Correct-by-Construction Finite Field Arithmetic in Coq

by

Jade Philipoom

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author .....
Department of Electrical Engineering and Computer Science
February 2, 2018

Certified by .....
Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Chris Terman
Chairman, Master of Engineering Thesis Committee

Correct-by-Construction Finite Field Arithmetic in Coq

by

Jade Philipoom

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

Elliptic-curve cryptography code, although based on elegant and concise mathematical procedures, often becomes long and complex due to speed optimizations. This statement is especially true for the specialized finite-field libraries used for ECC code, resulting in frequent implementation bugs. I describe the methodologies used to create a Coq framework that generates implementations of finite-field arithmetic routines along with proofs of their correctness, given nothing but the modulus.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Electrical Engineering and Computer Science

Contents

1 Related Work **13**

1.1 ECC Code Verification 14

1.1.1 Cryptol and SAW 14

1.1.2 HACL* 15

1.1.3 verify25519 15

1.2 Bignum Arithmetic Verification 16

1.2.1 gfverif 16

1.3 Verified Bignums 16

1.4 Adjacent Work 17

1.4.1 Protocol Verification 17

1.4.2 Compilers and Translation Validators 17

2 Mathematical Background **19**

2.1 Modular Reduction 19

2.1.1 Generalized Mersenne Reduction 20

2.1.2 Montgomery Reduction 22

2.2 Integer Representation 23

2.2.1 Unsaturated Digits 23

2.2.2 Mixed-Radix Bases 25

3 Representing Multidigit Integers in Coq **27**

3.1 Unsaturated 27

3.1.1 Two-lists Representation 27

3.1.2 Associational and Positional Representations 32
3.2 Saturated 37

4 Summary of Operations **41**
4.1 Unsaturated 41
4.2 Saturated 44
4.3 Other 46

5 Producing Parameter-Specific Code **49**
5.1 Selecting Parameters 50
5.2 Partial Evaluation 52
5.2.1 Synthesis Mode 52
5.2.2 Marking Runtime Operations 53
5.2.3 Continuation-Passing Style 54
5.3 Linearization and Constant-Folding 57
5.4 Bounds Inference 58

6 Extending with New Arithmetic Techniques **59**
6.1 Algorithm 59
6.2 Implementation 60
6.3 Overall Effort of Extension 62

A Performance Tables **63**

Introduction

Cryptography is only becoming more necessary in the modern world, as it becomes clear that communicating with other individuals over the internet exposes users to governments, companies, and other malicious actors who are eager to make use of such broadcasts. Most internet traffic is now encrypted, and the world would be a better place if that cryptography were *stronger* and *more ubiquitous*. However, these goals can be at odds. Ubiquitous encryption means that cryptographic routines are run very often, for instance on every connection with a website. This means that whoever is running the web server has a strong incentive to push for faster code. Faster code, though, often includes very nontrivial optimizations that make the code less easy to audit, and prone to tricky implementation bugs. This work is part of a project that uses a mathematical proof assistant, Coq, to produce elliptic-curve cryptography (ECC) code that is both fast and certifiably free of implementation error. Although the larger project has a scope ranging from elliptic-curve mathematical properties all the way to assembly-like C code, the work described in this thesis focuses specifically on crafting simplified functional versions of performance-sensitive finite-field arithmetic routines.

To motivate design decisions on this project, we (primarily my colleague Andres Erbsen) conducted an informal survey of project bug trackers and other internet sources, which is summarized in Figure 0-1. Interesting to note here is that, of the 28 bugs we looked at, 20 are mistakes in correctly handling the complicated finite-field arithmetic routines (encompassing three categories in the figure: *Carrying*, *Canonicalization*, and *Misc. Number System*). The distribution reflects the fact that finite-field routines, as the largest component of runtime in most cases, have

Reference	Specification	Implementation	Defect
Carrying			
go#13515	Modular exponentiation	uintptr-sized Montgomery form, Go	carry handling
NaCl ed25519 (p.2)	F25519 mul, square	64-bit pseudo-Mersenne, AMD64	carry handling
openssl#ef5c9b11	Modular exponentiation	64-bit Montgomery form, AMD64	carry handling
openssl#74acf42c	Poly1305	multiple implementations	carry handling
nettle#09e3ce4d	secp-256r1 modular reduction		carry handling
CVE-2017-3732	$x^2 \bmod m$	Montgomery form, AMD64 assembly	carry, exploitable
openssl#1593	P384 modular reduction	carry handling	carry, exploitable
tweetnacl-U32	irrelevant	bit-twiddly C	'sizeof(long)!=32'
Canonicalization			
donna#8edc799f	GF($2^{255}-19$) internal to wire	32-bit pseudo-Mersenne, C	non-canonical
openssl#c2633b8f	a + b mod p256	Montgomery form, AMD64 assembly	non-canonical
tweetnacl-m15	GF($2^{255}-19$) freeze	bit-twiddly C	bounds? typo?
Misc. number system			
openssl#3607	P256 field element squaring	64-bit Montgomery form, AMD64	limb overflow
openssl#0c687d7e	Poly1305	32-bit pseudo-Mersenne, x86 and ARM	bad truncation
CVE-2014-3570	Bignum squaring	asm	limb overflow
ic#237002094	Barrett reduction for p256	1 conditional subtraction instead of 2	no counterexample
go#fa09811d	poly1305 reduction	AMD64 asm, missing subtraction of 3	found quickly
openssl#a970db05	Poly1305	Lazy reduction in x86 asm	lost bit 59
openssl#6825d74b	Poly1305	AVX2 addition and reduction	bounds?
ed25519.py	Ed25519	accepts signatures other impls reject	missing $h \bmod l$
bitcoin#eed71d85	ECDSA-secp256k1 x*B	mixed addition Jacobian+Affine	missing case
Elliptic Curves			
openjdk#01781d7e	EC scalarmult	mixed addition Jacobian+Affine	missing case
jose-adobe	ECDH-ES	5 libraries	not on curve
invalid-curve	NIST ECDH	Irrelevant	not on curve
end-to-end#340	Curve25519 library	twisted Edwards coordinates	$(0,1)=\infty$
openssl#59dfcabf	Weier. affine <-> Jacobian	Montgomery form, AMD64 and C	∞ confusion
Crypto Primitives			
socat#7	DH in Z*p	irrelevant	non-prime p
CVE-2006-4339	RSA-PKCS-1 sig. verification	irrelevant	padding check
CryptoNote	Anti-double-spending tag	additive curve25519 curve point	missed order$(P) \neq l$

Figure 0-1: Survey of bugs in algebra-based cryptography implementations

been subject to extremely tricky optimizations. Elliptic-curve cryptography code often uses complex representations for finite-field elements, and avoiding over- or underflow requires tracking very precise upper and lower bounds in one's head. It is unsurprising, therefore, that humans make arithmetic errors and fail to account for some of the many edge cases. Historically, computers have proven much more reliable than humans at those tasks. Therefore, having a computer produce and check the code has the potential to drastically reduce error rates.

To understand various design decisions described in this document, it is helpful to note the three main goals that have guided the project since its inception:

- Scope: our proofs should be as deep as possible. Currently, we can verify from high-level functional specifications in terms of elliptic curves down to straight-line, assembly-like C.

- Speed: the code we produce must be fast enough to be competitive with real-world C implementations, which means making use of a variety of nontrivial optimizations. Comparing to specialized C implementations, we come in around 5% slower than donna-c64 and significantly faster than OpenSSL C. We are still roughly 20% slower than hand-optimized assembly, but we achieve between a 1.2 and 10× speedup (depending on the prime) compared to reasonable implementations using libgmp, a fast arbitrary-precision integer library. Figure 0-2 shows our performance compared to the libgmp implementations for various primes. For the full benchmark data, see Appendix A.

- Effort: we want to prove as many things as we can once-and-for-all, so that changing parameters (for instance, changing curves) will not require any manual work. Currently, it is possible to generate new finite-field operations by inputting only a prime; the rest is automatic.

This document is organized as follows. Chapter 1 details various pieces of related work to help set an academic context and provide insight into the design space. Chapter 2 describes the mathematical techniques underpinning our development, and

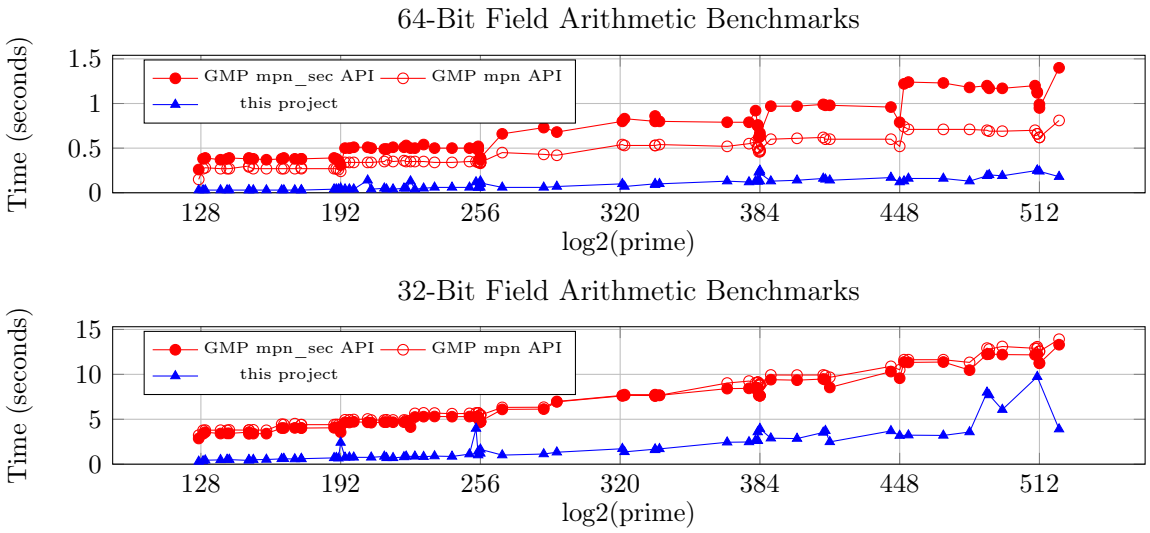

Figure 0-2: Performance comparison of our generated C code vs. handwritten using `libgmp`

Chapter 3 describes our overall strategies for how to encode those ideas in Coq. Chapter 4 provides a list of all the operations defined by the finite-field arithmetic library and gives some detail on the implementation of each one. Chapter 5 describes how we transform proven-correct general code into fast low-level implementations. Finally, Chapter 6 remarks on how the library can be extended with new techniques, using Karatsuba multiplication as an example.

Individual Contributions

The project described in this thesis is a collaboration between myself, Andres Erbsen, and Jason Gross, all advised by Professor Adam Chlipala. For an early version of the library, Robert Sloan built a low-level compilation pipeline which ended up being replaced as the requirements matured.

This document will describe the finite-field arithmetic section of the library only, which was designed collaboratively by myself and Andres but subsequently developed primarily by me. The low-level compilation work described in Chapter 5 was primarily done by Jason, also with design input from Andres. Some text about related work in Chapter 1 and the benchmark description in Appendix A is drawn from a single

jointly authored paper. Andres's thesis also influenced my writing about related work in Chapter 1.

Chapter 1

Related Work

Numerous other projects in this general domain are relevant to the academic context of this work and serve as a basis of comparison for our strategies, goals, and results. These can be grouped into two broad categories:

- ECC code verification efforts (Section 1.1), which are domain-specific projects that aim for guarantees of functional correctness.

- Bignum arithmetic verification efforts (Section 1.2), which are not specific to ECC but are highly useful for checking its most important properties. This is particularly relevant to verified finite-field arithmetic, the section of our project that this document describes.

