Crafting Certified Elliptic Curve Cryptography Implementations in Coq

Andres Erbsen

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2017

Andres Erbsen
Department of Electrical Engineering and Computer Science
May 25, 2017

certified by

Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor
May 25, 2017

accepted by

Christopher Terman
Chairman, Masters of Engineering Thesis Committee
May 25, 2017

Abstract

Elliptic curve cryptography has become a de-facto standard for protecting the privacy and integrity of internet communications. To minimize the operational cost and enable near-universal adoption, increasingly sophisticated implementation techniques have been developed. While the complete specification of an elliptic curve cryptosystem (in terms of middle school mathematics) fits on the back of a napkin, the fast implementations span thousands of lines of low-level code and are only intelligible to a small group of experts. However, the complexity of the code makes it prone to bugs, which have rendered well-designed security systems completely ineffective.

I describe a principled approach for writing crypto code simultaneously with machine-checkable functional correctness proofs that compose into an end-to-end certificate tying highly optimized C code to the simplest specification used for verification so far. Despite using template-based synthesis for creating low-level code, this workflow offers good control over performance: I was able to match the fastest C implementation of X25519 to within 1% of arithmetic instructions per inner loop and 7% of overall execution time. While the development method itself relies heavily on a proof assistant such as Coq and most techniques are explained through code snippets, every Coq feature is introduced and motivated when it is first used to accommodate a non-Coq-savvy reader.

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

Contents

Chapter 1

Introduction

1.1 The Need for Ubiquitous Encryption

Every internet-connected application needs to deal with the reality that the network is operated by numerous companies and governments, most of which have no connection to the user or the application developers. This has led to the advice to build "under the assumption that the network is not only untrusted, but untrustworthy"[1]: every application developer has to decide (on behalf of the users) to either accept pervasive surveillance[2] and modification[3] of the network communications, or deploy cryptographic countermeasures.

1. Eben Moglen. *Why Political Liberty Depends on Software Freedom More Than Ever*. Feb. 5, 2011. URL: https://www.softwarefreedom.org/events/2011/fosdem/moglen-fosdem-keynote.html.

2. Brett Max Kaufman. *A Guide to What We Now Know About the NSA's Dragnet Searches of Your Communications*. Aug. 9, 2013. URL: https://www.aclu.org/blog/guide-what-we-now-know-about-nsas-dragnet-searches-your-communications.

3. Gabi Nakibly, Jaime Schcolnik, and Yossi Rubin. "Website-Targeted False Content Injection by Network Operators". In: *CoRR* abs/1602.07128 (2016). URL: http://arxiv.org/abs/1602.07128.

1.2 Challenges in Deploying Cryptography

The factors that influence the choice whether to use cryptography can be roughly separated into three categories: engineering costs, operational costs, and confidence in the effectiveness of a potential solution.

Engineering Costs

The difficulty of integrating cryptography depends heavily on the application, and detailed consideration of it is outside the scope of this work. Easy-to-deploy pre-made

solutions are available for common application types. For example, CertBot[4] can automatically configure the most commonly used open source web servers to encrypt all connections between the user and the server. An even more comprehensive privacy guarantee can be achieved in apps where users communicate directly to each other, using the plug-and-play implementations of the Signal Protocol[5]. Simultaneously, some protocols (such as email) are notoriously difficult to retrofit with any meaningful security measures.

4. https://certbot.eff.org/

5. https://github.com/WhisperSystems/libsignal-protocol-java (or -c, or -javascript)

Operational Costs

A common concern is that enabling encryption will slow the application down or cause excess server load. In the vast majority of applications, the operational cost of properly implemented cryptography is insignificant in comparison to that of the rest of the system[6]. However, this comfortable performance level is not an integral feature of cryptography but rather a result of decades of painstaking research and engineering. Crucially, an application developer does not need to understand the extremely sophisticated design and implementation techniques that make crypto fast – it is a very good idea to use a dedicated crypto library written by experts.

6. "On our production frontend machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10KB of memory per connection and less than 2% of network overhead. Many people believe that SSL takes a lot of CPU time and we hope the above numbers (public for the first time) will help to dispel that.

If you stop reading now you only need to remember one thing: SSL/TLS is not computationally expensive any more."

Adam Langley. *Overclocking SSL*. Google, June 29, 2010. URL: https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html.

Confidence in Cryptography

While important cryptographic algorithms and protocols are shared between use cases and have been widely vetted for security, the space of implementations is much more varied. For example, IANIX lists 250 libraries that provide ed25519 digital signature functionality[7]. Most of them have never received an external audit, and unlike when choosing a cryptographic protocol, no one of them is obviously more confidence-inspiring than the rest. Among highly optimized

7. *Things that use Ed25519*. Nov. 9, 2016. URL: https://ianix.com/pub/ed25519-deployment.html.

implementations, even those written by widely respected, experienced implementers have suffered bugs[8].

Indeed, reviewing crypto code is a difficult and unrewarding task. A function whose mathematical definition using appropriate abstractions would fit on the back of a napkin might be expanded into tens of thousands of uniform-looking lines of optimized code. There is also an issue of incentives: while finding an important bug can bring the investigator eternal glory and maybe even money from a bug bounty or a black-market deal, a "successful" audit is inevitably concluded with a non-glamorous "I didn't find any issues". Worse, if a bug is later found in a library that previously received a non-critical audit, it can negatively affect the auditor's reputation – even to the point of questioning their goodwill, if there is reason to believe that the original bug was maliciously inserted.

Yet aggressively optimized implementations are necessary for "obviously fast enough" speeds, and questionable confidence in the correctness of an implementation is a non-starter.

8. The TweetNaCl[9] paper describes a typo of unknown impact in the "original" fast implementation `ed25519-amd64-64-24k`: `r1 += 0 + carry` should have been `r2 += 0 + carry` instead. Authors noted that this line was one of 16,184 similar lines, and the issue would not have been caught by random tests.

9. Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "TweetNaCl: A crypto library in 100 tweets". In: *Progress in Cryptology – LATINCRYPT 2014*. Ed. by Diego Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2015, pp. 64–83. URL: `http://cryptojedi.org/papers/#tweetnacl`.

1.3 A Vision For Correct and Fast Crypto Implementations

The long-term goal of this work is to eliminate the tradeoff between performance and correctness by connecting specifications and implementations using computer-checkable proofs. Given precise models of the cryptographic primitives and the machines we run them on, it should not be necessary to manually audit the implementation – if the proof checks, the code is good to go. Instead of looking for mistakes and hoping that we don't find any, we look for correctness proofs and routinely release them with the code. As more general theorems tend to be easier to prove, we mechanize generalized versions of common implementation strategies, allowing us to generate

decently performant correct-by-construction implementations for new primitives by just plugging in a couple of parameters.

This thesis describes my experience with the above-described development model across several abstraction layers: from specialized arbitrary-precision arithmetic implementations to proofs of high-level optimizations for elliptic curve cryptosystems. The “top” specifications used here are higher-level than in any previous verification work; the pipeline bottoms out to a minimal subset of the C language. Because no high-end CPU manufacturer has released an authoritative model of the timing properties of their processors, we use the standard informal process for checking that our code is constant time. The chapters are self-contained in terms of core technical content, but every Coq feature is explained only the first time it is used.

1.4 Academic Context and Individual Contributions

The work described in this thesis is a result of active collaboration between myself, Jason Gross, and Jade Philipoom, all advised by professor Adam Chlipala. Earlier, Robert Sloan built a prototype of the low-level compilation pipeline which ended up being replaced as the requirements matured.

I designed the overall architecture of the project, specified the components, and built early prototypes of their implementations. I wrote the first (inelegant and limited) sketch of finite field implementation (and proof) and defined the specification for the complete version, at which point Jade took over the work of designing, implementing, and proving the correctness of the full finite field implementation synthesis library. Similarly, I built a proof-of-concept implementation and proof sketch of the

range analysis pass and the word size selection compiler, based on which Jason implemented (and proved correct) a similar design that could handle the full set of operations used in our code. On the other hand, while I did most of the coding for the tactic automation for proving elliptic curve formulas, Jason pointed me to the `nsatz` tactic and first proposed the key idea behind extending it to handle divisions. While the formalization of properties of elliptic curves is exclusively my work, it was significantly helped by information gained from early experiments in the same direction performed by Jason. All in all, we think of this project as truly joint work.

Chapter 2

Existing Verification Work

There have been several efforts to mechanically check that a program precisely implements a primitive cryptographic operation. The problem of verifying functional correctness of programs has also been studied generically, and in various other application domains. This section will provide a brief overview of the landscape of approaches, explain the ones that would be applicable here, and finally justify the high-level choices made in this work. In particular, I seek to illustrate the reasons why I use Coq (and write my own automation) instead of relying on highly automated but less flexible tools predominant in previous work.

To understand the different sources of complexity in verifying cryptographic implementations, let's first consider a rather degenerate case. The salsa20 stream cipher is specified in terms of simple operations on 32-bit integers[1]. The original English-language specificationcan be easily transcribed to many general-purpose programming languages. I find it rather meaningless to talk about formally verifying functional correctness of such a translation: as the original specification is not machine-readable, there is no way to tell the difference between *interpreting* the specification (e.g., assigning meaning to the ellipsis in little-endian conversion described in section 8) and *violating* it (perhaps by only applying little-endian

1. Daniel J. Bernstein. *Salsa20 specification*. Apr. 27, 2005. URL: https://cr.yp.to/snuffle/spec.pdf

conversion to the first and last word of the input).

2.1 SMT Implementation Equivalence Checking, SAW

This does not mean that formal methods cannot be used to gain confidence in a straightforward implementation. For example, it would still be useful to verify that two independent translations of the specification to machine-readable languages coincide. In doing so, the original English specification would be irrelevant; the verification work would come down to the differences between the language and coding style used in the two implementations. If the implementations are similar enough, the equivalence can be checked completely automatically using an SMT solver (e.g., CVC4, Z3). Here "similar enough" roughly means that the differences between the implementations, as written in the source code, must either be within the bounds of the theories understood by the SMT solver (e.g., linear arithmetic, arrays, bit vectors, but not control flow or number theory) or amenable to brute-force exploration. The exercise of verifying the equivalence of two translations of the salsa20 spec has indeed been performed as an example to demonstrate the use of the Software Analysis Workbench[2] of Galois, Inc, along with AES, DES, and others [3] .

The extent to which these success stories of SMT-based one-shot verification can be generalized is a lively topic of debate. I think it is fair to extrapolate that this approach would work for traditional symmetric cryptography implementations where the most optimized code performs roughly the same intermediate computations as those described in the specification, just in a different manner. However, verification gets much more challenging (and interesting) as the specification and the implementation grow further apart.

2. Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. "Constructing Semantic Models of Programs with the Software Analysis Workbench". In: *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers*. Ed. by Sandrine Blazy and Marsha Chechik. Springer International Publishing, 2016, pp. 56–72. ISBN: 978-3-319-48869-1. DOI: 10.1007/978-3-319-48869-1_5. URL: https://saw.galois.com/files/saw-vstte-final.pdf.

