr.bits OppExpr.bits bits
  NToWord posToWord natToWord wordToNat wordToN wzero'
  Nat.mul Nat.add].

Ltac kill_conv := let p := fresh in
  pose proof N2Z.id as p; unfold Z.to_N in p;
  repeat rewrite p; clear p;
  repeat rewrite NToWord_wordToN.

Local Notation unary_eq f g
  := (forall x0 x1 x2 x3 x4 x5 x6 x7 x8 x9,
        f x0 x1 x2 x3 x4 x5 x6 x7 x8 x9
        = g x0 x1 x2 x3 x4 x5 x6 x7 x8 x9).
Local Notation binary_eq f g
  := (forall x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 y0 y1 y2 y3 y4 y5 y6 y7
      y8 y9,
        f x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 y0 y1 y2 y3 y4 y5 y6 y7 y8 y9
        = g x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 y0 y1 y2 y3 y4 y5 y6 y7 y8
            y9).
```

```
    Definition add'
      : {f: Binary |
         binary_eq f (NArgMap (fun x ⇒ Z.of_N (wordToN x))
             Add.AST.progW) }.
    Proof. eexists; intros; ast_simpl; kill_conv; reflexivity. Defined.

    Definition sub'
      : {f: Binary |
         binary_eq f (NArgMap (fun x ⇒ Z.of_N (wordToN x))
             Sub.AST.progW) }.
    Proof. eexists; ast_simpl; kill_conv; reflexivity. Defined.

    Definition mul'
      : {f: Binary |
         binary_eq f (NArgMap (fun x ⇒ Z.of_N (wordToN x))
             Mul.AST.progW) }.
    Proof. eexists; ast_simpl; kill_conv; reflexivity. Defined.

    Definition opp' : {f: Unary |
      unary_eq f (NArgMap (fun x ⇒ Z.of_N (wordToN x)) Opp.AST.progW) }.
    Proof. eexists; ast_simpl; kill_conv; reflexivity. Defined.

    Definition add := Eval simpl in proj1_sig add'.
    Definition sub := Eval simpl in proj1_sig sub'.
    Definition mul := Eval simpl in proj1_sig mul'.
    Definition opp := Eval simpl in proj1_sig opp'.
  End Instantiation.
End GF25519.

Extraction "GF25519Add" GF25519.Add.
Extraction "GF25519Sub" GF25519.Sub.
Extraction "GF25519Mul" GF25519.Mul.
Extraction "GF25519Opp" GF25519.Opp.
```

Listing A.2: Full pipeline instantiated with GF25519 operations.

Bibliography

[1] Practical realisation and elimination of an ECC-related software bug attack. 2011.

[2] BoringSSL. `https://www.imperialviolet.org/2015/10/17/boringssl.html`, 2017.

[3] LibreSSL. `http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libssl/src/ssl/`, 2017.

[4] OpenSSL vulnerabilities. `https://www.openssl.org/news/vulnerabilities.html`, 2017.

[5] Jason Gross Robert Sloan Andres Erbsen, Jade Philipoom and Adam Chlipala. Systematic synthesis of elliptic curve cryptography implementations. Working Draft, 2017. `https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf`.

[6] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., 2015. USENIX Association.

[7] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography - PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26*. Springer-Verlag.

[8] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. Document-ID: a1a62a2f76d23f65d622484ddd09caf8.

[9] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'07, pages 29–50, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Daniel J. Bernstein and Peter Schwabe.

[11] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 299–309. ACM, 2014. Document ID: 55ab8668ce87d857c02a5b2d56d7da38.

[12] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

[13] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011.

[14] Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM.

[15] Russ Cox. Lessons from the debian/openssl fiasco. `https://research.swtch.com/openssl`, 2008.

[16] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, 2015.

[17] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[18] Thai Duong et al. Project wycheproof. `https://github.com/google/wycheproof`, 2017.

[19] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[21] M. Crenshaw M. Afidler, J. Granick. Anarchy or regulation: Controlling the global trade in zero-day vulnerabilities. Doctoral dissertation, Master Thesis. Stanford University, 2014. `https://d1x4j6omi7lpzs.cloudfront.net/live/wp-content/uploads/2014/06/Fidler-Zero-Day-Vulnerability-Thesis.pdf`.

[22] Magnus O. Myreen and Gregorio Curello. A verified bignum implementation in x86-64 machine code.

[23] Loïc Pottier. Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. *CoRR*, abs/1007.3615, 2010.

[24] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

[25] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. 2 2016. orig. 2002.

[26] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. April 2003.

[27] W3Schools. Browser statistics. `https://www.w3schools.com/Browsers/default.asp`, 2017.

[28] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.