In Section 1.4, we will also discuss some adjacent domains: protocol verification and verified compilers/translation validators. Such projects do not have much overlap with our work but are plausible connecting points for a truly end-to-end proof, which could provide guarantees all the way from security properties formalized in the style of academic cryptography down to assembly. This bottom level could potentially be connected to a formal model of a hardware processor.

```
p384_field_mod : [768] -> [384]
p384_field_mod(a)
    = drop(if b1 then r0
           else if b2 then r1
           else if b3 then r2
           else if b4 then r3
           else r4)
  where
    [  a23, a22, a21, a20, a19, a18, a17, a16, a15, a14, a13, a12,
       a11, a10,  a9,  a8,  a7,  a6,  a5,  a4,  a3,  a2,  a1,  a0]
            = [ uext x | (x : [32]) <- split a ] : [24][64]

    chop : [64] -> ([64],[32])
    chop x = (iext(take(x):[32]), drop(x))

    (d0, z0)  = chop(    a0          +a12+a21        +a20-a23)
    (d1, z1)  = chop(d0 +a1          +a13+a22+a23        -a12-a20)
    (d2, z2)  = chop(d1 +a2          +a14+a23            -a13-a21)
    (d3, z3)  = chop(d2 +a3          +a15+a12+a20    +a21-a14-a22-a23)
    (d4, z4)  = chop(d3 +a4 +(a21<<1)+a16+a13+a12+a20+a22-a15-(a23<<1))
    (d5, z5)  = chop(d4 +a5 +(a22<<1)+a17+a14+a13+a21+a23-a16)
    (d6, z6)  = chop(d5 +a6 +(a23<<1)+a18+a15+a14+a22    -a17)
    (d7, z7)  = chop(d6 +a7          +a19+a16+a15+a23    -a18)
    (d8, z8)  = chop(d7 +a8          +a20+a17+a16        -a19)
    (d9, z9)  = chop(d8 +a9          +a21+a18+a17        -a20)
    (d10,z10) = chop(d9 +a10         +a22+a19+a18        -a21)
    (d11,z11) = chop(d10+a11         +a23+a20+a19        -a22)

    r : [13*32]
    r = (drop(d11):[32])
      # z11 # z10 #  z9 #  z8 #  z7 #  z6
      #  z5 #  z4 #  z3 #  z2 #  z1 #  z0

    p  = uext(p384_prime) : [13*32]
    // Fix potential underflow
    r0 = if (d11@0) then r + p else r
    // Fix potential overflow
    (r1,b1)   = sbb(r0, p)
    (r2,b2)   = sbb(r1, p)
    (r3,b3)   = sbb(r2, p)
    (r4,b4)   = sbb(r3, p)
```

Figure 1-1: Cryptol specification of reduction modulo p384

1.1 ECC Code Verification

1.1.1 Cryptol and SAW

Galois, Inc. has created Cryptol [14], a language for specifying cryptographic algorithms. This can be combined with the Software Analysis Workbench [9], which can compare Cryptol specifications with C or Java code using SAT and SMT solvers. This technique has been used, according to their GitHub page, on a variety of examples, including salsa20, ECDSA, and AES, illustrating greater breadth than our work as of yet.

The main difference between the SAW strategy and ours is the use of SAT and SMT solvers rather than interactive proof assistants like Coq. So-called "push-button" techniques are much less transparent, and the extent to which they are or are not generalizable is a hotly debated topic. However, in this case, the Cryptol specifications used by SAW must be much closer to the code itself than our own. Figure 1-1, for instance, shows the Cryptol specification of reduction modulo $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ from the SAW ECDSA example. This specification already encodes very nontrivial transformations, including distributing the integer among several different machine words. Our specification for the same `p384_field_mod` operation would be `forall x, eval (p384_field_mod x) = Z.mod (eval x) p`, where `Z.mod` is the standard library modular reduction for Coq's infinite-precision integers.

1.1.2 HACL*

HACL* [23] is a cryptographic library implemented and verified in the F* programming language. Using the Low* imperative subset of F* [18], which supports semantics-preserving translation to C, HACL* is able to connect high-level functional correctness specifications to code that beats or comes close to performance of leading C libraries. Additionally, abstract types for secret data rule out side-channel leaks.

The main difference between the current version of HACL* and our strategy is that we aim to verify techniques once and for all, so that there is no manual proof effort to generate finite-field arithmetic for a new curve. HACL* requires significant per-curve code and solver hints. A predecessor system [22] with overlapping authors performed a more high-level verification that was closer to our development. However, the code incurred a performance overhead above 100×.

1.1.3 verify25519

The verify25519 project [7] verified an assembly implementation of the Montgomery ladder step for curve25519 using a combination of the Boolector SMT solver and Coq. The assembly was translated to Boolector formulas (with the help of man-

ual "midconditions": assertions that act as hints to the translator) and related to a very minimal implementation, which was then transcribed to Coq and proven correct. This project verifies faster code than any of the others, which work with C rather than hand-optimized assembly, and is notable for requiring no changes to the existing implementation. However, this strategy also requires per-curve work in the midconditions, which can be quite nontrivial. Also, the trusted code base includes a 2000-line program to translate assembly to Boolector, which is not ideal, although the code is fairly straightforward.

1.2 Bignum Arithmetic Verification

1.2.1 gfverif

A tool called gfverif [6], still in experimental stages, uses the Sage computer-algebra system to check implementations of finite-field operations. The implementations they check use similar techniques to the code that we generate. The authors, who overlap with the authors of verify25519, say that compared to verify25519 gfverif requires significantly less annotation effort and has much faster checking time. However, it also works on C rather than assembly, meaning the verified code is significantly slower, and it did require some small changes to the source code. It is also unclear how widely applicable the strategy is, given that only a couple of different operations have been tried. For instance, it is unclear if gfverif would work on saturated arithmetic.

1.3 Verified Bignums

Myreen and Curello verified a general-purpose big-integer library [15] using HOL. The code does not allow for any bignum representation other than saturated 64-bit words, which would be a serious limitation for its use in the ECC domain. However, it is impressive in that the verification uses verified decompilation to extend to assembly.

1.4 Adjacent Work

Additionally, some projects exist which are on either end of the scope covered by this project. While we take code from the abstraction level of elliptic-curve operations down to low-level C, ideally we could connect this work to projects that go from security properties to elliptic-curve operations (protocol verification) and from low-level C to assembly (verified compilers, translation validators).

1.4.1 Protocol Verification

Efforts such as CertiCrypt [4], EasyCrypt [3] and FCF [17] attempt to connect functional specifications to high-level security properties like indistinguishability. CertiCrypt is built on top of Coq, but laborious proof development led the authors to create EasyCrypt, an SMT-based successor. FCF is a Coq system inspired by CertiCrypt that attempts to allow better use of Coq's tactic machinery to reduce proof effort.

1.4.2 Compilers and Translation Validators

CompCert [13] is a verified compiler from (only slightly constrained) C to the PowerPC, ARM, RISC-V and x86 (32- and 64- bit) architectures, implemented in Coq. Translation validators like Necula's [16] can sometimes check the work of unverified compilers.

Also worth mentioning is Jasmin [2], which is a low-level language with simple control flow reminiscent of C. It contains a compiler, verified in Coq, from this straightline code to 64-bit x86 assembly. Reductions to Dafny check memory safety and guarantee absence of information leaks.

Chapter 2

Mathematical Background

This chapter will describe a selection of applied-cryptography techniques in mathematical terms. These are the techniques we designed our library to accommodate. They fall into two major categories: modular reduction strategies (generalized Mersenne, Barrett, and Montgomery) and integer representations (unsaturated digits, mixed-radix bases).

It should be noted that the library ultimately contains two levels of representation: first, we represent very large integers and their operations in a computer with limited word sizes and performance constraints, and second, we represent *those* representations in Coq. This chapter will be devoted entirely to the former, while chapter 3 will address the latter.

2.1 Modular Reduction

Modular reduction, in its most familiar form, is the operation that, given a number x and a modulus p, returns a number y such that $x \equiv y \pmod{p}$ and $0 \leq y < p$. We will be using the term in a slightly more general sense: the upper bound on y is not necessarily exactly p. For instance, we might only care that y has approximately as many bits as p.

Naïvely, modular reduction can be implemented with a division:

$$y = x - p \cdot \lfloor x/p \rfloor$$

However, since the operation happens extremely frequently in cryptographic arithmetic and division is slow, it is highly advantageous to use more complicated strategies that replace the division with multiplications and additions. A number of algorithms exist for this purpose.

2.1.1 Generalized Mersenne Reduction

One of the most important procedures in our library is a specialized reduction defined for generalized Mersenne numbers. Also called Solinas numbers, these are numbers of the form $p = f(2^k)$, where $f(t) = t^d - c_1 t^{d-1} - ... - c_d$ is a degree-d integral polynomial and has a low "modular reduction weight" [19]. This weight measures the number of operations in the reduction algorithm; Solinas has a matrix-based formula for calculating it precisely, but the weight is typically low when $f(t)$ has few terms and those terms have small coefficients. Table 2.1 has some examples of Solinas primes that are used in modern cryptography.

The modular reduction routine for these primes is best illustrated with examples. First, let's try reducing modulo $p = 2^{255} - 19$. Let's say we have an input x, such that x has 512 bits, and we want to obtain a number that is equivalent to x modulo p but has only 256 bits.

Name	p	d	k	$f(t)$
NIST p-192 [1]	$2^{192} - 2^{64} - 1$	3	64	$t^3 - t - 1$
NIST p-224 [1]	$2^{224} - 2^{96} + 1$	7	32	$t^7 - t^3 + 1$
curve25519 [5]	$2^{255} - 19$	1	255	$t - 19$
NIST p-256 [1]	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	8	32	$t^8 - t^7 + t^6 + t^3 - 1$
NIST p-384 [1]	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	12	32	$t^{12} - t^4 - t^3 + t - 1$
Goldilocks p-448 [12]	$2^{448} - 2^{224} - 1$	2	224	$t^2 - t - 1$

Table 2.1: Commonly used Solinas primes and their parameters. For some of the primes on this list, Montgomery reduction is faster, but generalized Mersenne reduction works for all of them.

We can split the high and low bits of x to express it as $l + 2^{255}h$, where l has 255 bits and h has 257 bits. Then:

$$x = l + 2^{255}h = l + (2^{255} - 19)h + 19h \equiv l + 19h \pmod{2^{255} - 19}$$

So, if we multiply the high part of x by 19 and add it to the low part, we get a number congruent to x modulo p. $19h$ has $257 + 5 = 262$ bits, so $l + 19h$ has 263. If we do the reduction procedure again and split $l + 19h$ into $l' + 2^{255}h'$, there will only be 8 high bits, so $19h'$ will have 13 bits and $l' + 19h'$ will have 256.

As another example, let's reduce a number x with $448 \times 2 = 896$ bits modulo $p = 2^{448} - 2^{224} - 1$, to a number with around 448 bits.

$$\begin{aligned} x = l + 2^{448}h &= l + (2^{448} - 2^{224} - 1)h + 2^{224}h + h \\ &\equiv l + 2^{224}h + h \pmod{2^{448} - 2^{224} - 1} \end{aligned}$$

Since h has 448 bits, $2^{224}h$ has 672 bits, and $l + 2^{224}h + h$ has 674. Repeating the reduction produces an h' with $674 - 448 = 226$ bits, so $2^{224}h'$ has 450 bits and $l' + 2^{224}h' + h'$ has 452. If we do the reduction again, we get only 4 bits in h'', so $l'' + 2^{224}h'' + h''$ is dominated by the 448-bit l'' and results in 450 bits.

Because this method makes x smaller every step as long as $2^{448} \leq x$, we could use this method to make x have no more than 448 bits. But simple bit-length analysis will not get us there; there will always be 448 lower bits, and we will always add two numbers to them, so we will continue to have 450 bits. We could compare x to 2^{448} at every step, but that would open a timing side channel, since our algorithm would take different branches on different inputs.

However, a 450-bit number is necessarily less than $4p$. So, using a constant-time procedure that subtracts p from its input if the input is greater than p and subtracts zero otherwise, we can get to 448 bits. In some cases, we may care to do that, but in others 450 bits might be close enough.

In general, we can express this reduction algorithm in the following way. Assuming

a Solinas prime p such that $p = f(2^k)$ and $f(t) = t^d - c_1 t^{d-1} - ... - c_d$, we can express p as $2^{dk} - c$, where $c = c_1 2^{(d-1)k} + ... + c_d$. Then, we can reduce x modulo p by splitting the bits of x into the lowest dk bits and the higher bits, multiplying the higher bits by c and adding the product to the lower bits. If we have a number with about twice as many bits as p (such as the output of a multiplication), then doing this procedure twice will get us to a congruent number about the same size as p.