3. https://github.com/GaloisInc/saw-script/tree/master/examples

```
// P384 field reference modulus.
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 2.1: Cryptol specification of reduction modulo p384 by Galois, Inc. $[n]$ refers to the type of n-bit words, `sbb` stands for subtract-with-borrow, `drop` and `chop` cast between word sizes, # concatenates words with the most significant end on the left. If it is not obvious to you how this code implements reduction modulo $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, it may be the case that you didn't write it. No need to worry – we will return to simple specifications shortly.

Source: https://github.com/GaloisInc/saw-script/blob/master/examples/ecdsa/cryptol-spec/p384_field.cry#L50

Cryptol and SAW

The above-mentioned library of Cryptol specifications and SAW scripts does contain a verification of equivalence between a Java implementation of ECDSA-p384 and a Cryptol implementation of ECDSA-p384, but both of them are rather low-level when compared to the treatment of ECDSA in RFCs or academic papers . The authors state that the Cryptol implementation "is intended to closely resemble the English

specification of the algorithm" which already includes several non-trivial optimizations. For example, modular reduction is defined as the optimized multi-tap reduction formula specialized to the p384 prime (figure 2.1) instead of referring to the standard definition of modular arithmetic built into Cryptol. Anecdotally, I find the Cryptol reference much more difficult to read than the Java implementation – it is longer, has more complicated data flow, deviates from standard implementation strategies, and nevertheless contains the tricky optimizations that make writing ECC code hard. The ease-of-use argument for SMT-based verification starts to break down as well: the SAW script driving the equivalence check is roughly similar in size to the implementation itself.

2.2 Annotations, Invariants, Proofs

The unexciting assurance-effort trade-off of the verification of ECDSA described above is perhaps best understood as a mismatch between the tool and the task. While optimizations in traditional symmetric-key cryptography focus on reshuffling computations, implementations of most asymmetric cryptography rely on sophisticated algorithmic improvements. In the second scenario, the source code of the reference and optimized implementations

is not sufficient for understanding why they compute the same value. While an experienced programmer can recognize an algorithm and recall its correctness argument, it would be inadvisable to try to learn to write fast crypto code by only reading the optimized assembly-language implementations. Asking an SMT solver to do just that is inherently an uphill battle. See figure 2.2 for an example from the domain of numerical computation for computer graphics: both the C function itself and the correctness reference are relatively simple, but non-trivial domain knowledge is required to see the relationship between them.

```
float
approx_inv_sqrt(float x) {
  int m = 0x5f3759df;
  int i=m-((*(int*)&x)>>1);
  return *(float*)&i;
}
```

Figure 2.2: This code snippet computes an approximation of $1/\sqrt{x}$ accurate to within 4%. It is now feasible to verify this 32-bit version by brute force but the general reasoning behind its correctness (and the only argument for the correctness of the wider variants) requires several pages. Chris Lomont. *Fast Inverse Square Root*. 2003. URL: https://www.lomont.org/Math/Papers/2003/InvSqrt.pdf. URL: https://www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

To support checking the correctness of tricky optimizations, verification tools need to provide means for describing (and verifying) the relationship between two algorithms. Enlisting the programmer's assistance in guiding the correctness proof replaces the "brick wall" between what can and cannot be verified with a complicated trade-off between human time spent on writing object code, writing annotations, improving the verification software, and waiting for the verification to run. Iterating on these points simultaneously can also increase the risk of introducing soundness bugs in the verification software. Depending on the architecture of the verification tools, there may be no easy way to tell whether the change that made the verifier accept the object code as correct simply improved its reasoning ability or allowed some class of programs to pass regardless of whether they are actually correct. Nevertheless, iterating on the code and tools is incredibly valuable for productivity: well-structured code is easier to verify, and tool fit is extremely important for productivity.

The three remaining efforts at verifying elliptic curve cryptography implementations will serve as examples of different approaches within this design space.

2.3 X25519-amd64 verified using Boolector and Coq

The bodies of the main loops of the original fast implementations of Curve25519 written in qhasm for AMD64 processors were verified using a combination of the Boolector SMT solver and the Coq proof assistant[4]. The code was annotated with detailed invariants specifying the relationships between register values after every couple of instructions. The qhasm code was then machine-translated into Boolector formulas using a mostly straightforward 2000-line tool written specifically for this use case. The SMT solver was used to relate the qhasm code to a < 100-line

4. 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*. ACM, 2014, pp. 299–309. URL: http://cryptojedi.org/papers/#verify25519.

Figure 2.3: One out of 20 code-annotation blocks in the modular multiplication routine of the verified x25519-amd64-24k. Lines starting with #// are annotations; the code (in qhasm) has semantics similar to assembly but allows named variables (register-allocated without spilling) and C-style syntax for single assembly instructions.

```
x3 = *(uint64 *)(xp + 24)
rax = *(uint64 *)(yp + 0)
(uint128) rdx rax = rax * x3
carry? r3 += rax
c = 0
c += rdx + carry

#// var D0 = A2 + (xp[24]@u128 * yp[0]@u128)@u512 * 2**192
#// cut r7 = 0 && x3 = xp[24] &&
#//        A0 = (xp[0]@u128 * yp[0]@u128)@u512 + (xp[0]@u128 *
    yp[8]@u128)@u512 * 2**64 +
#//                (xp[0]@u128 * yp[16]@u128)@u512 * 2**128 +
    (xp[0]@u128 * yp[24]@u128)@u512 * 2**192 &&
#//         A1 = A0 +
#//                (xp[8]@u128 * yp[0]@u128)@u512 * 2**64 +
    (xp[8]@u128 * yp[8]@u128)@u512 * 2**128 +
#//                (xp[8]@u128 * yp[16]@u128)@u512 * 2**192 +
    (xp[8]@u128 * yp[24]@u128)@u512 * 2**256 &&
#//         A2 = A1 +
#//                (xp[16]@u128 * yp[0]@u128)@u512 * 2**128 +
    (xp[16]@u128 * yp[8]@u128)@u512 * 2**192 +
#//                (xp[16]@u128 * yp[16]@u128)@u512 * 2**256+
    (xp[16]@u128 * yp[24]@u128)@u512 * 2**320 &&
#//        D0 = A2 + (xp[24]@u128 * yp[0]@u128)@u512*2**192 &&
#//        r0@u512 + r1@u512 * 2**64 + r2@u512 * 2**128 +
    r3@u512 * 2**192 + (r4@u65 + c@u65)@u512 * 2**256 +
    r5@u512 * 2**320 + r6@u512 * 2**384 = D0
```

program that uses arbitrary-precision integer arithmetic. This program was then hand-transcribed to Coq and proven to compute the correct value modulo $2^{255} - 19$.

The verify25519 project is remarkable in that it verified an existing, highly optimized implementation without modifying the code at all, producing a verified implementation that is likely to remain the fastest on the machines that it was designed for. However, there are several reasons to be less than enthusiastic when it comes to applying the same strategy to other optimized implementations. First, even if taking the qhasm-to-SMT translation as a given, a significant amount of per-implementation effort must be required for writing the rather detailed annotations (see figure 7.1 for an example) – the authors express interest in seeing this process automated. Second, the use of ad-hoc tools for translating code to SMT formulas introduces these tools to the trusted

base – in this case, increasing the application-specific TCB (excluding Boolector and `coqchk`) from perhaps 20 lines to around 2000 lines. A similar concern can be raised about hand-translating code from Boolector to Coq, even though the code fragment for which this was necessary is indeed small.

The most serious concern I have with this SMT-heavy methodology is predictability. It is exceedingly important to be able to estimate, given a specification and an implementation, whether a strategy for verifying their correspondence would be fruitful and how the effort required would compare to previous similar projects. Given the data points from the verify25519 paper and codebase, I would not even venture to extrapolate as little as to guess whether one could verify X448 in this style. As described in section 6, several ad-hoc (but TCB-increasing!) heuristics were necessary to make different parts of the X25519 verification complete in tolerable time. Furthermore, the number of annotations required for verifying modular multiplication increased 27-fold when the modulus length was roughly doubled from $2^{127} - 1$ to $2^{255} - 19$. Furthermore, the authors report that Boolector was unable to verify the modular congruence relations in the representation that used 64-bit limbs, but worked just fine on the representation that used fewer bits than the size of the machine word (51). As far as I understand, the root cause for this limitation has not been explained beyond “Boolector is not good at non-linear modular arithmetic”.

2.4 hacl-star

A significant improvement in the ease and predictability of verification can be achieved by borrowing ideas from programming itself. Standard abstraction and modularity techniques have counterparts in verification land, and if the structure of the code coincides with the structure of its

correctness argument, the two can be interleaved. In the fairytale world of inexplicably easy verification, code could be verified by formatting its doc-comments into machine-readable specifications. However, performance requirements in particular are notorious for driving programmers to give up modularity. An elliptic curve cryptography library might contain an internal "field element addition" function that only adds correctly if each digit in the internal representation of the input field elements is "small enough" – a requirement which each caller carefully maintains. The contract of this addition function can no longer be explained in terms of field elements: details of the representation must leak through to allow for fast implementations.

The work of Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan on implementing elliptic curve cryptography in the F* language[5] can be seen as an exploration of this axis of the design space. The initial release of the library (described in the paper) implemented most big-integer arithmetic in generic code that were called from implementations of specific finite fields. However, the more tricky optimizations such as the modular reduction in the style of figure 2.1 were implemented and verified separately for each prime. Even though the reuse of arithmetic operation code between finite fields is important, perhaps the biggest reduction in effort was due to reuse of code (and verification) of the innermost loop of multiplication (figure 2.4), which was unrolled in qhasm code discussed earlier, requiring each iteration to be verified separately. Quantitatively, the generic verification of bignum operations was completed in several thousand lines of proof and code, similar to the verification of the X25519-specific inlined qhasm code.

Unfortunately, creating elliptic curve implementations by naively composing generic bignum operations makes it difficult to achieve good performance (the first version of

5. Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. "A Verified Extensible Library of Elliptic Curves". In: *IEEE Computer Security Foundations Symposium (CSF)*. 2016.