This reduction procedure is very similar to the one originally described by Solinas, although the explanation above looks very different from his; Solinas has a non-iterative description of the procedure that uses linear-feedback shift registers and matrices. He also assumes that the additions of low to high bits will be modular additions.

2.1.2 Montgomery Reduction

Montgomery reduction is a completely different strategy for fast modular reduction. Using this strategy, one avoids the division from the naïve algorithm by forcing the division to be by a number other than the prime–some number for which division is fast (for instance, a multiple of machine-word sizes). We operate in the "Montgomery domain": in this case, a is represented by $(a \cdot R) \bmod p$, where R is our chosen fast divisor. To compute a product $a \cdot b \cdot R \bmod p$ given the operands in Montgomery form and a number R' such that $(R \cdot R') \bmod p = 1$, we compute:

$$((aR \bmod p) \cdot (bR \bmod p) \cdot R') \bmod p$$

Correctness follows from some simple algebra:

$$((aR \bmod p) \cdot (bR \bmod p) \cdot R') \bmod p = (aR \cdot bR \cdot R') \bmod p = ((a \cdot b)R) \bmod p$$

If the multiplication by R' and modular reduction by p are done sequentially, this does not yield any performance gain. However, it is possible to combine the multiplication and reduction in a faster algorithm known as "redc". To be more

specific, because cryptography deals with large, multi-word numbers, we use the *word-by-word* variant of redc, as described by Gueron et al. [11].

This algorithm is harder to digest than the generalized Mersenne reduction described earlier, and it is not particularly instructive to reproduce it here. The most relevant feature of the algorithm is the requirements it imposes on our multidigit integer library, which includes the following operations:

- `divmod`: splits off lowest-weight digit from the rest of the digits
- `scmul`: multiplies all digits by a scalar
- `add`: adds two multidigit integers using an add-with-carry loop that appends the carry as an additional digit
- `drop_high`: removes the highest-weight digit (for instance, if the carry is 0, the extra digit added by *add* can be removed safely)

2.2 Integer Representation

Due to the size of the numbers used in cryptography, a single finite-field element will not fit in a single 32- or 64-bit register. It is necessary to divide the number somehow into several smaller chunks with different weighting. Naïvely, one could use 32- or 64-bit chunks, such that the ith chunk has weight 2^{32i} or 2^{64i}. Indeed, this strategy produces the most efficient representation in many cases, including Montgomery reduction. However, when using the generalized Mersenne reduction strategy, it can be highly advantageous to represent the numbers using a more complicated scheme.

2.2.1 Unsaturated Digits

In generalized Mersenne reduction, a crucial step is to split the input into high and low bits–in particular, we need to obtain two numbers l and h such that our input $x = l + 2^{kd}h$. Consider $p = 2^{255} - 19$, where $kd = 255$, and a platform with 64-bit integers. If we do a multiplication, our 510-bit product will be split across 8 64-bit

registers. The 255th bit occurs as the second-to-last in the fourth register. So to separate out the lowest 255 bits, we'd have to remove the last bit from the fourth register and essentially completely recalculate the remaining four registers to insert the 256th bit into the beginning, which wastes a great deal of time in an extremely performance-critical subroutine.

Instead, imagine if that 255th bit happened to be the last one in a register. Then, splitting the number would be completely free; it would have already been done by our representation, automatically. It turns out that it is possible to force the registers to align this way, by using base 2^{51} instead of base 2^{64}. Since $51 \times 5 = 255$, we can simply separate the first 5 registers in order to get l and h such that our input $x = l + 2^{255}h$. Because we do not use all the bits in the register, this representation is called an "unsaturated" one, and a more typical representation using all of the bits is "saturated".

This optimization saves so much time that it is well worth the time spent on maintaining extra registers. It also has another side benefit: since the base does not match the register width, there is no reason to make sure that we only have 51 bits in each digit. So, in addition for instance, rather than doing a sequence of add-with-carries, we could simply add each digit of one operand to the corresponding digit of the other and not carry at all. Eventually, of course, we would have to run a loop that would carry some of these accumulated bits, but we can do several additions before then.

Example: Delayed Carrying

To elaborate on the carrying trick a little, consider a normal schoolbook addition of 288 and 142 in base 10. First we add the 8 and the 2, getting a 0 with a carry of 1, then we add 1 and 8 and 4 and get 3 with a carry of 1, and then we add 1 and 2 and 1 to get 4, leaving us with a final result of 430. However, if we do not bother keeping digits small, we can avoid the dependencies. Using 288 and 142 again, we get the result $3, 12, 10$ (separating digits with commas for clarity), which represents the same answer; $10 + 10 * 12 + 100 * 3 = 10 + 120 + 300 = 430$. We can even convert it to the

usual base-10 format by setting the least significant digit, 10, to 0 and carrying a 1 to the next; then setting the next, which is now $12 + 1 = 13$, to 3 and again carrying 1, and then finally stopping with the last digit at $3 + 1 = 4$.

Another important note here is that the loop does not need to carry precisely in order. A common practice is to have two loops interleaved, starting from different places: that is, with ten digits, one might carry from the first digit, then the fifth, then the second, then the sixth, and so on. The sequence of digits used for carrying is called the *carry chain*, and is one of the parameters left unspecified in our generic implementations.

2.2.2 Mixed-Radix Bases

Beyond unsaturated digits, there is a further integer representation technique for fast generalized Mersenne reduction. Suppose we are still working with $p = 2^{255} - 19$, but now on a 32-bit platform. We could use 15 17-bit registers, but that leaves almost half the bits in each register unused. The better way, it turns out, is to use a representation in base $2^{25.5}$, meaning that the ith digit has weight $2^{\lceil 25.5i \rceil}$. So, we could represent a 255-bit number x with 10 registers, such that

$$x = r_0 + 2^{26}r_1 + 2^{51}r_2 + 2^{77}r_3 + 2^{102}r_4 + 2^{128}r_5 + 2^{153}r_6 + 2^{179}r_7 + 2^{204}r_8 + 2^{230}r_9$$

This representation is called "mixed-radix", because it is neither base 2^{25} or 2^{26}, but some mix of the two. It is not necessary, even, that the radix needs to alternate every digit. For instance, to represent 127-bit numbers on a 64-bit platform, one could use three registers with base coefficients $2^{\lceil (42+1/3)i \rceil}$, so that a number x could be represented as:

$$x = r_0 + 2^{43}r_1 + 2^{85}r_2$$

The usefulness of mixed-radix bases and unsaturated limbs greatly influenced the design of our finite-field arithmetic library, since we needed to permit these kinds of representations and handle them as automatically as possible.

Chapter 3

Representing Multidigit Integers in Coq

In Chapter 2, we discussed how to represent large integers using smaller machine words. There is a second layer to this problem, in our context: we must also decide how to represent the machine representation in Coq. It is very important to make this choice well. The right representation makes subsequent definitions and proofs substantially more tractable.

3.1 Unsaturated

To represent the unsaturated integers described in Section 2.2.1, we will first present an obvious representation and then explain its shortcomings, which will motivate a second representation. In our development, we in fact did build the library with the first technique before completely rewriting it with the second.

3.1.1 Two-lists Representation

In our first attempt, we defined the large integers as combinations of two lists of integers: the *base* and the *limbs*. So, the number 437 in base 10 would be represented by the base [1;10;100] and limbs [7;3;4]. The number 10 in base 2 would be represented

by base [1;2;4;8] and limbs [0;1;0;1].

First, just to make the definitions a bit clearer, let's define the types **base** and **limbs**, as aliases for the type "list of integers" (`list Z`, in Coq syntax).

```
Definition base : Type := list Z.
Definition limbs : Type := list Z.
```

Now we need to define a way to convert from this representation back to Coq's infinite-precision integers; we will call this procedure *evaluating* the number. In this representation, evaluation is a dot product of base and limbs. For a base b and limbs x, we need to 1) create a list with ith term $(b[i] * x[i])$, and 2) add all the terms of the list together. We will use two functions defined on general lists, with the following types:

```
map2 : forall A B C : Type,
         (A -> B -> C) -> list A -> list B -> list C
fold_right : forall A B : Type, (B -> A -> B) -> B -> list A -> B
```

Our evaluation function, then, looks like this:

```
Definition eval (b:base) (x:limbs) : Z :=
  fold_right Z.add 0 (map2 Z.mul b x).
```

So far, so good. Now, we should define addition. To add two sets of limbs in the same base, all we need to do is add corresponding limbs together. It is tempting to just use `map2` as before, but with addition; however, it should be noted that `map2` truncates the longer of the two input lists. To fix this issue, we can pad the shorter list with zeroes, like so:

```
Definition add (x y : limbs) : limbs :=
  map2 Z.add
       (x ++ repeat 0 (length y - length x))
       (y ++ repeat 0 (length x - length y)).
```

Note here that `length` produces Coq's `nat` type (natural numbers), which can never be negative; therefore, if a subtraction would in integers produce a negative number, a zero is substituted. So, the longer list will be padded with `repeat 0 0`, which repeats zero zero times. The shorter list, however, will have a positive number,

the length difference, as the second argument to `repeat`, and thus be padded to have an equal length to the longer list.

Now we can write a quick example that shows our function behaves as expected:

```
Example five_plus_six :
  let base2 := [1;2;4;8;16] in
  let five := [1;0;1] in
  let six := [0;1;1] in
  eval base2 (add five six) = 11.
Proof. reflexivity. Qed.
```

To prove `add` correct, let's first introduce some lemmas about `eval`.

```
Lemma eval_nil_r x : eval nil x = 0. Proof. reflexivity. Qed.
Lemma eval_nil_l b : eval b nil = 0.
Proof. destruct b; reflexivity. Qed.

Lemma eval_cons b0 b x0 x:
  eval (b0::b) (x0::x) = b0*x0 + eval b x.
Proof. reflexivity. Qed.

(* automatically use all three lemmas in later proofs by writing
   [autorewrite with push_eval] *)
Hint Rewrite eval_nil_r eval_nil_l eval_cons : push_eval.
```

Now, some helper lemmas about `add`. The proof text is longer on these but not particularly important; it suffices to pay attention to the lemma statements.

```
Lemma add_nil_r y : add y nil = y.
Proof.
  cbv [add]; induction y; list_simpl; simpl;
    rewrite ?app_nil_r,?Nat.sub_0_r in *; rewrite ?IHy;
      f_equal; ring.
Qed.
Lemma add_nil_l y : add nil y = y.
Proof.
  cbv [add]; induction y; list_simpl; simpl;
    rewrite ?Nat.sub_0_r in *; rewrite ?IHy; reflexivity.
Qed.
```

```
Lemma add_cons x0 y0 x y :
  add (x0 :: x) (y0 :: y) = x0 + y0 :: add x y.
Proof. reflexivity. Qed.
```

Finally, we can prove the correctness theorem:

```
Theorem add_correct b : forall x y,
  (length x <= length b)%nat ->
  (length y <= length b)%nat ->
  eval b (add x y) = eval b x + eval b y.
Proof.
  induction b; intros; autorewrite with push_eval;
    try ring.
  destruct x,y; simpl map2; autorewrite with push_eval;
    try ring; simpl length in *.
  { rewrite add_nil_l. autorewrite with push_eval. ring. }
  { rewrite add_nil_r. autorewrite with push_eval. ring. }
  { rewrite add_cons. autorewrite with push_eval.
    rewrite IHb by omega. ring. }
Qed.
```

Notice a couple of things about the correctness theorem. First, we require preconditions about the lengths of the inputs related to the length of the base. Second, in order to prove the theorem, we have to do induction on the base, but also case analysis on the lengths of the input lists (indicated by the `destruct` keyword). Already things are looking a bit complicated, for such a simple operation. Multiplication, it turns out, is even more tricky.

To define a standard shift-and-add multiplication, we first need to define a shift operation. This task is not particularly straightforward, since we may be working with a mixed-radix base. So, in order to multiply a number by the nth base coefficient, for instance, it is not sufficient to just move the limbs by n places. Consider the base $[1; 2^3; 2^5; 2^8; 2^{10}; \ldots]$: "base $2^{2.5}$". If we have a three-limb number $x = x_0 + 2^3 x_1 + 2^5 x_2$, multiplying this number by 2^3 should give us $x = 2^3 x_0 + 2^6 x_1 + 2^8 x_2$. This number corresponds to the limbs $[0; x_0; 2x_1; x_2]$; note the coefficient 2 in the second term, which appears because the base is mixed-radix and multiplying two base

coefficients does not necessarily result in another base coefficient.