```
val scalar_multiplication_tr_1:
res:bigint_wide -> a:bigint{Similar res a} -> s:limb -> \
                        ctr:nat{ctr<norm_length} ->
ST unit
 (requires (fun h ->
   (live h res) /\ (live h a) /\ (getLength h a >= norm_length)
                /\ (getLength h res >= norm_length)
   /\ (forall (i:nat). (i >= ctr /\ i < norm_length) ==>
              v (getValue h a i) * v s < pow2 platform_wide)))
 (ensures (fun h0 u h1 ->
   (live h0 res) /\ (live h1 res) /\ (live h0 a) /\ (live h1 a)
   /\ (getLength h0 res >= norm_length)
   /\ (getLength h1 res = getLength h0 res)
   /\ (modifies !{getRef res} h0 h1)
   /\ (getLength h0 a >= norm_length)
   /\ (forall (i:nat). (i >= ctr+1 /\ i < norm_length) ==>
                v (getValue h0 a i) * v s < pow2 platform_wide)
   /\ (forall (i:nat). (i < getLength h1 res /\ i <> ctr) ==>
                      (getValue h1 res i == getValue h0 res i))
   /\ (v (getValue h1 res ctr) = v (getValue h0 a ctr) * v s)
 ))
let rec scalar_multiplication_tr_1 res a s ctr =
let ai = index_limb a ctr in
let z = mul_limb_to_wide ai s in
upd_wide res ctr z
```

Figure 2.4: HACL-star bignum multiplication, body of the innermost loop. The language is F*, an ML-like impure functional language with SMT-based verification. The code is written in a tail-recursive style but using mutable memory. The code between `val` and `let` is the specification. `mul_limb_wide` refers to a function that multiplies two n-bit numbers to produce a $2n$-bit number. The SMT-based verifier is able to check that this code conforms to its specification using only the information presented here. Later, additional lemmas are used to connect this with the rest of the bignum multiplication code, arriving at the one-line specification of the full multiplication routine about a thousand lines later.

hacl-star described in the original paper was 290 times slower than a reference implementation written in C). The version of hacl-star currently available online has resorted to a style much closer to that of verify25519: the codebase has been split into mostly independent implementations of each finite field, which are verified separately.

While neither verify25519 nor hacl-star establish a formal connection between their elliptic curve implementations and a short mathematical specification of elliptic curves, the top-level specification of X25519 comes close to the way the same primitive is described in RFC 7748. Although particular implementations of some algebraic operations like finding the inverse of a finite field element are still inlined in the specification, I think it is likely that a separate verification effort could connect the implementation to a higher-level specification by reasoning about its specification alone. In particular, unlike verify25519, hacl-star's X25519 specification

implies that not only is the output correct modulo $2^{255} - 19$, but also that its specific representation is uniquely defined by its value modulo $2^{255} - 19$.

2.5 gfverif

The last project, `gfverif`,[6] is completely different from the last two in that it makes no attempt at achieving complete or general-purpose verification. Instead, the tool is restricted to analyzing C code with statically known control flow and can only deduce modular arithmetic properties about it. Due to the very specific domain, the verification software can contain elaborate domain-specific algorithms for checking these properties. While `gfverif` can verify code with loops by unrolling the loops, this description will focus on the case of straight-line code.

6. Daniel J. Bernstein and Peter Schwabe. *gfverif*. Jan. 1, 2016. URL: `http://gfverif.cryptojedi.org/`.

The key idea that makes `gfverif` work is decomposing the problem of verifying field arithmetic code into three parts that we already know how to solve:

1. Some linear combination of some run-time machine words represents one field element, and a fairly limited set of operations is performed on these machine words.
2. The machine words do not overflow.
3. If the machine words are treated as mathematical integers, the output of the optimized implementation is congruent to the output of the reference implementation.

The first requirement is easy to check by reading the code once.

The second is the subject of a well-known compiler pass, range analysis. To compute the ranges of all variables, annotate the input words with the full range and proceed step-by-step as the execution of the program would,

computing the range of the output of each operation based on the ranges of its inputs. It is considered obvious that if a program operating on machine words completes without ever exceeding the maximum range of a machine word, it produces the same output as it would if operating on mathematical integers.

The third verification step is more difficult than the first two, but fortunately common computer algebra systems already implement an algorithm that can be used for this purpose. In particular, `gfverif` reuses the decision procedure for inclusion in ideals of polynomial rings: when working in the ring $\mathbf{R}$ of polynomials with one variable for each input of the program, the outputs of the optimized and reference implementations are equivalent modulo m if their difference belongs to $\mathbf{R}/m$.

The result is very impressive, as long as it is applicable: only a couple of lines of annotations are necessary to verify the correctness of the finite field arithmetic of an elliptic curve implementation. As an additional bonus, the range analysis can also be used to check that the final output represents an integer smaller than $2^{255} - 19$, which for many integer representations implies that the value is represented canonically. However, the applicability of this simple strategy is limited: for example, while `gfverif` easily verifies the implementation of X25519 that uses 5 machine 64-bit words to represent a 255-bit integer (51 bits each), the version that ues 4 64-bit integers (64 bits each) does not seem to fit the requirements imposed by `gfverif`. While the current implementation of `gfverif` is an ad-hoc combination of separate tools (with ample "XXX" comments throughout its source code), the strategy is sound, and could in principle be implemented in a system that allows composing the `gfverif`-style verification of some finite field arithmetic with a higher-level proof that connects that formula to a simple mathematical definition of an elliptic curve.

Chapter 3

Finite Field Arithmetic

This chapter will start from a review of good practices in reasoning about arithmetic in Coq, and build up to a minimal library for synthesizing highly optimized finite-field arithmetic formulas. Detailed discussion of the full scope of field arithmetic synthesis used in this project is left for another document, but the version shown here is intended to be sufficient to form an intuitive understanding sufficient for reading the rest of this thesis.

3.1 Introduction to Verifying Arithmetic in Coq

Using Coq to verify code is often described as labor-intensive and tedious when compared to SMT-based tools. The real issues are more subtle than that. While the Coq standard library contains very convenient mechanisms for proving correctness of arithmetic, even the very same library contains numerous examples of tedious and fragile arithmetic proofs. This is an example of the more general phenomenon that Coq does not prescribe any particular implementation strategy or coding style. This is very different from languages like Go, where the standard toolchain enforces a specific set of practices (e.g., dead code is considered an error). Different groups use very different

conventions in their Coq developments (for both code and proofs) to the extent that it is not obvious from the source code that the programs are in the same language. Therefore, it is rather meaningless to criticize Coq for the existence of long and ugly proofs of simple programs.

I do not mean to say that Coq is just fine, and it is the user's problem that they can't write short and simple proofs. It is perfectly appropriate to express disappointment that the ambient knowledge of which strategies work and which don't has not been aggregated into a newcomer-friendly format. A curated repository of exemplary developments following consistent rules and conventions would be very valuable, as would high-level analysis of various recurring design choices. However, when working outside the realm of tried and true conventions, it is the developer's responsibility to notice when the code is going down a rabbit hole and the proof becoming a long transcript of mechanically generated steps rather than a minimal computer-readable encoding of the conceptual correctness argument.

Square of a Sum, Hard Way and Easy Way

As an example of how a simple verification task can turn into a morass, let's consider a straightforward attempt at proving the middle school formula for the square of a sum: $(a + b)^2 = a^2 + 2ab + b^2$. I suggest reading a couple of steps of the proof in figure 3.1 on the next page, skipping to the longest block near the bottom when boredom starts to kick in. The entire proof is reproduced here just to visualize its size and to show that it is *feasible*, I am not recommending following its example.

What went wrong in the page-long proof in figure 3.1? Every `rewrite` step seems necessary, almost obvious. The `SearchAbout` commands could have been avoided if I had remembered the naming convention for arithmetic lemmas, but effort spent on remembered lemma names is effort not

```
Lemma square_of_sum_the_hard_way (a:Z) (b:Z)
: (a+b)^2 = a^2 + 2*a*b + b^2.
Proof.
  SearchAbout (_^2 = _).
  (* Output: Z.pow_2_r: forall a : Z, a^2 = a*a *)
  rewrite Z.pow_2_r.

  (* Goal: (a + b)*(a + b) = a^2 + 2*a*b + b^2 *)
  SearchAbout ((_ + _)*_ = _).
  (* OMEGA16:forall v c l k : Z, (v*c+l)*k=v*(c*k)+l*k *)
  (* Z.mul_add_distr_r:forall n m p:Z, (n+m)*p=n*p+m*p *)
  rewrite Z.mul_add_distr_r.

  (* Goal: a*(a + b) + b*(a + b) = a^2 + 2*a*b + b^2 *)
  rewrite Z.mul_add_distr_l.
  (* Goal: a*a + a*b + b*(a + b) = a^2 + 2*a*b + b^2 *)
  rewrite Z.mul_add_distr_l.

  (* Goal: a*a + a*b + (b*a + b*b) = a^2 + 2*a*b + b^2 *)
  rewrite <-Z.pow_2_r.
  rewrite <-Z.pow_2_r.

  (* Goal: a^2 + a*b + (b*a + b^2) = a^2 + 2*a*b + b^2 *)
  SearchAbout (_ + (_ + _) = _).
  (* Z.add_assoc: forall n m p:Z, n + (m+p) = n+m + p *)
  rewrite Z.add_assoc.

  (* Goal: a^2 + a*b + b*a + b^2 = a^2 + 2*a*b + b^2 *)
  SearchAbout (?x*?y = ?y*?x).
  (* Z.mul_comm: forall n m : Z, n*m = m*n *)
  rewrite Z.mul_comm.
  (* Goal:  a^2 + b*a + b*a + b^2 = a^2 + 2*a*b + b^2 *)
  (* No, wanted to turn [b*a] into [a*b]! *)
  Undo 1.
  rewrite (Z.mul_comm b a).

  (* Goal: a^2 + a*b + a*b + b^2 = a^2 + 2*a*b + b^2 *)
  SearchAbout (?x + ?x = _).
  (* Zplus_diag_eq_mult_2: forall n : Z, n + n = n*2 *)
  (* Zred_factor1: forall n : Z, n + n = n*2 *)
  (* Z.add_diag: forall n : Z, n + n = 2*n *)
  Fail rewrite Z.add_diag.
  (* ...Found no subterm matching "?M1305 + ?M1305"... *)
  (* goal is read as [((a^2 + a*b) + a*b) + b^2 = ...] *)
  rewrite <-(Z.add_assoc (a^2) (a*b)).

  (* Goal: a^2 + (a*b + a*b) + b^2 = a^2 + 2*a*b + b^2*)
  rewrite Z.add_diag.
  (* Goal:  a^2 + 2*(a*b) + b^2 = a^2 + 2*a*b + b^2 *)
  (* Almost there... *)
  rewrite Z.mul_assoc.

  (* Goal: a^2 + 2*a*b + b^2 = a^2 + 2*a*b + b^2 *)
  trivial.
Qed.
```

Figure 3.1: A direct, low-level proof of a middle school algebra formula. This figure is a complete `.v` file. I included selected pieces of feedback from the interactive line-by-line interface as comments (enclosed between `(*` and `*)`). Even though it is *possible* to write valid Coq proofs in a plain text editor, immediate step-by-step feedback is essential to productivity, especially if the proof strategy is lacking as in this example.

`Require Import ZArith` summons the standard library definition of (unbounded, mathematical) integers; `Local Open Scope Z_scope` makes operators `+`, `*`, and `-` refer to that definition. The `Lemma` line can be read as "given integers a and b, return (a proof of) the equality $(a+b)^2 = a^2 + 2ab + b^2$"

The `SearchAbout` command queries the `Required` libraries for definitions (including proofs) that contain the given pattern as a subterm; an underscore in a pattern matches everything. `rewrite lem` and `rewrite <-lem` locate the left-hand side (resp. right-hand side) of the equivalence proof `lem` in the goal and replace it with the other side, automatically filling in arguments of `lem` if they are obvious.

`rewrite`, like all other tactics used in the interactive proof mode, is not baked into the logic of Coq. Every invocation simply generates a proof that the new goal implies the previous one under the hood; these proofs are checked at `Qed`.

spent on doing more useful proofs. At a slightly higher level, one could argue that maybe I should have tried to reduce both sides of the equation instead of manipulating the left-hand side to match the right-hand side. While this hints at the solution I am going to present at the end of this discussion, it doesn't go far enough: while simplifying both sides would have made this proof more systematic, it would not have made it significantly shorter, and would not have eliminated the need to look up (or know) the used lemmas.

A much less sophisticated heuristic is useful at diagnosing the issue that made this proof blow up to an entire page: what do we see when we look at the code? We see basic algebraic properties – distributivity, commutativity, associativity – applied one at a time. In some sense, they are necessary: if we were working with some algebraic structure other than the integers where these laws did not hold, this lemma might not be true. However, the entire reason we care about associativity and commutativity is that *reduce* the amount of detail we need to care about: associativity lets us ignore how a sequence of applications of the same operation is parenthesized, and commutativity lets us ignore the order of those operations. While this proof invoked associativity and commutativity, it made no use of the key insights that make these properties valuable.

Indeed, when writing the proof in figure 3.1 I manually executed a general technique for using associativity, commutativity and distributivity to simplify equations. I claim that manually executing a well-defined procedure to write Coq proofs should be avoided by default, even though it is perhaps acceptable in cases where it is possible to estimate with high confidence that the list of all manual steps that will *ever* be performed is shorter than an implementation of the procedure. Figure 3.2 presents a much simpler encoding of essentially the same proof: it calls a standard-library tactic that implements the said procedure. Unfortunately, the implementation is optimized

```
Lemma sum_of_square_ring_simplify (a:Z) (b:Z)
: (a+b)^2 = a^2 + 2*a*b + b^2.
Proof.
  ring_simplify.
  (* Goal:  a^2 + 2*a*b + b^2 = a^2 + 2*a*b + b^2 *)
  trivial.
Qed.
```

Figure 3.2: A proof of the same lemma as in figure 3.1, this time using standard library proof automation.

for speed at the cost of readability, so reproducing it would not be instructive.

Similarly to `rewrite`, `ring_simplify` is not a part of the verification side of Coq. It just generates proofs, which end up being conceptually quite similar to the one in figure 3.1. This means that a bug in `ring_simplify` could cause a valid proof to be rejected, but cannot allow a false claim to appear true. While it is debatable whether the Coq proof checker implementation is simpler than, say, an SMT solver, it is a huge advantage of Coq that arbitrarily sophisticated automation can be added without concern about its correctness: even if a bad proof is generated, it is rejected by the checker.

3.2 Introduction to Coq "Synthesis Mode"

There is no such thing as "Synthesis Mode" implemented in Coq. However, it is very useful to conceptually separate the technique described in this section from the use of Coq to write code or proofs, the same way as distinguishing between code and proofs is very useful for maintaining sanity in a Coq development[1]. We will start with a minimal but rather contrived example task: to synthesize some code for computing $(a + b)^2$. It won't even be fast or complicated code; in fact in most circumstances it will be much slower than simply computing $(a + b)^2$, and it would be easier to write down this synthesized formula directly rather than deriving it. Nevertheless, I hope this example will be instructive: coming up with a technique for writing fast code is often orthogonal to implementing it in proof-producing

1. In principle, the interactive interface mode of Coq can be used to write code, and the static, compiler-like interface can be used to write proofs. However, doing so in an unprincipled manner is universally considered unwise to the extent that the interactive mode is commonly referred to as "proof mode".

Figure 3.3: Deriving the standard formula for $(a+b)^2$.

The lemma statement can be read "given integers a and b, return a c and a proof that $c = (a+b)^2$", which is the conventional way of encoding "c such that $c = (a+b)^2$" in Coq.

The `eexists` defers specifying the value of c, creating a goal with an *evar* on the left-hand side of the equality. While most proof objects created using the interactive mode are very difficult to read for humans, this one is quite informative. Using `eexists` in the very beginning of the derivation forces the value of c to appear in the beginning of the proof, even though the expression for it was determined later on.

We make a separate definition for c by using compile-time evaluation to inline `sum_of_square_synthesis` and extract the appropriate field.