So, to define a shift operation, we first need to figure out what these coefficients are, then multiply the limbs by the correct coefficients and then shift them, like so:

```
(* Notation for getting an element from the base *)
Local Notation "b {{ n }}" := (nth_default 0 b n) (at level 70).
(* the ith term in shift_coefficients is (b[i] * b[n]) / b[i+n] *)
Definition shift_coefficients (b : base) (n : nat) :=
  map (fun i => (b{{i}} * (b{{n}})) / (b{{i + n}}))
      (seq 0 (length b)).
(* eval (shift b x n) = b[n] * eval x *)
Definition shift (b : base) (x : limbs) (n : nat) :=
  repeat 0 n ++ (map2 Z.mul x (shift_coefficients b n)).
```

Using the same example as before, and setting $x_0 = 0$, $x_1 = 1$, $x_2 = 1$, we can check that `shift` works as expected:

```
Example shift_ex1 :
  let b := [1;2^3;2^5;2^8;2^10] in
  let x := [0;1;1] in
  let x' := shift b x 1 in
  eval b x' = eval b [0;0;2;1].
Proof. reflexivity. Qed.
```

And now we can define `mul` in terms of `shift` and `add`:

```
(* multiply each limb by z *)
Definition scmul (z : Z) (x : limbs) := map (fun xi => z * xi) x.
(* multiply two multidigit numbers *)
Definition mul (b: base) (x y : limbs) : limbs :=
  fold_right (fun i => add (scmul (x{{i}}) (shift b y i)))
                           nil (seq 0 (length x)).
```

The correctness proofs for `shift` and `mul` are much more complex than that of `add`, so we will not reproduce them here. Much of the difficulty comes from the fact that we are constantly having to make terms (either the sums of corresponding limbs in `add`, or the partial products in `mul`) line up with each other and the base. Converting between bases would also require going through every limb and multiplying it by

something to make it line up with a base coefficient. These problems will motivate the next representation strategy, which is the one our project currently uses.

3.1.2 Associational and Positional Representations

Our key insight in creating a better representation was that all of this lining-up can be done with one definition, which takes a list of pairs (each with the compile-time coefficient and the runtime limb value) and a list representing the base, and produces a list of limbs. All of the arithmetic operations can be defined on the list-of-pairs representation, which can then be converted to the two-lists one. From here on, we will call the list-of-pairs the *associational* representation, since it directly associates the compile-time and runtime values, and the two-lists representation *positional*, because it infers the compile-time coefficients from the position of the runtime values in the list. For example, here are some representations of 437 in base 10:

- **Associational** : `[(1,7); (10,3); (100,4)]`
- **Associational** : `[(1,7); (100,4); (10,1); (10,2)]` (order and repetition of first coefficient do not matter)
- **Positional** : `[1,10,100]` and `[7,3,4]`

In this case, the compile-time values are 1, 10, and 100; 4, 3, and 7 are only known at runtime.

Let's start by defining `eval` in the associational format:

```
Module Associational.
  Definition eval (p:list (Z*Z)) : Z :=
    fold_right Z.add 0%Z (map (fun t => fst t * snd t) p).
```

This definition is pretty similar to the `eval` we defined earlier, except that instead of using `map2 Z.mul` we use `map (fun t => fst t * snd t)` to multiply the first and second elements of each pair together.

Now we can prove some lemmas about `eval`:

```
Lemma eval_nil : eval nil = 0. Proof. trivial. Qed.
Lemma eval_cons p q :
  eval (p::q) = fst p * snd p + eval q. Proof. trivial. Qed.
```

Addition, in this representation, is completely trivial; we just append two lists. The proof is quite straightforward:

```
Lemma eval_app p q: eval (p++q) = eval p + eval q.
Proof. induction p; rewrite <-?List.app_comm_cons;
         rewrite ?eval_nil, ?eval_cons; nsatz. Qed.
```

Defining `mul` is also fairly simple. Since we no longer need to make the values line up with any base, we just need to create partial products by getting each pair of pairs and multiplying both the first values and the second values. For instance, the partial product from the pairs $(2^3, x_1)$ and $(2^5, x_2)$ would be $(2^8, x_1x_2)$.

In order to take each term from one list and multiply it with all the terms of the other, we could use nested calls to `map`. However, since the inner `map` would produce `list (Z*Z)`, our final result would have the type `list (list (Z*Z))`; we want to concatenate all those inner lists into one. To perform this concatenation, we replace the outer `map` with the list function `flat_map`, which has the type:

```
flat_map : forall A B : Type, (A -> list B) -> list A -> list B
```

We also use the marker `%RT` to indicate that the multiplication of the second values should only be evaluated at runtime; this helps our simplification procedure later on (see Section 5.2). So the definition of `mul` is:

```
Definition mul (p q:list (Z*Z)) : list (Z*Z) :=
  flat_map
    (fun t => map (fun t' => (fst t*fst t', (snd t*snd t')%RT)) q)
    p.
```

The correctness proof can actually be shown here, unlike for the previous representation:

```
Lemma eval_map_mul (a x:Z) (p:list (Z*Z))
  : eval (List.map (fun t => (a*fst t, x*snd t)) p) = a*x*eval p.
Proof. induction p; push; nsatz. Qed.
```

```
Hint Rewrite eval_map_mul : push_eval.

Theorem eval_mul p q : eval (mul p q) = eval p * eval q.
Proof. induction p; cbv [mul]; push; nsatz. Qed.
```

We can also represent `split` and `reduce`. Recall from Section 2.1.1 that splitting a number involves separating the high and low bits, and is a subroutine of generalized Mersenne reduction.

```
Definition split (s:Z) (p:list (Z*Z)) : list (Z*Z) * list (Z*Z)
  := let hi_lo := partition (fun t => fst t mod s =? 0) p in
     (snd hi_lo, map (fun t => (fst t / s, snd t)) (fst hi_lo)).

Definition reduce (s:Z) (c:list _) (p:list _) : list (Z*Z) :=
  let lo_hi := split s p in fst lo_hi ++ mul c (snd lo_hi).
```

...and prove them correct:

```
Lemma eval_split s p (s_nz:s<>0) :
  eval (fst (split s p)) + s * eval (snd (split s p)) = eval p.
Proof. cbv [split]; induction p;
  repeat match goal with
  | |- context[?a/?b] =>
    unique pose proof (Z_div_exact_full_2 a b ltac:(trivial) ltac
       :(trivial))
  | _ => progress push
  | _ => progress break_match
  | _ => progress nsatz end. Qed.


Lemma reduction_rule a b s c (modulus_nz:s-c<>0) :
  (a + s * b) mod (s - c) = (a + c * b) mod (s - c).
Proof. replace (a + s * b) with ((a + c*b) + b*(s-c)) by nsatz.
  rewrite Z.add_mod,Z_mod_mult,Z.add_0_r,Z.mod_mod;trivial. Qed.

Lemma eval_reduce s c p (s_nz:s<>0) (modulus_nz:s-eval c<>0) :
  eval (reduce s c p) mod (s - eval c) = eval p mod (s - eval c).
Proof. cbv [reduce]; push.
```

```
      rewrite <-reduction_rule, eval_split; trivial.      Qed.
Hint Rewrite eval_reduce : push_eval.
```

So much for the associational representation. The positional representation is very similar in principle to the initial representation we described earlier, with three main differences. First, in order to stop carrying around proofs about list length, we will change the two-lists representation to use tuples instead, so we can require a particular length in the type signatures of functions. The type signature `tuple Z n` means an n-tuple of integers. Second, rather than requiring a list of base coefficients, we will require a weight function with type `nat -> Z`. Third, all of the arithmetic operations in the positional representation will be defined as 1) converting to associational, 2) performing the operation in the associational representation, and 3) converting back.

Even the positional `eval` will follow this pattern:

```
Definition eval {n} x :=
  Associational.eval (@to_associational n x).
```

However, we still have not discussed how exactly to convert between the positional and associational representations. One direction is trivial; converting from positional to associational is just a matter of matching up corresponding terms in the base and in the limbs:

```
Definition to_associational {n:nat} (xs:tuple Z n) : list (Z*Z)
  := combine (map weight (List.seq 0 n)) (Tuple.to_list n xs).
```

The other direction is trickier, since the compile-time coefficients might not line up with the runtime values. Roughly speaking, we need to look at the compile-time value of each pair, pick the highest base coefficient c such that their remainder is 0, and multiply the runtime value by their quotient before adding it to the correct position. For instance, suppose we need to place a term `(24,`x`)` into a base-8 positional list. Since $24 \bmod 8^1 = 0$ but $24 \bmod 8^2 \neq 0$, this term goes in position 1; since $24/8 = 3$, the value that gets added to position 1 is $3x$. The full definition is as follows (assuming a weight function with type `weight : nat -> Z`):

```
Fixpoint place (t:Z*Z) (i:nat) : nat * Z :=
  if dec (fst t mod weight i = 0)
```

```
  then (i, let c := fst t / weight i in (c * snd t)%RT)
  else match i with
       | O => (O, fst t * snd t)%RT
       | S i' => place t i' end.

Definition from_associational n (p:list (Z*Z)) :=
  fold_right (fun t =>
    let p := place t (pred n) in
    add_to_nth (fst p) (snd p)    ) (zeros n) p.
```

Here, `place` determines what value should be added at what index for each term. In `from_associational`, we use a list function `add_to_nth` to accumulate the term in the correct place.

The correctness proofs are relatively compact:

```
Lemma place_in_range (t:Z*Z) n : (fst (place t n) < S n)%nat.
Proof. induction n; cbv [place] in *; break_match; autorewrite
   with cancel_pair; try omega. Qed.
Lemma weight_place t i :
  weight (fst (place t i)) * snd (place t i) = fst t * snd t.
Proof. induction i; cbv [place] in *; break_match; push;
  repeat match goal with |- context[?a/?b] =>
    unique pose proof
       (Z_div_exact_full_2 a b ltac:(auto) ltac:(auto))
  end; nsatz. Qed.
Hint Rewrite weight_place : push_eval.

Lemma eval_from_associational {n} p (n_nz:n<>O) :
  eval (from_associational n p) = Associational.eval p.
Proof. induction p; cbv [from_associational] in *; push; try
pose proof place_in_range a (pred n); try omega; try nsatz. Qed.
```

Armed with these definitions, we can fairly easily define addition and modular multiplication.

```
Definition add (x y : tuple Z n) :=
  let x_a := to_associational x in
  let y_a := to_associational y in
```

```
  let xy_a := Associational.add x_a y_a in
  from_associational n xy_a.

Definition mulmod {n} s c (x y:tuple Z n) : tuple Z n
  := let x_a := to_associational x in
     let y_a := to_associational y in
     let xy_a := Associational.mul x_a y_a in
     let xym_a := Associational.reduce s c xy_a in
     from_associational n xym_a.
```

Using this definition, we can convert freely between positional and associational representations, and convert between bases trivially (by simply converting to associational and then back to positional, but with a different weight function). These representation changes impose no performance cost on the output, since all of the conversion details compile away once we know the exact base. However, reasoning this way significantly simplifies proofs and builds a stronger foundation for more complex developments like saturated arithmetic.

3.2 Saturated

Although our unsaturated library could synthesize arithmetic with base 2^{32} or 2^{64}, this code would overflow; the additions in `from_associational` assume that additions do not need to be carried. The assumption is true when we are using unsaturated arithmetic and have extra unused bits, but not for saturated arithmetic. Since some algorithms, most notably Montgomery modular reduction, require saturated arithmetic, we needed to implement an extension for this case. The key feature we needed was a way to go from associational to positional using an addition loop with carries instead of using non-carrying additions. To perform this procedure, we first had to accumulate all the values for each position as before, but not add them together immediately.