```
Lemma sum_of_square_synthesis (a b:Z) : { c | c=(a+b)^2}.
Proof.
  (* Goal: { c | c = (a+b)^2 } *)
  eexists ?[c].
  (* Goal:  ?c = (a + b)^2 *)

  set (please_dont_touch_c := ?c).

  (* Goal:  please_dont_touch_c = (a + b)^2  *)
  ring_simplify.
  (* Goal:  please_dont_touch_c = a^2 + 2*a*b + b^2 *)

  subst please_dont_touch_c.

  (* Goal:  ?c = a^2 + 2*a*b + b^2 *)
  trivial. (* the value of ?[c] is fixed here *)
Defined. (* Not [Qed] - we don't want [c] to be private*)

Print sum_of_square_synthesis.
(* fun a b : Z =>                                        *)
(*   exist                                               *)
(*     (fun c:Z => c = (a+b)^2) - invariant in the type*)
(*     (a^2 + 2*a*b + b^2)      - synthesized [c]      *)
(*     ( ... )  - unreadable proof from [ring_simplify]*)

Definition sum_of_square_formula (a:Z) (b:Z) :=
 Eval cbv [sum_of_square_synthesis] in (
 let '(exist _ c _) := sum_of_square_synthesis a b in c).

Print sum_of_square_formula.
(* fun (a:Z) (b:Z) => a^2 + 2*a*b + b^2 *)

Lemma sum_of_square_formula_correct (a:Z) (b:Z)
: sum_of_square_formula a b = (a+b)^2.
Proof.
  apply (proj2_sig (sum_of_square_synthesis a b))
Qed.
```

synthesis-mode code, this example will illustrate the latter.

Please read figure 3.3, paying extra attention to the lines with comments next to them. There is a fair amount of boilerplate because no part of Coq was designed for doing this. Nevertheless, we are able to reuse the same `ring_simplify` script that we used in the previous proofs to do synthesis. This is not quite an accident: it is possible to theorize about how constructive proofs of existence of some program correspond to recipes for synthesizing that program, but there is a much more important property of

this approach that I would like to highlight: it works. The overhead caused by the boilerplate for separating out derived code and its correctness proof quickly becomes negligible when compared to the derivation itself; standard programming modularity techniques apply to derivation rules. Delaware, Pit–Claudel, Gross, and Chlipala describe how to use (an extension of) this style to build a verified ahead-of-time query planner for a SQL-like language[2]. Pit-Claudel extends the same framework to cover the "synthesis problem" of optimizing compilation in his thesis[3]. The remaining sections in this chapter will describe how this technique is used to synthesize fast implementations of low-level arithmetic.

2. Benjamin Delaware, Clément Pit–Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proc. POPL*. 2015. URL: http://plv.csail.mit.edu/fiat/papers/fiat-popl2015.pdf.

3. Clément Pit-Claudel. "Compilation using Correct-by-Construction Program Synthesis". Master's thesis. Sept. 2016. URL: http://pit-claudel.fr/clement/MSc/FiatToFacade_Pit-Claudel_2016.pdf.

3.3 Synthesizing Field Arithmetic

While many applications can be implemented exclusively by manipulating numbers using the bounded integer arithmetic provided by the CPU, cryptographic computations rely extensively on computing with numbers too large to fit inside registers. Even standard general purpose arbitrary-precision arithmetic libraries are rarely applicable because they do not run in constant time (which is necessary to avoid leaking secrets through timing). Instead, specialized arithmetic libraries are implemented separately for each elliptic curve and finite field. This does not mean that each implementation is a unique snowflake: a handful of algorithms cover the vast majority of concrete implementations. Nevertheless, performance gains of several orders of magnitude are achieved by writing out the code for specific parameters: wishing that a compiler specialized a high-level algorithm has been simply too much to ask. The remainder of this chapter will build up towards a verified template that when specialized to specific parameters produces modular reduction code similar to that written by experts. The plan is as follows:

1. Define a representation for multi-word integers, prioritizing mathematical simplicity and convenience of proof over execution performance.
2. Write code and proofs for the most general algorithms.
3. Specify concrete parameters and simplify to get straight-line code akin to what an expert write by hand.
4. Assign appropriately-sized finite-width registers to each variable in the synthesized code (next chapter).

I would like to stress that the primary motivation behind the choice of this strategy is to simplify the proofs. Being able to reuse the code and the proofs for different implementations is a nice side benefit, but we decided on this architecture when we were planning to produce an implementation of arithmetic modulo a single prime to be run on a single architecture. It is difficult to distill the gist of the experience that guided us towards this choice, but since this was perhaps the most important factor that shaped this field arithmetic library, I will offer some fuzzy extrapolations in hope that they will be helpful:

- More general statements tend to be easier to prove. Quantifying over something means its details cannot pop up to clutter the proof view and lead the effort astray.
- Thinking in terms of one model of the problem domain and writing code and proofs in another is very cumbersome: you end up manually translating every proof step from the correct framework to the one that is in the code, often with little help from proof automation. On the other hand, directly transcribing a mentally workable model into Coq can be a breeze, and if all relevant information is present in the goal, any mechanical process involved in the proofs can be automated. Therefore, it pays off to first formalize the

most intuitive representation of the and fix up performance in a separate proof or synthesis step.

- Synthesis mode with well-written scripts is very predictable; template metaprogramming is established. While the additional layers do complicate performance engineering, the extra effort is immediately made up for by the simplifications on the verification side.

3.4 Multi-Word Arithmetic

Our library uses the standard representation of numbers as multiple digits where each digit is associated with its weight, which is often left implicit when writing numbers. Even though we also want the weights to be implicit in the final code, we include them as a part of our fundamental representation of numbers for most of the code to simplify proofs. For example $(10a_1 + a_0)(10b_1 + b_0) = 100a_1b_1 + 10a_0b_1 + 10b_0a_1 + a_0b_0$ goes by `ring_simplify; trivial`, while proving $(10\lfloor a/10\rfloor + (a \bmod 10))(10\lfloor b/10\rfloor + (b \bmod 10)) = 100\lfloor ab/100\rfloor + 10(\lfloor ab/10\rfloor \bmod 10) + (ab \bmod 10)$ directly would be much more annoying. Later, we implement conversions between the associational and positional representation and have them fused with the arithmetic itself so that the associational representation of any number is never constructed at runtime.

Notice how the associational representation code in figure 3.4 passes the proofs even though the implementation of addition is rather sloppy: `add [(10,a)] [(10,b)] = [(10,a), (10,b)]` instead of `[(10, a+b)]`. Indeed, different limbs with the same weight are implicitly aggregated when the output is interpreted using `eval`, so we'd better follow the same convention when implementing the conversion from this representation to a positional number system. This particular hack is not crucial to this

Figure 3.4: The weights and values of digits are encoded in a list of pairs in Coq, representing $x = \sum_i^n w_i x_i$. The functions `fst` and `snd` return individual components of a pair; `fold_right Z.add` corresponds to a big $\sum$. Operators `::` and `++` prepend a single element or a list to another list respectively. `nsatz` proves polynomial equations using similarly formed givens.

```
Let limb : Type := Z*Z. (* weight and runtime value *)
Module Associational.
Definition eval (p:list limb) : Z :=
fold_right Z.add 0 (map (fun t => fst t * snd t) p).

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

general strategy, but the general idea of writing the easiest code that passes the proofs and *eventually* simplifies to something that runs fast shows up throughout this development.

Multiplication is only slightly more tricky: we will first verify multiplication of a multi-limb number by a single-limb number and then define general multiplication in terms of that. While the compound definition is reasonable even if the single-digit multiplication was inlined, the proofs would be noticeably messier in that style. Proofs by `induction` are sensitive to the exact statement of the goal at the moment `induction` is invoked, so it is good form to do that in the beginning of a proof where the goal directly matches the lemma statement. As a corollary, a proof that needs two inductions is best broken into two lemmas:

$$\sum_i (v w_i)(x y_i) = v x \sum_i w_i y_i$$

$$\sum_i \sum_j (v_i x_j)(w_i y_j) = \left(\sum_i v_i x_i\right)\left(\sum_j w_j y_j\right)$$

This completes our implementation and verification of basic ring operations in the associational representation. In some sense, it would be fair to say that it was cheating: I intentionally picked this representation to make the proofs

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

Lemma eval_map_mul (v x:Z) (q:list limb)
 : eval (map (fun t => (v*fst t, x*snd t)) q)
 = v * x * eval q.
Proof. induction q;
  autorewrite with push_map push_eval cancel_pair; nsatz.
Qed.
Hint Rewrite @eval_map_mul : push_eval.

Definition mul (p q:list limb) : list limb :=
  flat_map (fun t =>
    map (fun t' =>
        (fst t*fst t', (snd t*snd t')%RT)
      ) q
    ) p.
Lemma eval_mul p q : eval (mul p q) = eval p * eval q.
Proof.
  induction p; cbv [mul];
    autorewrite with push_flat_map push_eval cancel_pair;
    nsatz.
Qed.
Hint Rewrite @eval_mul : push_eval.
End Associational. Import Associational.
```

Figure 3.5: Code and proofs for multiplication. We will use the `(...)%RT` notation to mark operations that should not be evaluated during synthesis.

Throughout the development we formulate lemmas in the "operation pushing" style whenever possible: the `autorewrite` database `push_f` contains lemmas of the form $f(g(x)) = g'(f'(x))$, $f(a + b) = f'(a) + f'(b)$, etc. Here, `flat_map f (x::xs) = f x ++ flat_map f xs` happens to also be evident by computation if the definition of `flat_map` was unfolded but the conventional advice to avoid depending directly on the implementation of some library function is as applicable in proofs as it is in code.

easy. Yet these simple functions capture all we need to know about multi-digit multiplication or addition: more optimized versions can be derived by reasoning about the representation alone, leveraging the above functional correctness proofs for all operation-specific justification. But first, let's see what code would be generated from the current template alone.

The simplified expression contains the correct partial products, but again lists the two with weight 10 separately, rather than adding them together. Furthermore, most of the expression is clutter: whatever code specifically calls a base 10 2-digit multiplication function presumably already knows that it is going to get back three partial products with weights 100, 10, and 1, so the list structure and weights are redundant. The next representation will leave the weights implicit, and specify the number of limbs in the type.

Figure 3.6: A rather naive derivation of a formula for multiplying 2-digit numbers in base 10. Note that the original goal (the specification) only mentions the reference integer multiplication `*`, not our `mul`, which is introduced using a `rewrite` during the derivation.

`f_equal` proves `x=y -> f x = f y`. `cbv -[f]` performs partial evaluation, leaving `f` as is; in `(...)%RT`, `*` means the `runtime_mul` wrapper of `*`.

```
Lemma base10_2digit_mul (x y a b : Z) :
 {z | eval z = eval [(10,y);(1,x)] * eval [(10,b);(1,a)]}.
Proof.
  eexists ?[z].

  (*eval ?z = eval [(10,y);(1,x)] * eval [(10,b);(1,a)]*)
  rewrite <-eval_mul.
  (*eval ?z = eval (mul [(10,y);(1,x)] [(10, b);(1,a)])*)

  apply f_equal.

  (* ?z = mul [(10, y);(1,x)] [(10,b);(1,a)] *)
  cbv -[runtime_mul].
  (* ?z = [(100,(y*b));(10,y*a);(10,x*b);(1,x*a)]%RT *)

  trivial.
Defined.
```

3.5 Positional Representations

Figure 3.7: The `Context` command specifies common parameters to all following definitions and lemmas in the `Section`. Unused section variables are removed from definitions. `tuple T n` is a container for `n` values of type `T`.