This need led us to a third representation, which we call `columns`. This representation has type `tuple (list Z) n`; for a known number of positions, it has a list

of values that align with each. We then use a two-output addition function, which produces both low and high bits, to add the values together and accumulate the high bits into a carry number that will be added to the next column. The function for adding together the values at a single position looks like this:

```
Fixpoint compact_digit n (digit : list Z) : (Z * Z) :=
  let cap := weight (S n) / weight n in
  match digit with
  | nil => (0, 0)
  | x :: nil => (modulo x cap, div x cap)
  | x :: y :: nil => add_get_carry cap x y
  | x :: tl =>
    let r := compact_digit n tl in
    let sum_carry := add_get_carry cap x (fst r) in
    (fst sum_carry, snd sum_carry + snd r)
  end.
```

In this case, `div` and `modulo` will later be instantiated with bitshifts, once it is clear that the weights (and their quotients) are powers of two. For each limb, we call this function to get the value for that position and the carry, append the carry to the next limb, and then repeat for the next position.

It is somewhat instructive to have an example here. Let's suppose we are multiplying 571 and 645 in base 10. At first, the partial products look like this (arranged above their corresponding weights):

```
              1*6
        1*4   7*4    7*6
  1*5   7*5   5*5    5*4     5*6
 ------------------------------------
   1     10    100   1000    10000
```

Arranged this way, it is clearer why we call this the "columns" representation. Multiplying each partial product gives us:

```
                    6
              4     28      42
        5     35    25      20       30
    ------------------------------------
        1     10    100     1000     10000
```

To process the first column, we call `compact_digit 0 [5]`. Since there is exactly one element in the list, and `weight 1 / weight 0 = 10`, the function returns `(modulo 5 10, div 5 10) = (5,0)`. Our output therefore will have a 5 in the first column; we move to the second with a carry of 0.

To process the second column, we append the carry to the list in that position and call `compact_digit 1 [0,4,35]`. This has more than two elements, so we hit the last case in the match and do a recursive call to `compact_digit 1 [4,35]`. This hits the exactly-two-element case and uses `add_get_carry` to get the high and low parts of the sum. Since $4 + 35 = 39$ and we are working in base 10, these are 3 and 9 respectively. We return from the recursive call with $r = (9, 3)$. Then we call `add_get_carry` on 0 and 9, which returns (9,0). The call finally returns $(9, 3)$ for the second column.

Here is the full tree of calls to `compact_digit`:

```
compact_digit 0 [5] => (5,0)


compact_digit 1 [0,4,35] => (9,3)
    compact_digit 1 [4,35] => (9,3)


compact_digit 2 [3,6,28,25] => (2,6)
    compact_digit 2 [6,28,25] => (9,5)
        compact_digit 2 [28,25] => (3,5)


compact_digit 3 [6,42,20] => (8,6)
    compact_digit 3 [42,20] => (2,6)
```

```
compact_digit 4 [6,30] => (6,3)
```

The end result is the first terms of all the non-recursive calls: $(5, 9, 2, 8, 6)$, and the last carry term, 3, which represents the number 368,295. A quick check on a calculator confirms the result.

Interestingly, and surprisingly, this major change to the output code required no change in the basic arithmetic operations. Everything is still done in associational representations; all we need to do is convert to columns and then to positional instead of directly to positional. The ease of this change was a strong signal to us that we had made the correct choice in terms of Coq representations; we never had to encode the same thinking twice.

Chapter 4

Summary of Operations

In this chapter, we will list the significant finite-field operations provided by our library and provide succinct explanations of any relevant implementation details.

4.1 Unsaturated

These operations are written for unsaturated arithmetic and are defined on operands that are in the positional representation. They typically operate by converting to associational and then converting back using the `from_associational` definition described in Section 3.1.2.

add

This operation converts to associational, performs associational addition as described in Section 3.1.2, and converts back to positional using `from_associational`.

sub

Subtraction takes three inputs: the two operands and another input which is a multiple of the modulus. This third input is added to the first operand using `add`, helping to avoid underflow in case the second operand is larger than the first in some limb; since we added a multiple of the modulus, the modular value is preserved. Next, the

second operand is negated; a function runs over the associational list of pairs and replaces each pair (w, x) with $(w, -x)$. Finally, we `add` the sum of the first and third operand to the negated second operand.

unbalanced_sub

This operation is similar to `sub` but does not make use of a multiple of the modulus. It simply negates the second operand and combines it with the first using `add`. It is only safe to use `unbalanced_sub` in cases where we know for sure that the second operand is smaller than the first in all limbs.

mul

This operation converts to associational, performs associational multiplication as described in Section 3.1.2, and converts back to positional using `from_associational`.

reduce

This operation converts to associational, performs associational `reduce` as described in Section 3.1.2, and converts back to positional using `from_associational`.

split

This operation converts to associational, performs associational `split` as described in Section 3.1.2, and converts back to positional using `from_associational`.

carry

This operation performs the delayed carrying technique described in Section 2.2.1. It takes an operand in positional format and an index which indicates the limb to carry from. Like other operations, it first converts the input to associational format, which may be a surprising move. Associational format does not "know" what the final weights are, so the operation has to be passed, explicitly, the weight w of the limb

to be carried from and the quotient fw of the weight of the next limb and w. So, if we were operating in base 10 and carrying from the limb at index 3, $w = 10^3 = 1000$ and $fw = 10000/1000 = 10$.

For each pair (w', x) in the associational format, `carry` checks whether $w' = w$. If not, the limb is left in the list unchanged. Otherwise, `carry` replaces the pair from the list and adds two new pairs: $(w, x \bmod fw)$ and $(w \cdot fw, x/fw)$. Then, we can convert back to positional, and the new terms will be accumulated into the correct positions.

chained_carries

This is a wrapper definition which takes in a list of indices and carries in that order. If the limb to carry from is the very last limb, `chained_carries` also calls `reduce` to eliminate the extra limb that will be produced by `carry`.

scmul

This operation takes an operand in positional representation and a scalar value x. It converts the operand to associational format and performs associational multiplication with $[(1, x)]$. Then, it converts back to positional format, producing an output in which all limbs have been multiplied by x.

select

This operation takes an operand in positional representation and a scalar value, which is treated as a conditional. If the conditional is 0, `select` returns a positional output with the same number of limbs as the input, but with zeroes as all the limb values. If the conditional is nonzero, the input is unchanged. The trick to `select` is to perform the operation in a non-branching way, to prevent timing side channels.

`select` works by using the conditional and (a functional implementation of) the `cmovnz` instruction to select either 0 or a mask with all bits set. Then, `select` uses a bitwise `and` to combine that value with each limb, and returns the result.

4.2 Saturated

Like the unsaturated operations, saturated arithmetic expects input in positional format and also often operates through an associational intermediate step, but the process for converting from associational is different. Instead of the `from_associational` defined in Section 3.1.2, we instead use an operation called `compact` that performs a loop of calls to the `compact_digit` operation described in Section 3.2.

Many of the specialized operations exposed here are motivated by the requirements of Montgomery modular reduction. They are not technically confined to saturated arithmetic but are put in this section because Montgomery is used on saturated limbs only.

sat_add

This operation chains carrying additions to add together two numbers in positional format without overflow. It produces a result and a carry bit.

sat_sub

This operation chains carrying subtractions to subtract two numbers in positional format without underflow. It produces a result and a carry ("borrow") bit.

unbalanced_sub

Works identically to the unsaturated version, except for converting back to positional representation using `compact`.

mul

Works identically to the unsaturated version, except for converting back to positional representation using `compact`.

conditional_add

This operation takes two operands and a conditional. It uses `select` and the conditional to get either the second operand or zeroes, then adds the result to the first operand. Thus, if the conditional is zero, the operation is a no-op; otherwise, it adds the two operands.

join0

This operation adds a new limb to the most significant end of the input, with runtime value zero.

divmod

This operation takes an operand in positional format and an index at which to split. It produces the high and low limbs, on either side of the index, as a pair.

drop_high

This operation removes the highest limb of an operand in positional format.

scmul

This operation takes an operand in positional format and a scalar value x, and uses saturated `mul` to multiply the operand with x as a single-limbed positional number.

sub_then_maybe_add

This operation takes three operands, p, q, and r, and returns $p - q$ if $p \geq q$ and $p - q + r$ otherwise. We implement it by chaining `sat_sub` and `conditional_add`, using the negation of the borrow bit from `sat_sub` as the conditional.

conditional_sub

This operation takes operands p and q and returns $p - q$ if $p \geq q$ and p otherwise. We perform the subtraction in any case to avoid branches and then use the borrow bit to select which output to return.

4.3 Other

These operations are composed of those described in the previous sections and can often be used for either saturated or unsaturated arithmetic. In some cases, like `freeze`, they use both kinds of operation.

pow

For exponentiation by a constant, we go beyond the *binary exponentiation* that would, for instance, express x^{15} with 6 multiplications as $x^{15} = x \times (x \times (x \times x^2)^2)^2$. Instead, we parameterize exponentiation over arbitrary *addition chains* that encode the sharing of subcomputations, so that we can compute x^{15} with only 5 multiplications as $x^{15} = x^3 \times ((x^3)^2)^2$. Figuring out what these addition chains should be is very nontrivial, but binary exponentiation can easily be encoded in this form.

inv

For inversion (finding a multiplicative inverse in the field), we use Fermat's Little Theorem (whose proof we import from the Coqprime [21] library) to show that a^{p-2} is the multiplicative inverse of a modulo p, and use our exponentiation routine.

sqrt

For finding square roots (modulo prime p), we employ prime-specific tricks for fast execution based on Euler's criterion. For instance, when $p \equiv 3 \mod 4$, the square root of a (if a has a square root) is given by $a^{\frac{p+1}{4}}$, again relying on exponentiation.

freeze

This operation "canonicalizes" unsaturated limbs. Because unsaturated formats allow multiple representations of the same number, it is necessary to convert them to a form that does not leak information through choice of representation. Furthermore, canonicalization must be done in constant time, which we accomplished by using the technique used in the code for Ed448-Goldilocks.

This technique involves first subtracting the modulus using a sub-with-borrow loop (in our case, `sat_sub`). If the final borrow is negative, we add the modulus back; otherwise, we add zero. The combination of a modular reduction for large values and a sequence of carries forces the saturated limbs not to use the extra digits available; for instance, if we are using base 2^{51} with 64-bit integers, all of the limbs should have runtime values less than 2^{51} by the end of `freeze`.

karatsuba_mul

This operation is described in detail in Chapter 6; it implements multiplication using Karatsuba's algorithm.

Chapter 5

Producing Parameter-Specific Code

So far, all of the Coq code we have shown has been parameterized over the choice of finite field and mathematical base (even the saturated code is parameterized over what the machine word size is). As we have seen, there are many mathematical transformations that can be performed at this level. However, once we have done all we can in general, we progress to another phase, in which we perform transformations that are specific to the piece of code being produced. For instance, we remove the dynamic allocation of lists, replacing a loop over all entries in a length-n list with some specific number of statements.

Ultimately, we will transform the functional program into an abstract syntax tree made up of simple operations like addition and multiplication of single registers, bitshifts, and memory allocation, with finite-sized integers. This code can be pretty-printed as runnable C, like Figure 5-1.

We do this transformation in four phases, as follows:

- Parameter selection: Based on the modulus, specify remaining parameters for how the code should be structured (e.g. number of limbs).
- Partial evaluation: inline function definitions until only a few types of functions (e.g. bitshifts) remain.
- Linearization/constant folding: remove nested variable assignments and simplify using arithmetic identities.

- Bounds inference: Track upper and lower bounds on each variable and, from a list of possible integer sizes, assign it a finite size.

Each of these steps will now be explained in more detail.

5.1 Selecting Parameters

The generic finite-field arithmetic code requires the instantiation of a number of compile-time parameters:

- Number of limbs
- Allowable bitwidths of target architecture
- Carry chains (see Section 2.2.1)
- Subtraction balance coefficient (see summary of unsaturated `sub` in Chapter 4)

In addition, there are some parameters to be specified for the simplification process itself. For instance, we declare which finite-field operations should be generated, whether the implementation should use karatsuba multiplication, and whether to use Montgomery or generalized Mersenne reduction.

Sometimes, if we are working with a well-known implementation for which the most performant choices of these parameters are known, we might want to specify these things manually. However, most of the time, the decisions are quite mechanical, or there are a few clear guesses to make. To that end, our development includes a python script which, given the modulus, determines reasonable guesses at these parameters. Our benchmark results, as shown in Figure 0-2 and Appendix A, are the result of applying this script to primes scraped from the archives of the mailing list `curves@moderncrypto.org`.