```
Module Positional. Section Positional.
Context (weight : nat -> Z)
        (weight_0 : weight 0 = 1)
        (weight_nonzero : forall i, weight i <> 0).

Definition to_associational {n:nat} (xs:tuple Z n)
  : list limb
  := combine (map weight (seq 0 n)) (Tuple.to_list n xs).
Definition eval {n:nat} (x:tuple Z n) : Z
  := Associational.eval (to_associational x).
```

As the definition of length-indexed tuples is not a part of the standard library, our development contains numerous auxiliary definitions and lemmas. Nevertheless, the next two definitions in the arithmetic library (and the proofs of their basic properties) are rather verbose (20 lines total) due to manual manipulation of lists and tuples. Thus, I only present a brief summary:

- `zeros n : tuple Z n`, satisfying `eval (zeros n) = 0`.
- `add_to_nth {l} n x : tuple Z l -> tuple Z l` s.t. `eval (add_to_nth n x xs) = weight n*x+eval xs` if the index n is within the bounds of the tuple.

While 20 lines of spaghetti-proof about lists and tuples is a step down from the concise style so far, mixing arithmetic proofs and list/tuple proofs is much, much worse. Before deciding to have separate associational and positional representations, I chose to define each biginteger arithmetic directly on lists of digits, prove what a simple formula for the length of the output, and them wrap the function in a `tuple`-based API. This was a distinctively suboptimal choice: while we were able to build up the same basic operations as here and synthesize modular reduction code for pseudo-mersenne primes, getting to that point required over a thousand lines of code and a couple of months of work by two people. While hypothetical improvements within that design space might bring down the workload by a factor of a couple, I cannot imagine it being reduced as much as to have any chance of maintaining the reader's attention when sprinkling most of the code throughout this document.

The last code snippet (3.7) enables us to convert multi-digit numbers from positional representation to associational representation. This means that the 2-digit multiplication function synthesized in 3.6 would no longer need to take each digit individually as a parameter, rather the new inputs would be two positional-representation numbers of 2 digits each (`tuple Z 2`). However, to get the multiplication output back a positional representation, we need to recognize that $100 = 10^2$ to place the value next to it at index 2. Furthermore, in mixed-radix positional systems the weights of multiplication outputs do not necessarily match the weight of any position: for example $(1x_0 + 2^{26}x_1 + 2^{51}x_2 + \ldots)(1y_0 + 2^{26}y_1 + 2^{51}y_2 + \ldots) = (1x_0y_0 + 2^{26}(x_1y_0 + x_0y_1) + \mathbf{2^{52}}x_1y_1 + \ldots)$ even though each position has a well-defined weight $w_i = 2^{\lceil 25.5i \rceil}$. The multiplication output can still be forced back into the desired positional representation by replacing $2^{52}x_1y_1$ with $2^{51} \cdot (2x_1y_1)$. More generally: each term in the associational

representation will be assigned to the largest weight in the positional representation that it is divisble by, with runtime value scaled by the quotient.

```
Fixpoint place (t:limb) (i:nat) : nat * Z :=
  if dec (fst t mod weight i = 0)
  then let c := fst t / weight i in (i, (c * snd t)%RT)
  else match i with
       | O => (* never happens *) (0, fst t * snd t)%RT
       | S i' => place t i'
       end.
Lemma place_in_range t n : (fst (place t n) < 1+n)%nat.
Proof. induction n; cbv [place] in *;
  break_match; autorewrite with cancel_pair; 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; autorewrite with cancel_pair;
  repeat match goal with
  | [H: _ |- _] => progress (unique pose proof
           (Z_div_exact_full_2 _ _ (weight_nonzero _) H))
  end;
  trivial; nsatz.
Qed.
```

Figure 3.8: `Fixpoint` is like `Definition`, except it allows recursion on the structure of the arguments as examined using `match ... with`. Every `nat` is either zero (0) or a successor S.

`break_match` indicates that each case of any `if` or `match` should be considered separately. `omega` proves any true statement about linear arithmetic.

`repeat match goal with | [H:`**`pattern`**` |- `**`pattern`**`] => progress (`**`script`**`) | [ ... ] => ... end` is the general mechanism for repeatedly running different proof scripts based on what the givens and the goal look like. The construct keeps trying the different cases until none of them do anything. Here I use it to augment any hypotheses $a \bmod w_i = 0$ with $a = w_i\lfloor a/w_i \rfloor$.

To put everything together, a complete weight-value list is converted from associational representation to positional representation by initializing the output to all zeros and then adding each limb in the input in the correct place (figure 3.9).

This conversion function finally addresses the issue with repeated terms with the same weight that was postponed earlier: converting `[(100,(a1*b1));(10,a1*b0);(10,a0*b1);(1,a0*b0)]` to a 3-digit positional representation with $w_i = 10^i$ results in the expected lowest-digit-first tuple `(a0*b0, a1*b0 + a0*b1, a1*b1)`.

```
Definition from_associational n (p:list limb) :=
  List.fold_right (fun t =>
    let p := place t (pred n) in
    add_to_nth (fst p) (snd p)
  ) (zeros n) p.
Lemma eval_from_associational {n} p (n_nonzero: n <> 0)
 :  eval (from_associational n p) = Associational.eval p.
Proof.
  induction p; cbv [from_associational] in *;
    repeat match goal with
    | _ => progress (unique pose proof
         (place_in_range a (pred n)))
    | _ => progress (autorewrite with
         push_fold_right push_eval)
    | _ => progress (rewrite weight_place)
    | _ => progress (omega)
    | _ => progress (nsatz)
    end.
Qed.
Hint Rewrite @eval_from_associational : push_eval.
End Positional.
```

Figure 3.9: The correctness proof of placing each limb at the correct position in the positional representation is a straightforward combination of the properties of `add_to_nth`, `fold_left`, and the last two lemmas. Instead of specifying the exact combination in the proof, it is often convenient to write an unordered list of steps, each of which deals with one aspect of the problem. This style is used extensively in our library, and is so far the only sanity-preserving way I have found to describe a proof that handles hundreds of cases. After all, good lecturers just explain how to do the proof instead of writing out each step.

3.6 Synthesizing Curve25519 Modular Reduction

This section will review (and translate to Coq) the standard method for implementing reduction modulo numbers of the shape $2^k - c_l 2^{t_l} - \ldots - c_0 2^{t_0}$, encompassing what have been called "generalized Mersenne primes", "Solinas primes", "Crandall primes", "pseudo-Mersenne primes", and "Mersenne primes". Furthermore, while the trick in question is particularly fast when k and each t_i are multiples of the radix, the derivation here will cover the general case. In fact, the formalization does not even assume that the coefficients are powers of two as shown, even though this generality will most likely never be put to use.

Set $s = 2^k$, and $c = c_l 2^{t_l} + \ldots + c_0 2^{t_0}$ so $m = s - c$. To reduce x modulo m, find a and b such that $x = as + b$. Then

$$\begin{aligned} x \bmod m &= (as + b) \bmod (s - c) \\ &= (a(s - c + c) + b) \bmod (s - c) \\ &= (a(s - c) + ac + b) \bmod (s - c) \\ &= (ac + b) \bmod m \end{aligned}$$

The choice of a and b does not further affect the correctness of this formula, but it does influence how much the input is reduced: picking $b = x$ and $a = 0$ would make this formula a no-op. One might pick $b = x \bmod s$, although the formula does not require it (and it is often more efficient to not aim for the minimal b). Even if $b = x \mod s$, the final output $ac + b$ is not guaranteed to be the minimal residue. For many choices of a and b it is sufficiently smaller to significantly speed up future computations involving this number, which is why fast reduction is important.

Figure 3.10: A straigtforward transcription of the proof above. Chapter 6 presents dedicated scripts for automatically proving modular arithmetic facts, but I inlined a manual proof here to keep this self-contained.

The function `split` decomposes its argument x into (a, b). `eval_split` proves $x = sa + b$ (code in figure 3.11).

`mul` refers to the associational multi-digit multiplication defined in figure 3.5, expanding it for a particular c results in a modulus-specific reduction formula.

```
Lemma reduction_rule a b s c (modulus_nonzero: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.

Definition reduce (s:Z) (c:list limb) (p:list limb) : list limb
  := let ab := split s p in
     (fst ab) ++ mul c (snd ab).

Lemma eval_reduce s c p
  (s_nonzero:s<>0) (modulus_nonzero:s-eval c<>0)
  : eval (reduce s c p) mod (s-eval c)
  = eval p mod (s-eval c).
Proof.
  cbv [reduce].
  rewrite eval_app, eval_mul,
    <-reduction_rule, eval_split; trivial.
Qed.
```

In principle, `reduce` is parametric over `split`: any a and b such that $x = as + b$ suffice, and one could imagine using different choices in different contexts. However, all

modulus-specific reduction formulas I have found in the literature follow a very simple heuristic: a is picked as large as possible without requiring any runtime computation to extract it. Concretely: in an associational representation $x = \sum_i w_i x_i$, all limbs with $w_i \bmod s = 0$ are considered part of a, and the rest is b.

```
Fixpoint split (s:Z) (xs:list limb)
  : list limb * list limb
  := match xs with
     | nil => (nil, nil)
     | x::xs' =>
       let sxs' := split s xs' in
       if dec (fst x mod s = 0)
       then (fst sxs',    (fst x / s, snd x)::snd sxs')
       else (x::fst sxs', snd sxs')
     end.

Lemma eval_split s p (s_nonzero:s<>0)
  : eval (fst (split s p)) + s * eval (snd (split s p))
  = eval p.
Proof.
  induction p; cbv [split] in *;
  repeat match goal with
  | H:_ |- _ => progress (unique pose proof
        (Z_div_exact_full_2 _ _ s_nonzero H))
  | _ => progress(autorewrite with push_eval cancel_pair)
  | _ => progress (break_match)
  end; nsatz.
Qed.
```

Figure 3.11: `split` returns (b, a) following the least-significant-part-first convention in the code so far.

The correctness proofs in figures 3.10 and 3.9 share a structure that is quite common throughout our library:

1. Specify `induction`, if necessary.
2. Open up all definitions that will be reasoned about directly (as opposed to using already-proven facts).
3. `repeat-match-progress` with an arbitrarily-ordered collection of proof steps.

Assuming no step can turn a provable goal into an unprovable goal, and no infinite loops are created, this style allows for groups of common steps to be factored out from proofs whenever a meaningful pattern of what kinds of goals they prove is discovered.

Now we are ready to put everything together and derive a specific formula for multiplication modulo a prime chosen to make this operation particularly efficient.

Figure 3.12: `mulmod` is just a wrapper for definitions from the last three sections. The input and output are taken in compact positional representations, but the formula itself is derived using the associational representation that enables local reasoning about each limb.

The correctness is proven by `autorewrite` using the correctness lemmas for the definitions being wrapped.

The concrete goal is to compute $fg \bmod (2^{255} - 19)$ given f, g in positional representation with $w_i = 2^{\lceil 25.5i \rceil}$ as one would use on a computer with native 32-bit arithmetic.

The derivation itself is slightly more complicated than the previous ones because the desired implementation technique is not obvious from the goal itself. The script here manually passes 2^{255} and 19 as arguments to the `mulmod` function and then calls `vm_decide` to discharge the side conditons: the modulus is nonzero, the lowest weight is 1, etc.

As in figure 3.6, the the script uses `cbv -[]` to evaluate all functions except for those listed, and then also opens up the specified wrappers (using `cbv []`) to reveal the normal integer multiplication and addition.

```
Import Positional. Section Mulmod. Context
  (w: nat -> Z) (w: w 0 = 1) (w_nz: forall i, w i <> 0).
  (m:Z) (m_nz:m <> 0) (s:Z) (s_nz:s <> 0)
  (c:list limb) (Hm:m = s - Associational.eval c).
  Definition mulmod {n} (a b:tuple Z n) : tuple Z n
    := let a_a   : list limb := to_associational w a in
       let b_a   : list limb := to_associational w b in
       let ab_a  : list limb := mul a_a b_a in
       let abm_a : list limb := reduce s c ab_a in
       from_associational w n abm_a.
  Lemma eval_mulmod {n} (H:(n<>0)%nat) (f g:tuple Z n) :
    eval (mulmod f g) mod m = (eval f * eval g) mod m.
  Proof. cbv [mulmod]; rewrite Hm in *.
    autorewrite with push_eval; trivial.
  Qed.
End Mulmod.

Definition w (i:nat) : Z := 2^Qceiling((25+1/2)*i).
Lemma base_25_5_mul (f g : tuple Z 10) :
  { fg : tuple Z 10 | (eval w fg) mod (2^255-19)
          = (eval w f * eval w g) mod (2^255-19) }.
Proof.
  (* manually assign names to limbs for pretty-printing*)
  destruct f as [[[[[[[[[f9 f8]f7]f6]f5]f4]f3]f2]f1]f0].
  destruct g as [[[[[[[[[g9 g8]g7]g6]g5]g4]g3]g2]g1]g0].
  eexists ?[fg].
  erewrite <-eval_mulmod with (s:=2^255) (c:=[(1,19)]) by
      (try eapply pow_ceil_mul_nat_nonzero; vm_decide).
(* eval w ?fg mod (2 ^ 255 - 19)
     = eval w (mulmod w (2 ^ 255) [(1, 19)]
         (f9, f8,...f0) (g9, g8,...g0)) mod (2^255-19) *)
  eapply f_equal2; [|trivial]. eapply f_equal.
(* ?fg = mulmod w (2 ^ 255) [(1, 19)] (f9...) (g9...) *)
  cbv -[runtime_mul runtime_add];
  cbv  [runtime_mul runtime_add].

  ring_simplify_subterms.
(* ?fg = *)
(f0g9+   f1g8+   f2g7+   f3g6+   f4g5+   f5g4+   f6g3+   f7g2+   f8g1+   f9g0,
 f0g8+ 2f1g7+   f2g6+ 2f3g5+   f4g4+ 2f5g3+   f6g2+ 2f7g1+   f8g0+ 38f9g9,
 f0g7+   f1g6+   f2g5+   f3g4+   f4g3+   f5g2+   f6g1+   f7g0+ 19f8g9+ 19f9g8,
 f0g6+ 2f1g5+   f2g4+ 2f3g3+   f4g2+ 2f5g1+   f6g0+ 38f7g9+ 19f8g8+ 38f9g7,
 f0g5+   f1g4+   f2g3+   f3g2+   f4g1+   f5g0+ 19f6g9+ 19f7g8+ 19f8g7+ 19f9g6,
 f0g4+ 2f1g3+   f2g2+ 2f3g1+   f4g0+ 38f5g9+ 19f6g8+ 38f7g7+ 19f8g6+ 38f9g5,
 f0g3+   f1g2+   f2g1+   f3g0+ 19f4g9+ 19f5g8+ 19f6g7+ 19f7g6+ 19f8g5+ 19f9g4,
 f0g2+ 2f1g1+   f2g0+ 38f3g9+ 19f4g8+ 38f5g7+ 19f6g6+ 38f7g5+ 19f8g4+ 38f9g3,
 f0g1+   f1g0+ 19f2g9+ 19f3g8+ 19f4g7+ 19f5g6+ 19f6g5+ 19f7g4+ 19f8g3+ 19f9g2,
 f0g0+ 38f1g9+ 19f2g8+ 38f3g7+ 19f4g6+ 38f5g5+ 19f6g4+ 38f7g3+ 19f8g2+ 38f9g1)
  trivial.
Defined.
```

The resulting formula is not novel; it has been manually derived as a part of multiple performance engineering efforts targeting similarly constrained implementations. For example, the representation shown here appears on page 12 of “NEON crypto” by Bernstein and Schwabe along with advice about how to implement it efficiently using vector instructions.

3.7 Work Not Covered Here

This chapter described the parts of the field arithmetic library that I worked on. Since then, the framework has been extended to cover additional operations and implementation strategies. The details of that are the subject of another story, to be told another time. However, as the following chapters inherently rely on a broad verified field arithmetic library, here is a quick summary of the relevant functionality:

- Common subexpressions: the code derived in these examples expresses the output as a direct combination of the function inputs without reusing any intermediate computations. Arbitrary common subexpressions represented in the code using `let _ := _ in _` can be preserved during synthesis if the template code is first converted to continuation passing style.
- Carries: in the code shown, the values of the limbs increase. To fit them in machine registers, values are carried from lower limbs to higher limbs regularly throughout the computation.
- Saturated limbs: when the weights of the positions differ by a factor of the machine word size, even outputs of primitive operations such as addition or multiplication cannot necessarily be stored in single registers. Instead, the primitive limb addition and

multiplication produce output that has been split into two different variables.

- Modular canonicalization: the modular reduction formula shown here does not necessarily produce the smallest representative of the output congruence class, but as the choice of the representative might leak information, a full modular reduction is required at the end of the computation.
- Limb canonicalization: similarly, multiple different valuations of limbs can represent the same integer, again risking an information leak.
- Conversion between bases: as weights of the mixed-radix representations used in fast implementations are chosen based on hardware features of the target platform, the inputs and outputs need to be converted from the wire format and back.

Chapter 4

Certified Compilers for Bounded Arithmetic

In the previous chapter we derived correct-by-construction code for implementing modular arithmetic by representing each field element using a fixed number of limbs. The synthesis procedure and correctness proofs assume that each limb is represented as an arbitrary-precision integer. Fortunately, an informed choice of parameters leads to code where the value of each limb always stays within a desired range, thus enabling fast constant-time implementations using the arithmetic instructions and registers provided by the target CPU. For example, the code for multiplication modulo $2^{255} - 19$ derived in figure 3.12 uses 32-bit multiplication, 64-bit addition, and needs to store 32-bit and 64-bit intermediates assuming each input limb is between 0 and 2^{26}. Yet the last statement is not obvious, and failure to correctly account for the maximum range of values each limb can have during execution has resulted in several bugs (section 7.1). We want to have proofs that the code works correctly even if the arbitrary-precision integers are replaced with the native integer types of our favorite CPU.

4.1 Why Not "Just" Prove Lack of Overflow?

We could go back to the definitions and proofs in the previous chapter and try to augment them with the possible ranges of run-time integers. I claim that this would inherently need to break the modularity that made the definitions and proofs short. Currently, the proofs about multiplication consider partial products one at a time, completely independently of each other. Separately, the coefficients that appear in front of partial products on mixed-radix operations do not appear in the definition of multiplication but rather as a part of the conversion to positional representation. If we wanted to change the proofs to include ranges of these output integers, we would need to bring together all contributors to these ranges. This is directly contradictory to the strict separation of concerns we rely on to build short and simple proofs about arithmetic correctness.

In the first prototype of our field arithmetic library we did define all operations directly on the positional representation. We did not get to proving range properties of individual integers; just showing correctness of modular multiplication operating directly on a positional representation required tens of times more proof code than the version shown here. However, I find it highly unlikely that the range proofs would have been any simpler – the complications were not specific to the arithmetic properties being proven; they were caused by the general awkwardness of the chosen representation. The hacl-star development discussed in section 2.4 does include input and output range specifications for each bignum operation, and statements and proofs of related invariants make up a large fraction of the code.

We chose to keep our simple proofs of arithmetic properties as-is and establish possible ranges of individual limbs in a separate pass. The `gfverif` tool discussed in

section 2.5 also separates out range analysis from arithmetic verification. However, `gfverif` and this project can operate in opposite directions. `gfverif` starts from concrete C code where the range of values each variable can hold is unambiguously determined by its type and checks that these ranges are never exceeded. Doing so establishes that the sizes of variables do not matter and the code might as well have been using arbitrary-precision integers. Our field arithmetic library *produces* code that uses arbitrary-precision integers but does not provide any information about their ranges. In the end we want to output code that uses for each variable the fastest type whose range includes all values that this variable needs to be able to hold. Importantly, we do not need to verify that the transformation doesn't change the code in other ways – just knowing that it outputs the same value (although represented using a different type) is sufficient. This hints that the specification should be roughly of the form `∀ expr, interp (f expr) = interp expr`. Of course, the precise form of the correctness specification will depend on the implementation strategy, and in particular the representation used for the programs being transformed.

4.2 Synthesis Mode Proof Scripting?

The Coq system includes elaborate proof scripting capabilities that can be used for rewriting programs (see 3.3 for an arithmetic example). We tried to use proof scripting to implement range analysis, but opted for a different strategy after realizing that the inherent performance limitations of proof-producing automation are prohibitive in our use case. This section describes the now-abandoned attempt.

It would be rather convenient if we could prototype individual range analysis steps using small, illustrative goals and then glue them together to create what would essentially be a compiler, synthesizing code that uses

fixed-precision integers given code that uses arbitrary-precision integers. For example, we could use the basic properties of fixed-width words in the Bedrock library to straightforwardly prove the following rule:

Figure 4.1: Proposed rule for deriving code with fixed-size words.

```
wordToNat_add 64 : forall (n m:nat) (nw mw:word 64),
  wordToNat nw = n ->
  wordToNat mw = m ->
  n+m < 2^64 ->
    wordToNat (Word.add nw mw) = Nat.add n m
```

To synthesize finite-word code, we would state a goal like `{opt : word 64 | wordToNat opt = Nat.add a b}`, do `eexists` to get `wordToNat ?opt = Nat.add a b` and `eapply wordToNat_add` to get two subgoals `wordToNat ?1 = a` and `wordToNat ?2 = b`, which we could solve recursively. The third subgoal `a + b < 2^ 64` could be solved deriving specific bounds on a and b and showing that the sum of these bounds is below `2^64`.

This approach does not actually work because it requires duplicating subexpressoions to state their ranges. Proving $f(g(x)) < B_f$ inherently requires proving $g(x) < B_g$ before proving $f(g(x)) < B_f$ but the synthesis procedure above consumes these facts in the opposite order, requiring $f(g(h(x))) < B_f$ before $g(h(x)) < B_g$. As the former does not imply the latter, the context would need to contain both $f(g(h(x))) < B$ and $g(h(x)) < B_g$ at the beginning of the derivation to avoid re-doing the entire bounds analysis for each synthesis step. An expression tree of depth n would have $O(n)$ subexpressions of size $O(n)$ each, so storing all subexpressions with their bounds would require $O(n^2)$ space[1] . Similarly, re-proving the bounds of all subexpressions from scratch for every synthesis step would require quadratic time. Yet we want to output code with thousands of primitive integer operations in it, and creating millions of expression tree nodes is far beyond what one could expect to finish in a reasonable time when executed in the rather simplistic interpreter Coq uses for proof scripts.

1. Careless inlining of common subexpressions can even cause exponential blow-up, but it is avoidable. It is naturally avoided in the solution presented in the next section.

The conclusion of back-of-the-envelope calculation was experimentally confirmed by one of us trying to implement this strategy anyway.

Even if Coq performance issues were magically fixed, I would still advocate for abandoning this strategy: doing quadratic work for annotating each integer with a type based on its range is as far from intuitive as it can get; a correct solution would use the range of the previous intermediate expression to determine the range and the new code for the next operation.

4.3 Certified Range Analysis

If we do not want to prove the ranges of the outputs of the generalized field operations and cannot find a way to write a script to generate a proof for any particular implementation, what else can we do? More abstractly, the situation is as follows: we don't want to write a proof about all code generated by our template, and don't want to run proof scripts to generate case-by-case proofs for any piece of generated code. At a first glace, these two options look like the opposite ends of the proof generality spectrum and suggest that a good solution, if any, must be in the middle. Instead of starting to try out various intermediate degrees of integration, I suggest re-thinking the problem statement in even more general terms, without constraining ourselves to thinking about the code generated from the field arithmetic library: it is obviously sufficient (and I claim, easier!) to verify a procedure for range analysis on arbitrary straight-line code consisting of known integer operations. This way, the range range of any specific code (be it synthesized by the field arithmetic library or not) can be proven by simply appealing to the correctness proof of the transformation that performs range analysis and picks appropriate types for variables.

It is rather straightforward to imagine how a range-based

type selection compiler could be implemented: to determine the range of $f(x)$, first recursively determine the range of x and then use the knowledge about f to bound its output range given its input range. For example, if $f(x) = x^2$ and we know $-4 \leq x \leq 3$, then $0 \leq f(x) \leq 16$. Fixed-size integers can be assigned based on the ranges: perhaps it makes sense to store $f(x)$ in a single byte. Making this work while preserving common subexpressions requires slightly more care: to infer the range of `let` y `:=` $f(x)$ `in` $g(y)$, the analysis first finds the range of $f(x)$, then uses it to find the range of $g(y)$, and reports the latter as the range for the overall expression.

Code Representation

To implement a compiler, we will first need to define the syntax and semantics of the language it accepts. I chose to use a simply typed representation with named (or if you prefer, numbered) variables, parametrized over the available types and operations. For ease of understanding, I will present the most minimal prototype that contains just these features and is specialized to binary operations. However, rest assured, none of these simplifications are essential to the range analysis and optimization: after I built the prototype I describe here, the core ideas were ported to a much richer language already used in our project. Read figures 4.2 and 4.3 and try to map field arithmetic code (such as that derived in figure 3.12) to explicit expressions this form.

Of course, the field library does not produce syntax trees in the language just prescribed, it produces regular Coq code. However, if we somehow aqcuired an `e` representing the synthesized code, it would be easy to check (and prove!) that it is correct: partially evaluating `interp ctx e` would return the exact synthesized code. We use a simple unverified script to crawl the synthesized code and construct `e`.