The script, for each prime, picks two reasonable choices of limb counts each for generalized Mersenne implementations on both 32- and 64-bit systems, as well as choosing the minimal number of limbs for Montgomery implementations. It also

```
static void femul(uint64_t out[5], const uint64_t in1[5], const
   uint64_t in2[5]) {
  const uint64_t x10 = in1[4];
  const uint64_t x11 = in1[3];
  const uint64_t x9 = in1[2];
  const uint64_t x7 = in1[1];
  const uint64_t x5 = in1[0];
  const uint64_t x18 = in2[4];
  const uint64_t x19 = in2[3];
  const uint64_t x17 = in2[2];
  const uint64_t x15 = in2[1];
  const uint64_t x13 = in2[0];
  uint128_t x20 = (((uint128_t)x5 * x18) + (((uint128_t)x7 * x19) +
     (((uint128_t)x9 * x17) + (((uint128_t)x11 * x15) + ((uint128_t)
     x10 * x13)))));
  uint128_t x21 = ((((uint128_t)x5 * x19) + (((uint128_t)x7 * x17) +
      (((uint128_t)x9 * x15) + ((uint128_t)x11 * x13)))) + (0x13 *
     ((uint128_t)x10 * x18)));
  uint128_t x22 = ((((uint128_t)x5 * x17) + (((uint128_t)x7 * x15) +
      ((uint128_t)x9 * x13))) + (0x13 * (((uint128_t)x11 * x18) + ((
     uint128_t)x10 * x19))));
  uint128_t x23 = ((((uint128_t)x5 * x15) + ((uint128_t)x7 * x13)) +
      (0x13 * (((uint128_t)x9 * x18) + (((uint128_t)x11 * x19) + ((
     uint128_t)x10 * x17)))));
  uint128_t x24 = (((uint128_t)x5 * x13) + (0x13 * (((uint128_t)x7 *
      x18) + (((uint128_t)x9 * x19) + (((uint128_t)x11 * x17) + ((
     uint128_t)x10 * x15))))));
  uint64_t x25 = (uint64_t) (x24 >> 0x33);
  uint64_t x26 = ((uint64_t)x24 & 0x7fffffffffff);
  uint128_t x27 = (x25 + x23);
  uint64_t x28 = (uint64_t) (x27 >> 0x33);
  uint64_t x29 = ((uint64_t)x27 & 0x7fffffffffff);
  uint128_t x30 = (x28 + x22);
  uint64_t x31 = (uint64_t) (x30 >> 0x33);
  uint64_t x32 = ((uint64_t)x30 & 0x7fffffffffff);
  uint128_t x33 = (x31 + x21);
  uint64_t x34 = (uint64_t) (x33 >> 0x33);
  uint64_t x35 = ((uint64_t)x33 & 0x7fffffffffff);
  uint128_t x36 = (x34 + x20);
  uint64_t x37 = (uint64_t) (x36 >> 0x33);
  uint64_t x38 = ((uint64_t)x36 & 0x7fffffffffff);
  uint64_t x39 = (x26 + (0x13 * x37));
  uint64_t x40 = (x39 >> 0x33);
  uint64_t x41 = (x39 & 0x7fffffffffff);
  uint64_t x42 = (x40 + x29);
  uint64_t x43 = (x42 >> 0x33);
  uint64_t x44 = (x42 & 0x7fffffffffff);
  out[0] = x41;
  out[1] = x44;
  out[2] = (x43 + x32);
  out[3] = x35;
  out[4] = x38; }
```

Figure 5-1: Final output for 64-bit curve25519 finite field multiplication

suggests a carry chain based on a rough heuristic and provides a default choice of subtraction balance coefficient.

5.2 Partial Evaluation

Once we plug in the parameters, we can begin applying transformations that rely on the compile-time constants we now know. The first of these is to collapse our carefully constructed layers of abstraction by unrolling loops and inlining all high-level functions.

5.2.1 Synthesis Mode

To redefine operations with equivalent, simplified versions, we use a method called "synthesis mode", which was first developed in the Fiat [8] project. I will provide a short explanation; a more detailed walkthrough can be found in Andres Erbsen's Master's thesis [10].

Let's suppose we are simplifying an operation `foo`. The definition is as follows:

```
Definition foo (a : nat) (b c : Z) : Z :=
  fold_right (fun next st => st * next) c (repeat b a).
```

The loop simply starts with c and multiplies it by b, a times, producing $c \cdot b^a$. We want to be able to transform this expression arbitrarily, as long as we retain the proof that it is equivalent to the original. In our project, it is often the case that we want to do these transformations because some of the inputs have been instantiated. For instance, arithmetic operations are initially generic over the target-architecture bitwidth and the modulus, which we know by compile time. To simulate that situation, we will say we know the first two inputs for `foo`; $a = 3$ and $b = 4$.

Now, as shown in Figure 5-2, we tell Coq we are defining something with a *sigma type*. That is, rather than just defining the operation, we include in the type a requirement for a proof about the operation. In this case, the proof says that for all c, produces the same result as `foo 3 4 c`. The comments in between lines show the state of the proof. At first, we have our original goal; an operation `foo'` that is

```
Definition foo34_sig : { foo' | forall c, foo' c = foo 3 4 c }.
Proof.
  (* {foo' : Z -> Z | forall c : Z, foo' c = foo 3 4 c} *)
  eexists; intro c.
  (* ?foo' c = foo 3 4 c *)
  cbv - [Z.mul].
  (* ?foo' c = c * 4 * 4 * 4 *)
  ring_simplify.
  (* ?foo' c = 64 * c *)
  reflexivity.
Defined.

(* Inspect the result *)
Eval cbv [foo34_sig proj1_sig] in (proj1_sig foo34_sig).
(* = Z.mul 64 *)
```

Figure 5-2: Synthesis mode for the example operation `foo`

equivalent to `foo 3 4`. Using `eexists`, we essentially delay telling Coq exactly what `foo'` is; we will instantiate it via a proof. Then we perform two simplifications; first we compute away everything except multiplication of integers, then we run a routine `ring_simplify`, a tactic from Coq's standard library that simplifies arithmetic expressions on rings, like the integers. Finally, we are left with `?foo' c = 64 * c`. Without notation, this is `?foo' c = Z.mul 64 c`. Calling `reflexivity` tells Coq the two sides should match; therefore, `foo'` is instantiated with `Z.mul 64`.

5.2.2 Marking Runtime Operations

The main challenge involved with partial evaluation is making sure that only certain functions and variable assignments get inlined. For instance, multiplications of constants should be unfolded, but multiplications of runtime values with constants should not. Some variable assignments must be inlined to simplify the expression; however, inlining all of them will result in many common subexpressions.

Since we know when writing the high-level functions which values will be known at compile time, we choose to fix this issue by marking certain operations and assignments explicitly. We do this by creating wrapper definitions. For instance, for integer multiplication, we write:

```
Definition runtime_mul := Z.mul.
Infix "*" := runtime_mul : runtime_scope.
Delimit Scope runtime_scope with RT.
```

This means that while `(x * y)` is still interpreted as `Z.mul x y`, `(x * y)%RT` is interpreted as `runtime_mul x y`. So, if we have the expression `fun x => (x * (2 * 3))%RT` and we write `cbv -[runtime_mul]`, we get `x * 6`; the multiplication of 2 and 3 is computed, but the multiplication of `x` and 6 is not.

Similarly, for variable assignments, a wrapper definition for `let`, called `Let_In` and denoted by `dlet`, lets us mark which variable assignments not to inline:

```
Eval cbv - [runtime_mul runtime_add Let_In] in
  (fun x => dlet y := (x * x)%RT in
            let z := (y + y)%RT in
            (z + 1)%RT).
```

This code produces `dlet y := (x * x)%RT in ((y + y)%RT + 1)%RT`, rather than the fully inlined `(((x * x)%RT + (x * x)%RT)%RT + 1)%RT`. This technique helps us avoid inlining too many common subexpressions. It is tempting to think that we could just inline everything and then eliminate common subexpressions later; however, this strategy results in exponentially sized terms that, at our scale, far surpass Coq's limitations.

5.2.3 Continuation-Passing Style

However, preserving `let`s has some unfortunate consequences. If `dlet`s get trapped inside inner terms (for instance, elements of a list or arguments to a function), then the function will not simplify correctly. We fix this snag by using a technique called continuation-passing style (CPS) [20]. A CPS function requires a *continuation* argument, which is similar to a callback; it is the next action to be performed and takes the CPS function's result as an argument. This concept is best explained by example.

Let's suppose we want to add corresponding elements of two `list Z`s (so on inputs `[1,2,3]` and `[2,3,1]`, we get `[3,5,4]`). We might write the function like this:

```
Fixpoint add_lists (p q : list Z) :=
```

```
  match p, q with
  | p0 :: p', q0 :: q' =>
    dlet sum := p0 + q0 in
    sum :: add_lists p' q'
  | _, _ => nil
  end.
```

A CPS equivalent of `add_lists` would look like this:

```
Fixpoint add_lists_cps (p q : list Z) {T} (k:list Z->T) :=
  match p, q with
  | p0 :: p', q0 :: q' =>
    dlet sum := p0 + q0 in
    add_lists_cps p' q' (fun r => k (sum :: r))
  | _, _ => k nil
  end.
```

In this case, `k` is the continuation; it is a function from the output of the list-adding function to some other unknown type. Because we can access `k` explicitly, we can make sure that `dlet`s do not end up in its arguments.

We define two specific expressions, one using `add_lists` and the other using `add_lists_cps`:

```
Definition x :=
    (fun a0 a1 a2 b0 b1 b2 =>
      let r := add_lists [a0;a1;a2] [b0;b1;b2] in
      let rr := add_lists r r in
      add_lists rr rr).

Definition y :=
    (fun a0 a1 a2 b0 b1 b2 =>
      add_lists_cps [a0;a1;a2] [b0;b1;b2]
                    (fun r => add_lists_cps r r
                    (fun rr => add_lists_cps rr rr id))).
```

These expressions are equivalent; if we run `Eval cbv -[Z.add] in x` and `Eval cbv -[Z.add] in y`, we get identical output:

```
fun a0 a1 a2 b0 b1 b2 : Z =>
```

```
       [a0 + b0 + (a0 + b0) + (a0 + b0 + (a0 + b0));
       a1 + b1 + (a1 + b1) + (a1 + b1 + (a1 + b1));
       a2 + b2 + (a2 + b2) + (a2 + b2 + (a2 + b2))]
```

However, there are a lot of common subexpressions, which is what `dlet` is designed to avoid. Let's try `Eval cbv -[Let_In Z.add] in x`:

```
fun a0 a1 a2 b0 b1 b2 : Z =>
       (fix add_lists (p q : list Z) {struct p} :
        list Z :=
          match p with
          | [] => []
          | p0 :: p' =>
              match q with
              | [] => []
              | q0 :: q' =>
                  dlet sum := p0 + q0 in
                  sum :: add_lists p' q'
              end
          end)
         ((fix add_lists (p q : list Z) {struct p} :
           list Z :=
             match p with
             ...
             [30 similar lines]
             ...
             end)
            (dlet sum := a0 + b0 in
             sum
             :: (dlet sum0 := a1 + b1 in
                 sum0 :: (dlet sum1 := a2 + b2 in
                          [sum1])))
            (dlet sum := a0 + b0 in
             sum
             :: (dlet sum0 := a1 + b1 in
                 sum0 :: (dlet sum1 := a2 + b2 in
                          [sum1]))))
```

Clearly, the expression failed to simplify; if we look at the list near the end of the output we can see why. The `dlet`s are obscuring how long the list is, so the match expressions cannot be simplified. However, if we try the same command on the CPS version (`Eval cbv -[Let_In Z.add] in y`), we get:

```
fun a0 a1 a2 b0 b1 b2 : Z =>
       dlet sum := a0 + b0 in
       dlet sum0 := a1 + b1 in
       dlet sum1 := a2 + b2 in
       dlet sum2 := sum + sum in
       dlet sum3 := sum0 + sum0 in
       dlet sum4 := sum1 + sum1 in
       dlet sum5 := sum2 + sum2 in
       dlet sum6 := sum3 + sum3 in
       dlet sum7 := sum4 + sum4 in
       [sum5; sum6; sum7]
```

This version simplifies much better; because we can control where the `dlet`s end up, we can keep them from getting in the way of match expressions.