```
Section language.
  Context
    (name : Type)
    (type : Type)
    (opcode : forall (t1 t2 tR : type), Type).

  Inductive expr : forall (t:type), Type :=
  | BinOp
      {t1 t2 tR : type}
      (op : opcode t1 t2 tR)
      (e1 : expr t1)
      (e2 : expr t2)
    : expr tR
  | LetIn
      (x : name)
      {tx : type}
      (ex : expr tx)
      {tC : type}
      (eC : expr tC)
    : expr tC
  | Var
      {tx : type}
      (x : name)
    : expr tx.
```

Figure 4.2: Complete structure of the minimal language I used to prototype and explain range analysis and optimized representation selection.

By convention, lowercase identifiers such as `type` represent actual constructs in the language on which the compiler operates whereas uppercase `Type` refers to Coq types.

While the definition of expressions here does not go out of its way to ensure that the code is well-formed (for example, it is possible to have unbound variables), the Coq type representing expressions is indexed by the type of the expression, and binary operations can only be constructed if the types match.

```
  Context
    (interp_type : forall (t : type), Type)
    (interp_op : forall (t1 t2 tR) (op:opcode t1 t2 tR)
      (v1:interp_type t1) (v2:interp_type t2),
      interp_type tR)
    (ValueContext : context name interp_type).

  Local Notation "x <- mx ; y" :=
    (match mx with
     | Some x => y
     | None => None
     end) (only parsing).

  Fixpoint interp (ctx : ValueContext) {t} (e : expr t)
    : option (interp_type t)
    := match e with
       | BinOp _ _ _ op arg1 arg2
         => v1 <- interp ctx arg1;
            v2 <- interp ctx arg2;
            Some (interp_op _ _ _ op v1 v2)
       | LetIn x _ ex _ eC
         => vx <- interp ctx ex;
            interp (extendb ctx x vx) eC
       | Var t x
         => lookupb ctx x t
       end.
```

Figure 4.3: Complete semantics of the minimal language for range analysis. As the syntax is parametrized over the types and the operations, the semantics is parametrized over the mapping from syntactic types to Coq types, and the specifications of the operations.

As interpreting a program may fail due to unbound variables, we will need to propagate errors through the interpretation. The notation is just for the convenience of reading.

`ValueContext` implements a mapping from names to types and values. It can be manipulated using `extendb` and `lookupb`.

Range Analysis, Dependently Typed Programming

Figure 4.4 contains a readability-oriented version of the range analysis and optimized type selection transformation. The remainder of this section is dedicated to how to actually write this code: trying to come up with the form shown here line-by-line is unlikely to result in anything but frustration. In particular, the code in figure 4.4 is only readable in practice because all confusing details are carefully left implicit, to be inferred by Coq's type inference during type checking. Unfortunately, type inference only works reliably on well-typed code; the error messages for non-type-annotated code correspond to arbitrary guesses of which parts of the code could be changed to make it type-check and are hardly ever helpful. Just re-printing the function using the `Print` command produces output twice as long as the shown code due to return type annotations on `match` and `let` expressions; further giving up implicit arguments and notations by first doing `Set Printing All` produces 170 lines. Writing all of it out by hand would be type-error-prone and tedious.

Fortunately, even though the rules for typechecking dependently typed Coq code are alien to most earthly programmers, they can be summarized on a napkin, learned, and applied mentally without the help of a computer[2]. The key insight that helped me grasp dependent types is that if types are allowed to depend on values, type checking must rely on evaluation. For example, the type checker looking up a type by its alias is no different from the interpreter looking up the value assigned to a variable. This means that knowing the rules of computation and the exact definitions of everything that is used as a type is crucial for mentally typechecking code. For example, types `tuple T 5` and `tuple T (2+3)` are interchangable but `tuple T (a+b)` and `tuple T (b+a)` are not because `a + b = b + a` does not follow from computation alone, it requires proof.

The case of `match` statements is particularly confusing

2. This is in stark contrast to numerous other dependently typed languages where type checking inherently relies on heuristics (perhaps because it is undecidable in general). One example of this is F*, which uses an SMT solver for type-checking. Indeed, the hacl-star framework files are sprinkled with commands to tune various heuristics and timeouts of the solver.

```
Context
  {range : forall type, Type}
  (type_for_range : forall {t} (r:range t), type)
  (op_range :
    forall {t1 t2 tR} (op:opcode t1 t2 tR)
           (r1:range t1) (r2:range t2), range tR)
  (cast_op :
     forall t1 t2 tR
            (op:opcode t1 t2 tR)
            (r1:range t1)
            (r2:range t2),
       opcode (type_for_range r1)
              (type_for_range r2)
              (type_for_range (op_range op r1 r2)))
  {RangeCtx : context name range}.

Local Arguments type_for_range {t}. (* now implicit *)
Local Arguments op_range {t1 t2 tR}.
Local Notation "( a , b )" := (existT _ a b).

Fixpoint type_by_range (ctx : RangeCtx) {t} (e : expr t)
  : option { r : range t & expr (type_for_range r) }
  := match e with
     | BinOp t1 t2 tR op arg1 arg2
       => b1 <- type_by_range ctx arg1;
          let '(range1, arg1') := b1 in
          b2 <- type_by_range ctx arg2;
          let '(range2, arg2') := b2 in
          let rangeR := op_range op range1 range2 in
          let op' := cast_op t1 t2 tR op range1 range2 in
          Some (rangeR, BinOp op' arg1' arg2')
     | LetIn x tx ex tC eC
       => bx <- type_by_range ctx ex;
          let '(range_x, ex') := bx in
          let ctx' := extendb ctx x range_x in
          bC <- type_by_range ctx' eC;
          let '(range_C, eC') := bC in
          Some (range_C, LetIn x ex' eC')
     | Var t x
       => range <- lookupb ctx x;
          Some (existT _ range (Var x))
     end.
```

Figure 4.4: As the language is parametrized over types, the range analysis is parametrized over the representation of ranges of these types and rules for propagating range information across operations. Arbitrary heuristics for picking new types based on a derived range can be passed in – it is often unwise to use the type with smallest range of values of those applicable because might be slower than a less restrictive type. As a technicality, `cast_op` is needed to turn an operation on the original types into an operation on the inferred types (even though both might be printed as `*`) because the type of an operation needs to match its operands.

Range analysis and representation type selection are performed in one pass, and the return type of the recursive function encodes the relationship between the inferred range and type. `{ x : A & B x }` is the type of pairs where the first component has type `A`, and the type of the second component is determined by passing the value of the first component, `x : A`, into the function `B : forall (x:A), Type`. Here `B` specifies the inferred return type of the new expression given its inferred range, the type of which depends on the original return type.

because even though the variable being matched has a known structure in each case of the match and this information is used when checking the return type of the case, it is not considered when referencing any variables bound outside the match. In particular, while types of variables bound outside a match may contain references to the variable being matched on, these occurrences remain unchanged across the typechecking of all cases instead of[3]. being filled in with the constructor and variables bound in each case To incrementally construct large dependently typed definitions, it is helpful to use the Coq interactive mode normally used for proofs, inserting _ for code yet to be written. The generated goal will show all information that would be used to type-check the omitted code. For example, consider the following attempt at implementing range analysis with a more specific but incorrect type annotation:

3. For a more complete explanation of dependent pattern matching (for example, for the case where the expression being matched on is not a variable), I recommend CPDT section 8.2 (Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant*. MIT Press, 2013. URL: http://adam.chlipala.net/cpdt/cpdt.pdf).

```
Fixpoint type_by_range (ctx : RangeContext) t (e:expr t)
  : option { r : range t & expr (type_for_range r) }.
  refine (
     match
       e
     return
       option { r : range t & expr (type_for_range r) }
     with
     | BinOp t1 t2 tR op arg1 arg2 => _
     | _ => _
     end
  ).
  (* Givens: ..., op : opcode t1 t2 tR, ... *)
  (* Goal: option {r:range t & expr(type_for_range r)}*)
```

Figure 4.5: The interactive display shows that all connection between t, the type of e, and tR, the return type of op, has been lost. Even though the definition of expr would not allow a BinOp e to be well-typed if tR didn't match t, from the perspective of the match, t is just any old variable that might be bound arbitrarily far away. No value of _ would make this typecheck.

As t appears in the type of e, we can rebind it inside the match by adding an in clause to the match:

```
  refine (
     match e
     in expr t
     return
       option {r : range t & expr (type_for_range r)}
     with BinOp t1 t2 tR op arg1 arg2 => _ | _ => _ end
).
  (* Givens: ..., op : opcode t1 t2 tR, ... *)
  (* Goal: option {r:range tR & expr(type_for_range r)}*)
```

Figure 4.6: The correct type annotation for the outermost match in type_by_range. The interactive-mode feedback shows tR both in the goal and in the type of op.

Variables not appearing in the type of the variable being `matched` on can be bound inside the `match` by having it return anonymous functions of those variables:

```
Local Set Universe Polymorphism. (* for [unit : Type] *)
Definition some_tuple_if_nonzero (n:nat) :
  forall (_:match n with 0 => unit | S n' => tuple n end),
    option { r : nat & tuple r }.
  refine (
    match n
    return
      (forall (tn:match n with 0=>unit |S _=>tuple n end),
                option { r : nat & tuple r })
    with
    | 0 => fun (useless:unit) => None
    | S n'=> fun (ntuple : tuple (S n')) =>
                                       Some (S n', ntuple)
    end).

Definition dep2sum
    (b:bool) : forall (_:if b then Z else Q), Z + Q.
  refine (match  b
          return forall (_:if b then Z else Q), Z + Q
          with
          | true  => fun (x:Z) => inl x
          | false => fun (x:Q) => inr x
          end).
```

Figure 4.7: If `n` is `0`, the input contains no other information. If `n` is nonzero, the input contains a tuple of length `n`. The function returns a tuple and its length if one is available and `None` otherwise. Note that the argument that potentially contains the tuple is not even named outside the `match` by putting an `_` instead of its name. However, it is bound in an anonymous function inside each case of the `match`. Replacing one `match` case with a `_` would show a goal with the type of `tn` appropriately related to `n`.

Correctness Proof

I prove the correctness of the combined range inference and finite type selection optimization using a single induction over the structure of the input program. To keep the task manageable, the main proof takes as parameters the correctness proofs of bounds inference rules for individual operations, and the fact that the finite type chosen for each range can in fact represent all values in that range so that they can be cast back losslessly:

When stating the correctness of the transformation, it is tempting to try to talk about only expressions that can be evaluated in an empty context. This is a bad idea in two ways. First, this notion of correctness is not suitable for induction: when transforming a `LetIn`, the recursive call is made with a non-empty context, so having an inductive

Figure 4.8: Correctness requirements the range analysis transformation. Note that while the language being modeled may contain casts, the `cast_back` function here is only used to state the correctness lemma, it does not appear in output code. `interp_op_bounds_correct` and `cast_back` are only used in the correctness proof not shown here.