5.3 Linearization and Constant-Folding

Note: *The work in this section and Section 5.4 was primarily conducted by my collaborator Jason Gross; I am describing it because it provides important context for choices made in my own work.*

After the partial-evaluation phase, we have more work to do before the program is in the right form to do bounds-inference. Consider the example program:

```
let (x_1, x_2, x_3) := x in
let (y_1, y_2) := ((let z := x_2 * 1 * x_3 in z + 0), x_2) in
y_1 * y_2 * x_1
```

The linearization phase removes nested lets and cancels out internal tuples, producing the following expression:

```
let (x_1, x_2, x_3) := x in
let z := x_2 * 1 * x_3 in
```

```
let y_1 := z + 0 in
y_1 * x_2 * x_1
```

Next, the constant-folding phase simplifies according to arithmetic identities and inlines constants.

```
let (x_1, x_2, x_3) := x in
let z := x_2 * x_3 in
z * x_2 * x_1
```

The expression now has a much simpler structure and is suitable for the next stage.

5.4 Bounds Inference

Until this point, all our work has been using infinite-precision integers, and we have not reasoned at all about overflow or underflow. In the bounds-inference stage, we transition to finite-sized integers by analyzing each variable and assigning it the minimum possible finite size from a list of size options. (For instance, if our architecture supports 64-bit and 128-bit integers, the sizes would be 64 and 128.) The analysis is fairly simple, just tracking upper and lower bounds on each variable, and is implemented as a certified compiler pass rather than with manual proofs. If any variable goes below zero or above the maximum size, the compilation stage will fail.

It is important to note here that bounds-inference works only when our infinite-precision integer code is written with finite-sized integers in mind. One might ask why we do not reason about finite-precision integers from the start; the reason is twofold. First, the bounds on a function input might vary depending on context; for instance, we cannot in general add two 64-bit numbers and get a 64-bit result, but we can if there are upper bounds on those numbers that sum to less than 2^{64}. Second, adding preconditions about bounds would increase the complexity of our proofs greatly.

Once bounds inference is complete, we have a simplified and linearized piece of code with finite-sized integers, which can be easily printed as straightline C as in Figure 5-1.

Chapter 6

Extending with New Arithmetic Techniques

So far, we have discussed the development of an arithmetic library that applies standard techniques to all parameter choices. However, in some cases, an arithmetic technique only applies to a few parameter choices; we need to have a flexible enough design so that we can extend the library with these techniques. As an example, I will walk through the motivation and implementation of Karatsuba multiplication for Ed448-Goldilocks [12].

6.1 Algorithm

The prime for Ed448-Goldilocks is $2^{448} - 2^{224} - 1$. It is often represented with 8 limbs of 56 bits each, on 64-bit architectures, or 16 limbs of 28 bits on 32-bit ones. Let's suppose we want to multiply two numbers x and y modulo this prime, represented in 8 limbs each:

$$x = x_0 + 2^{56}x_1 + 2^{112}x_2 + 2^{168}x_3 + 2^{224}x_4 + 2^{280}x_5 + 2^{336}x_6 + 2^{392}x_7$$
$$y = y_0 + 2^{56}y_1 + 2^{112}y_2 + 2^{168}y_3 + 2^{224}y_4 + 2^{280}y_5 + 2^{336}y_6 + 2^{392}y_7$$

Using the techniques described in previous chapters, we would do a multiplication

that created 64 partial products and then align them to 16 weights from 1 to $2 \cdot 448$. Then we would split at weight 2^{448} and do a generalized Mersenne reduction. However, with the Goldilocks prime, we can do better. We can do a modular reduction to reorganize the terms in the multiplication:

$$\begin{aligned} x \cdot y &= (a + 2^{224}b) \cdot (c + 2^{224}d) \\ &= ac + 2^{224}(ad + bc) + 2^{448}bd \\ &= ac + 2^{224}(ad + bc) + (2^{448} - 2^{224} - 1)bd + 2^{224}bd + bd \\ &\equiv (ac + bd) + 2^{224}(ad + bc + bd) \pmod{p} \\ &= (ac + bd) + 2^{224}((a + b)(c + d) - ac) \end{aligned}$$

Since a, b, c, and d have four limbs each, the multiplications ac, bd, and $(a+b)(c+d)$ have 16 partial products each, meaning 48 partial products total instead of the 64 one would get without the Karatsuba rearrangement. However, this step relies upon the structure of the prime. We could not split a prime that did not have an intermediate term like 2^{224} here, one that was on a limb boundary and about half the size of the prime. It is also not necessarily true that the bounds would all work out given that we are doing several steps of addition and even multiplication without carrying; since this curve's 56-bit base has 8 bits of slack per limb, we do not overflow in this case.

6.2 Implementation

Expressing and proving this logic is no more complicated than in the previous chapter; we can reuse the unsaturated addition and multiplication routines we have already defined. Even the `split` operation can be reused. Our top-level system can look at the structure of the prime and set a flag specifying that we should use Karatsuba, which simply means that, instead of starting partial evaluation on the usual multiplication function, we use the Karatsuba definition. So ultimately, the integration of this algorithm into our system is fairly smooth.

There is one complexity involved, however, which has to do with the subtraction of

ac from $(a+b)(c+d)$ in the multiplication expression. Generally, to prevent underflow in modular subtraction, we add a multiple of the modulus before subtracting. In this case, there is no need for that step; we can prove arithmetically that each limb of $(a+b)(c+d)$ is greater than each limb of ac. The proof sketch is as follows:

- Every partial product $x_i y_i$ in ac has a corresponding partial product in the same location in $(a+b)(c+d)$, which has the form $(x_i + x_{i+4})(y_i + y_{i+4})$.
- Since all limbs are positive, there is a strict inequality between corresponding partial products: $(x_i + x_{i+4})(y_i + y_{i+4}) = x_i y_i + x_i y_{i+4} + x_{i+4} y_i + x_{i+4} y_{i+4} > x_i y_i$.
- These corresponding partial products will always be accumulated into the same limbs as each other, so since all of ac's partial products are smaller than all of those in $(a+b)(c+d)$, each limb of ac will be smaller than each limb of $(a+b)(c+d)$.

However, the fancy reasoning caused a problem for our automatic bounds checker. Recall from Section 5.4 that our bounds checker works by tracking possible ranges for the value of each variable and using a set of rules to calculate the bounds after a given operation. So, for subtraction, if some integer with range $[4, 16]$ was subtracted from an integer with range $[16, 32]$, the output would have range $[0, 28]$. This rule does not take into account whether there is an algebraic relationship between the integers. Therefore, the bounds checker could not guarantee that subtracting the limbs of ac from those of $(a+b)(c+d)$ would not underflow.

To work around this issue we wrapped that particular subtraction in a special operation called `id_with_alt_bounds`, which took as arguments two expressions and a proof that they were arithmetically equal. When unfolded, this operation was actually an identity function on the first expression. However, this allowed the bounds checker to safely check the second expression in order to get a guarantee for the first. We could put as the second expression something equivalent but slow: in this case, $ad + bc + bd$ instead of $(a+b)(c+d) - ac$.

The bounds checker remains sound with this modification because we produce a proof obligation that all variables are literally equal in the two versions, and this proof

obligation uses bounded words rather than arbitrary-precision integers. Therefore, under- or overflow in intermediate computations that make the expressions non-equal would not pass.

6.3 Overall Effort of Extension

Adding new arithmetic techniques to the library has in our experience been nontrivial but manageable. Both Karatsuba and saturated arithmetic required significant knowledge of Coq and a detailed understanding of the techniques involved. However, neither required a complete rewrite, and we were surprised at the level of reusability we got out of the unsaturated code (for instance, reusing `split` in Karatsuba).

Appendix A

Performance Tables

These are our full benchmark results from generating finite-field arithmetic from the primes we scraped off of `curves@moderncrypto.org` archives. Our batch 64-bit trials run on an x86 Intel Haswell processor, while 32-bit trials run on an ARMv7-A Qualcomm Krait (LG Nexus 4). Benchmark time is measured for 1000 sequential computations of a Montgomery ladder step. For each configuration, we show whichever of our two synthesized strategies (Solinas vs. Montgomery) gives better performance. We see a significant performance advantage for our code, even compared to the GMP version that "cheats" by leaking secrets through timing. Speedups range between 1.25X and 9X.

Table A.1: Full 64-bit benchmark data. Our code tried both Solinas and Montgomery implementations for each prime, and we test against three GMP-based implementations: one that is constant-time (gmpsec), one that is variable time (gmpvar), and GMP's C++ API. Our code is constant-time, so gmpsec is the best comparison; however, even with that constraint removed from GMP and not us, we compare favorably to gmpvar.

Prime	**Our Code**		**GMP Code**			**Speedup**
	Sol.	**Mont.**	**const time**	**var time**	**C++**	
$2^{127} - 1$	0.03	0.04	0.26	0.15	0.67	5.0
$2^{129} - 25$	0.03	0.07	0.38	0.27	0.8	9.0
$2^{130} - 5$	0.03	0.09	0.39	0.28	0.79	9.33
$2^{137} - 13$	0.03	0.08	0.37	0.27	0.8	9.0
$2^{140} - 27$	0.03	0.08	0.38	0.27	0.8	9.0
$2^{141} - 9$	0.03	0.08	0.39	0.27	0.83	9.0
$2^{150} - 3$	0.03	0.08	0.38	0.3	0.8	10.0
$2^{150} - 5$	0.03	0.08	0.39	0.29	0.84	9.67
$2^{152} - 17$	0.03	0.08	0.38	0.27	0.82	9.0
$2^{158} - 15$	0.03	0.08	0.37	0.27	0.76	9.0
$2^{165} - 25$	0.03	0.08	0.38	0.27	0.78	9.0
$2^{166} - 5$	0.03	0.08	0.39	0.27	0.79	9.0
$2^{171} - 19$	0.03	0.08	0.38	0.27	0.79	9.0
$2^{174} - 17$	0.03	0.08	0.38	0.28	0.78	9.33
$2^{174} - 3$	0.03	0.08	0.38	0.27	0.78	9.0
$2^{189} - 25$	0.04	0.08	0.39	0.27	0.8	6.75
$2^{190} - 11$	0.04	0.08	0.38	0.27	0.78	6.75
$2^{191} - 19$	0.04	0.09	0.36	0.26	0.78	6.5
$2^{192} - 2^{64} - 1$	0.05	0.07	0.31	0.24	0.79	4.8
$2^{194} - 33$	0.04	0.12	0.5	0.34	0.93	8.5

Prime	Our Code		GMP Code			Speedup
	Sol.	Mont.	const time	var time	C++	
$2^{196} - 15$	0.04	0.12	0.5	0.34	0.89	8.5
$2^{198} - 17$	0.04	0.12	0.51	0.34	0.87	8.5
$2^{205} - 45 \cdot 2^{198} - 1$	-	0.14	0.51	0.34	0.87	2.43
$2^{206} - 5$	0.04	0.14	0.5	0.34	0.84	8.5
$2^{212} - 29$	0.05	0.12	0.49	0.35	0.87	7.0
$2^{213} - 3$	0.04	0.13	0.49	0.37	0.88	9.25
$2^{216} - 2^{108} - 1$	0.04	0.12	0.51	0.35	0.88	8.75
$2^{221} - 3$	0.05	0.15	0.51	0.36	0.89	7.2
$2^{222} - 117$	0.05	0.12	0.53	0.35	0.91	7.0
$2^{224} - 2^{96} + 1$	-	0.13	0.5	0.35	0.88	2.69
$2^{226} - 5$	0.04	0.13	0.5	0.35	0.92	8.75
$2^{230} - 27$	0.05	0.13	0.54	0.35	0.91	7.0
$2^{235} - 15$	0.06	0.13	0.5	0.34	0.89	5.67
$2^{243} - 9$	0.06	0.13	0.5	0.34	0.89	5.67
$2^{251} - 9$	0.06	0.13	0.5	0.35	0.94	5.83
$2^{254} - 127 \cdot 2^{240} - 1$	-	0.12	0.5	0.35	0.92	2.92
$2^{255} - 19$	0.06	0.13	0.48	0.35	0.9	5.83
$2^{255} - 765$	0.06	0.13	0.52	0.34	0.9	5.67
$2^{256} - 189$	0.06	0.14	0.38	0.34	0.87	5.67
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	-	0.11	0.38	0.33	0.84	3.0
$2^{256} - 2^{32} - 977$	0.1	0.12	0.38	0.34	0.87	3.4
$2^{256} - 4294968273$	0.14	0.13	0.37	0.34	0.86	2.62
$2^{256} - 88 \cdot 2^{240} - 1$	-	0.11	0.39	0.34	0.88	3.09
$2^{266} - 3$	0.06	0.18	0.66	0.45	1.13	7.5
$2^{285} - 9$	0.06	0.18	0.73	0.43	0.97	7.17
$2^{291} - 19$	0.07	0.18	0.68	0.42	1.0	6.0

Prime	Our Code		GMP Code			Speedup
	Sol.	Mont.	const time	var time	C++	
$2^{321}-9$	0.1	0.26	0.8	0.54	1.18	5.4
$2^{322}-2^{161}-1$	0.07	0.27	0.83	0.53	1.15	7.57
$2^{336}-17$	0.1	0.27	0.8	0.53	1.11	5.3
$2^{336}-3$	0.09	0.27	0.86	0.53	1.08	5.89
$2^{338}-15$	0.1	0.25	0.8	0.54	1.06	5.4
$2^{369}-25$	0.13	0.26	0.79	0.52	1.1	4.0
$2^{379}-19$	0.12	0.26	0.79	0.55	1.07	4.58
$2^{382}-105$	0.13	0.25	0.92	0.57	1.11	4.38
$2^{383}-187$	0.13	0.28	0.75	0.5	1.05	3.85
$2^{383}-31$	0.13	0.26	0.75	0.51	1.05	3.92
$2^{383}-421$	0.13	0.25	0.76	0.51	1.06	3.92
$2^{384}-2^{128}-2^{96}+2^{32}-1$	-	0.25	0.64	0.47	0.98	1.88
$2^{384}-317$	0.13	0.26	0.67	0.48	1.0	3.69
$2^{384}-5\cdot 2^{368}-1$	-	0.23	0.63	0.46	0.99	2.0
$2^{384}-79\cdot 2^{376}-1$	-	0.23	0.62	0.46	0.99	2.0
$2^{389}-21$	0.13	-	0.97	0.6	1.22	4.62
$2^{401}-31$	0.14	-	0.97	0.61	1.17	4.36
$2^{413}-21$	0.16	-	0.99	0.62	1.22	3.88
$2^{414}-17$	0.15	-	0.98	0.6	1.21	4.0
$2^{416}-2^{208}-1$	0.14	-	0.98	0.6	1.16	4.29
$2^{444}-17$	0.17	-	0.96	0.6	1.2	3.53
$2^{448}-2^{224}-1$	0.12	-	0.79	0.52	1.06	4.33
$2^{450}-2^{225}-1$	0.13	-	1.22	0.74	1.34	5.69
$2^{452}-3$	0.16	-	1.24	0.71	1.32	4.44
$2^{468}-17$	0.16	-	1.23	0.71	1.29	4.44
$2^{480}-2^{240}-1$	0.13	-	1.18	0.71	1.28	5.46

Prime	Our Code		GMP Code			Speedup
	Sol.	Mont.	const time	var time	C++	
$2^{488} - 17$	0.19	-	1.2	0.7	1.28	3.68
$2^{489} - 21$	0.2	-	1.17	0.69	1.27	3.45
$2^{495} - 31$	0.19	-	1.17	0.69	1.3	3.63
$2^{510} - 290 \cdot 2^{496} - 1$	-	-	1.2	0.7	1.28	-
$2^{511} - 187$	0.25	-	1.13	0.66	1.21	2.64
$2^{511} - 481$	0.25	-	1.12	0.66	1.24	2.64
$2^{512} - 491 \cdot 2^{496} - 1$	-	-	0.99	0.62	1.15	-
$2^{512} - 569$	0.24	-	0.95	0.62	1.14	2.58
$2^{521} - 1$	0.18	-	1.4	0.81	1.44	4.5

Table A.2: Full 32-bit benchmark data. Many of the 32-bit Montgomery implementations exceeded the one-hour timeout for proofs, because 32-bit code involves approximately twice as many operations. The C++ GMP program was not benchmarked on 32-bit.

	Our Code		**GMP Code**		
Prime	**Solinas**	**Mont.**	**const time**	**var time**	**Speedup**
$2^{127} - 1$	0.3	1.19	2.86	3.23	9.53
$2^{129} - 25$	0.35	1.7	3.38	3.77	9.66
$2^{130} - 5$	0.44	1.87	3.56	3.79	8.09
$2^{137} - 13$	0.48	2.06	3.41	3.78	7.1
$2^{140} - 27$	0.51	1.98	3.43	3.77	6.73
$2^{141} - 9$	0.51	2.0	3.43	3.81	6.73
$2^{150} - 3$	0.42	2.0	3.56	3.79	8.48
$2^{150} - 5$	0.49	1.99	3.38	3.8	6.9
$2^{152} - 17$	0.5	1.96	3.4	3.82	6.8
$2^{158} - 15$	0.52	2.04	3.4	3.77	6.54
$2^{165} - 25$	0.59	2.46	4.02	4.45	6.81
$2^{166} - 5$	0.61	2.43	4.02	4.43	6.59
$2^{171} - 19$	0.57	2.68	4.04	4.51	7.09
$2^{174} - 17$	0.58	2.63	4.03	4.39	6.95
$2^{174} - 3$	0.61	2.62	4.02	4.4	6.59
$2^{189} - 25$	0.7	2.65	4.05	4.4	5.79
$2^{190} - 11$	0.71	2.64	4.1	4.42	5.77
$2^{191} - 19$	0.66	2.69	4.03	4.4	6.11
$2^{192} - 2^{64} - 1$	-	2.41	3.56	4.23	1.48
$2^{194} - 33$	0.75	-	4.66	4.94	6.21
$2^{196} - 15$	0.77	-	4.64	4.94	6.03
$2^{198} - 17$	0.76	-	4.72	4.97	6.21
$2^{205} - 45 \cdot 2^{198} - 1$	-	-	4.66	5.03	-

Prime	Our Code		GMP Code		Speedup
	Solinas	**Mont.**	**const time**	**var time**	
$2^{206} - 5$	0.76	-	4.62	4.91	6.08
$2^{212} - 29$	0.86	-	4.68	4.91	5.44
$2^{213} - 3$	0.7	-	4.68	4.94	6.69
$2^{216} - 2^{108} - 1$	0.7	-	4.67	4.92	6.67
$2^{221} - 3$	0.8	-	4.68	4.92	5.85
$2^{222} - 117$	0.87	-	4.72	4.87	5.43
$2^{224} - 2^{96} + 1$	-	-	4.13	4.85	-
$2^{226} - 5$	0.87	-	5.25	5.65	6.03
$2^{230} - 27$	0.83	-	5.29	5.71	6.37
$2^{235} - 15$	0.9	-	5.31	5.69	5.9
$2^{243} - 9$	0.86	-	5.29	5.62	6.15
$2^{251} - 9$	1.12	-	5.3	5.65	4.73
$2^{254} - 127 \cdot 2^{240} - 1$	-	3.97	5.26	5.7	1.32
$2^{255} - 19$	1.01	-	5.25	5.7	5.2
$2^{255} - 765$	1.43	-	5.27	5.71	3.69
$2^{256} - 189$	1.2	-	4.71	5.49	3.93
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	-	-	4.7	5.46	-
$2^{256} - 2^{32} - 977$	1.65	-	4.72	5.45	2.86
$2^{256} - 4294968273$	-	-	4.77	5.48	-
$2^{256} - 88 \cdot 2^{240} - 1$	-	-	4.78	5.46	-
$2^{266} - 3$	1.01	-	6.1	6.32	6.04
$2^{285} - 9$	1.13	-	6.13	6.34	5.42
$2^{291} - 19$	1.33	-	6.94	6.98	5.22
$2^{321} - 9$	1.72	-	7.6	7.66	4.42
$2^{322} - 2^{161} - 1$	1.37	-	7.66	7.74	5.59
$2^{336} - 17$	1.67	-	7.64	7.74	4.57
$2^{336} - 3$	1.59	-	7.58	7.69	4.77
$2^{338} - 15$	1.7	-	7.66	7.67	4.51

Prime	Our Code		GMP Code		Speedup
	Solinas	**Mont.**	**const time**	**var time**	
$2^{369} - 25$	2.44	-	8.41	9.03	3.45
$2^{379} - 19$	2.47	-	8.44	9.25	3.42
$2^{382} - 105$	2.66	-	8.41	9.04	3.16
$2^{383} - 187$	2.63	-	8.44	9.11	3.21
$2^{383} - 31$	2.6	-	8.47	9.13	3.26
$2^{383} - 421$	3.58	-	8.45	9.11	2.36
$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	-	-	7.62	8.8	-
$2^{384} - 317$	3.95	-	7.62	8.82	1.93
$2^{384} - 5 \cdot 2^{368} - 1$	-	-	7.64	8.94	-
$2^{384} - 79 \cdot 2^{376} - 1$	-	-	7.66	8.84	-
$2^{389} - 21$	2.89	-	9.41	9.93	3.26
$2^{401} - 31$	2.85	-	9.35	9.92	3.28
$2^{413} - 21$	3.53	-	9.48	9.93	2.69
$2^{414} - 17$	3.72	-	9.4	9.86	2.53
$2^{416} - 2^{208} - 1$	2.48	-	8.54	9.67	3.44
$2^{444} - 17$	3.7	-	10.31	10.89	2.79
$2^{448} - 2^{224} - 1$	3.18	-	9.57	10.51	3.01
$2^{450} - 2^{225} - 1$	-	-	11.37	11.63	-
$2^{452} - 3$	3.23	-	11.33	11.63	3.51
$2^{468} - 17$	3.2	-	11.37	11.63	3.55
$2^{480} - 2^{240} - 1$	3.58	-	10.47	11.33	2.92
$2^{488} - 17$	7.99	-	12.23	12.92	1.53
$2^{489} - 21$	7.7	-	12.26	12.81	1.59
$2^{495} - 31$	6.07	-	12.2	13.1	2.01
$2^{510} - 290 \cdot 2^{496} - 1$	-	-	12.17	12.9	-
$2^{511} - 187$	9.73	-	12.21	13.07	1.25
$2^{511} - 481$	-	-	12.23	12.9	-
$2^{512} - 491 \cdot 2^{496} - 1$	-	-	11.26	12.58	-

Prime	Our Code		GMP Code		Speedup
	Solinas	**Mont.**	**const time**	**var time**	
$2^{512} - 569$	-	-	11.23	12.55	-
$2^{521} - 1$	3.9	-	13.3	13.91	3.41

Bibliography

[1] Fips 186-2. Technical Report 186-2, National Institute of Standards and Technology, 2000.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proc. CCS*, 2017.

[3] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Proc. Advances in Cryptology - CRYPTO 2011. 31st Annual Cryptology Conference.*

[4] 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*, pages 90–101. ACM, 2009.

[5] 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, 2006.

[6] Daniel J. Bernstein and Peter Schwabe, 2016.

[7] 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.

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

[9] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. *Constructing Semantic Models of Programs with the Software Analysis Workbench*, pages 56–72. Springer International Publishing, 2016.

[10] Andres Erbsen. Crafting certified elliptic curve cryptography implementations in coq. Master's thesis, Massachusetts Institute of Technology, 6 2017.

[11] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. 5:141–151, 06 2014.

[12] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. 2015.

[13] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[14] J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 2, pages 820–825 Vol.2, Oct 2003.

[15] Magnus O. Myreen and Gregorio Curello. A verified bignum implementation in x86-64 machine code. In *Proc. CPP*, 2013.

[16] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[17] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 53–72, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[18] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in f*. *Proc. ACM Program. Lang.*, 1(ICFP):17:1–17:29, August 2017.

[19] Jerome Solinas. Generalized mersenne numbers. Technical Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, 1999.

[20] Gerald J. Sussman and Guy L. Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Memo 349, MIT AI Lab, 1975.

[21] Laurent Théry and Benjamin Grégoire. Coqprime, 2016.

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

[23] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proc. CCS*, 2017.