```
Local Arguments interp_op {t1 t2 tR}.
Context
  (in_range : forall t (r:range t), interp_type t-> Prop)
  (interp_op_bounds_correct:
    forall t1 t2 tR op r1 r2
      (v1 : interp_type t1) (v2 : interp_type t2)
      (H1 : in_range t1 r1 v1) (H2 : in_range t2 r2 v2),
    in_range tR (op_range op r1 r2) (interp_op op v1 v2))
  (cast_back: forall {t r}
    (v:interp_type (type_for_range r)), interp_type t)
  (pull_cast_back:
     forall t1 t2 tR op r1 r2
            (v1 : interp_type (type_for_range r1))
            (v2 : interp_type (type_for_range r2))
            (H1 : in_range t1 r1 (cast_back v1))
            (H2 : in_range t2 r2 (cast_back v2)),
       interp_op op (cast_back v1) (cast_back v2) =
 cast_back (interp_op (cast_op t1 t2 tR op r1 r2) v1 v2))
```

hypothesis that only applies to empty calls with contexts is unhelpful. Worse, it does not even capture the notion of correctness we care about: the `expr` type is used to represent bodies of functions, and since the functions can have arguments, the context would most likely not be empty. Instead, the proof quantifies over all possible contexts in which the input and transformed code is run, and requires as a precondition that the values in these are appropriately related to each other and to the "context" for tracking ranges of variables.

Figure 4.9: Complete inductive correctness statement for range analysis and automatic selection of finite types and operations. The key to reading this one is to notice that most variables are uniquely specified by the precondition stated right after introducing them; the only reason for using a quantifier is that it is not obvious that such a value exists.

```
Lemma type_by_range_correct {t} (e:expr t) :
  forall {r} e' (rangeCtx : RangeContext)
  (He' : type_by_range rangeCtx e = Some (existT _ r e'))
  (oldValues : ValueContext) (newValues : ValueContext)
  (Hctx : forall {t} n v,
      lookupb oldValues n = Some v
      -> exists r, lookupb rangeCtx n = Some r
                   /\ in_range _ r v
                   /\ exists v',
                       lookupb newValues n = Some v'
                       /\ cast_back t r v' = v)
  v  (Hr  : interp oldValues e  = Some v )
  v' (Hr' : interp newValues e' = Some v'),
  in_range _ r v /\ cast_back _ _ v' = v.
```

The proof starts by invoking `induction e` right away: the location of `: forall` in the statement separates

variables that stay the same throughout induction from those on the right that are allowed to vary. This is particularly important because recursive calls use different contexts than the caller, and thus require an inductive hypothesis that is valid for all contexts. The `BinOp` and `Var` cases are solved by a straightforward combination of the inductive hypotheses and the fundamental properties of `cast_back` and `op_range`. Unfortunately, while the `LetIn` case is conceptually even simpler ("if we add a variable to the context with the correct range, looking it up later will return the correct range"), the first order representation of context and the possibility of unsuccessful lookups inflate the proof to 80 lines of unexciting instructions for reasoning about contexts.

4.4 Why Did This Optimization Not Work in PHOAS?

Proof overhead caused explicit by context manipulation is a well-known issue in compiler verification; that it shows up here is completely unsurprising. In fact, the bulk of the compiler-like transformations in our codebase are implemented in a style whose main selling point is that it reuses Coq's own context manipulation mechanisms for the embedded language: parametric higher-order abstract syntax or PHOAS[4]. Initially, we tried to implement range analysis and type selection as PHOAS transformations. This section will describe why the implementation strategy shown above is not compatible with PHOAS. To the best of my knowledge, this is the first documented limitation inherently caused by the use of Coq binders to represent object-language binders.

Unlike most of this thesis, this section is only tangentially related to cryptographic implementations. If you are not interested in compiler verification, it is safe to skip ahead: nothing in the rest of this thesis references this section.

4. Adam Chlipala. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics". In: *Proc. ICFP*. 2008, pp. 143–156. URL: `http://adam.chlipala.net/papers/PhoasICFP08/PhoasICFP08.pdf`.

Syntax and Semantics in PHOAS

The key idea behind PHOAS is to use Coq functions to represent code that references previously bound variables so that Coq function application can be used to substitute in information about the variable. This directly addresses the concern of undefined variables left open in figure 4.2: the second expression in a `LetIn` node is replaced with a function from the information about the bound variable to an expression. This makes it impossible to access the remainder of the program without first filling in the bound variable, essentially eliminating the proof effort spent on handling the case where a variable appearing in an expression is not associated with a value in the context.

Of course, interpreting an expression where `eC` does something other than substitution would not match our intuition of what an expression is, even though it is technically defined above. These cases need to be excluded in correctness proofs of all transformations that change the binder structure: each one requires an expression well-formedness precondition which is proven automatically for each program.

Producing Code from a PHOAS `expr`

We already saw an example of how to write a function that returns a simply typed value based on a PHOAS expression – the interpreter does just that. Other information can be gathered in a similar manner: A function computing the number of `BinOp` nodes in an expression would recurse similarly to `interp` but wouldn't need to substitute anything for variables. An estimate of the resulting size of the expression in case all `LetIn`-bound were substituted in can be computed by annotating each variable with `var t := nat` – the size of the fully inlined expression for that variable.

Returning code is more complicated because the `var`

```
Section Syntax.
  Context {var : forall (t:type), Type}.
  Inductive expr : forall (t:type), Type :=
  | BinOp
      {t1 t2 tR : type}
      (e1 : expr t1)
      (e2 : expr t2)
      (op : opcode t1 t2 tR)
    : expr tR
  | LetIn
      {tx : type}
      (ex : expr tx)
      {tC : type}
      (eC : forall (x:var tx), expr tC)
    : expr tC
  | Var
      {t : type}
      (v : var t)
    : expr t.
End Syntax.
Local Arguments expr : clear implicits.

Fixpoint interp {t:type} (e:expr interp_type t)
  : interp_type t
  := match e with
     | BinOp _ _ _ e1 e2 op
       => interp_op op (interp e1) (interp e2)
     | LetIn _ ex _ eC
       => let x := interp ex in
          interp (eC x)
     | Var _ v
       => v
     end.
```

Figure 4.10: PHOAS syntax and semantics for the same language defined using a simply typed representation in figures 4.2 and 4.3.

Different values of `var t` can be specified to implement transformations that track different information about each variable of type `t`. Interpreting a program is done with `var t := interp_type t`, so every variable is annotated with the value assigned to it.

The `LetIn` case of `interp` first interprets the expression for the bound variable to get `x : interp_type tx`. The code after the `LetIn` is represented as a function `eC` from `var tx` to `expr tC`. We picked `var t := interp_type t`, so we can plug in the value by calling it: this case returns `interp (eC x)`.

The value plugged into `eC` is encountered again inside the `Var` case: with the chosen `var`, the `expr` type guarantees that every `Var` of type `t` will contain a value of type `interp_type t` when it is `match`ed on.

annotation is dictated by the next function the resulting code is passed into. To avoid coupling tranformations to each other, it makes sense to parametrize each transformation over the `var` required by the next one. But how would the first transformation construct a `Var` node which requires a value of type `var t` without knowing `var`? For all it can tell, `var t` might be a type with no values at all!

This issue is solved in two parts. First, we can move the problem around by requiring each `Var` node in the input

expression to be annotated with the correct answer (`inputvar t := expr var t`), making the `Var` case trivial. This creates a second complication in the `LetIn` case: now `eC` needs an `expr var tx` as input before revealing the code after the let binder. However, the input `eC` is only used to produce the return value of the output post-binder code `eC'`, and `eC'` gets `v : var tx` as input! Thus, we fill `eC` with `Var v`.

Figure 4.11: PHOAS implementation of the identity transformation that does not change the code at all. Real compiler optimizations of course implement more sophisticated logic but the PHOAS-specifc bits are essentially the same as here.

```
Section WithVar.
  Context (var : type -> Type).
  Fixpoint ident {t:type} (e:expr (expr var) t)
    : expr var t
    := match e with
       | BinOp _ _ _ e1 e2 op
         => BinOp (ident e1) (ident e2) op
       | LetIn tx ex _ eC
         => let ex' := ident ex in
            let eC' := fun(v:var tx)=>ident (eC(Var v)) in
            LetIn ex' eC'
       | Var _ v => v
       end.
```

Producing Data *and* Code from a PHOAS `expr`?

While the patterns for producing data or code are relatively lightweight, they are also mutually incompatible. It is not sufficient, for example, to set var t := nat * expr var t because there is no way to pass a function (eC) half of its input in return for half of the output. To see the issue more closely, consider the following attempt at writing a function that translates all types in an `expr`.

In short, the types just don't line up. We briefly experimented with adding casts to the code and manually controlling evaluation order to bring the time complexity back down to linear, which worked. However, the resulting code was cluttered beyond what I could effectively reason about, so I did not even attempt to prove it correct in Coq.

As in the field arithmetic case study, different representations of the same concept can have drastically different ergonomics in different uses: I wouldn't want to

```
Context (f_op : forall {t1 t2 tR} (op:opcode t1 t2 tR),
          forall t1' t2', {tR':type & opcode t1' t2' tR'}).
Local Arguments f_op {t1 t2 tR}.
Fixpoint f
  {t:type} (e:expr (fun _ => { t:type & expr var t } ) t)
  : { t:type & expr var t}
  := match e with
     | Var _ v => v
     | BinOp t1 t2 tR e1 e2 op =>
       let (t1', e1') := f e1 in
       let (t2', e2') := f e2 in
       let (tR', op') := f_op op t1' t2' in
       (tR', BinOp e1' e2' op')
     | LetIn tx ex tC eC =>
       let (tx', ex') := f ex in
       let vx'' : var tx' := _ (* ??? *) in
       let tC' := projT1 (f (eC (tx', Var vx''))) in
       let eC''
         : forall v, expr var (projT1 (f (eC (tx',Var v))))
         := fun v => projT2 (f (eC (tx', (Var v)))) in
       let eC' : forall (v:var tx'), expr var tC' := _ in
       (tC', LetIn ex' eC')
     end.
```

Figure 4.12: In the `LetIn` case, we would like to call `f` recursively to determine the output type, but neither pattern applies. We cannot just substitute the `tx'` into `eC` because `eC` expects `tx` and a corresponding variable to be used in the `Var` case. If we assume access to a suitable `var`, we can fill the arguments on a `var` to pass into `eC` to get `tx'`. Notice that we can't get that `var` from a `LetIn` as in the definition of `ident`, because `t` in `t : type  expr var t` needs to be *outside* of the `LetIn`.

Even if we managed to make one recursive call to get `tx'` (perhaps by making up a meaningless `expr` for `eC`), we would need to do another recursive call to actually output code `eC"`, degrading the time complexity from linear to exponential. Worse, the type of `eC"` does not match the one dictated by `tx'` due to the use of a `Var` different from the one used for the first recursive call.

write an inliner in a first order representation, and apparently I cannot write a range analysis optimized in PHOAS. Fortunately, translating code between the two representations described here is uncomplicated.

Chapter 5

Elliptic Curves

5.1 Why Verify Elliptic-Curve-Level Code?

It is tempting to think that low-level arithmetic is responsible for vast majority of the complexity in cryptographic implementations. I consider this to be an oversimplification: the higher abstraction levels are surprisingly tricky, and verifying them is also worthwhile. This section will review some sources of complexity in elliptic-curve-level code.

Minimizing Operational Cost Using Efficient Representations

Any serious implementation of elliptic curve arithmetic would use a representation different from the whiteboard-level specification. Adding two points in a naive representation requires a division of field elements, which is hundreds of times slower than the other operations. As a cryptographic operation involves at least one point-scalar multiplication, which is implemented as a series of point additions, it is almost always beneficial to change to the most convenient coordinate system for performing the bulk of the computation and then transfer the answer back.

Optimized coordinate systems for elliptic curve arithmetic are a subject of active research. The Explicit Formulas

Database[1] contains tens of representations and hundreds of addition procedures, including specialized addition formulas for certain values of curve parameters for which especially good performance can be achieved. A fast addition formula might contain no divisions plus a handful of coordinate multiplications and coordinate additions[2].

As not every elliptic curve admits a single addition formula that works for all pairs of points (without triggering a division by zero or equivalent), most addition procedures specify multiple cases. If the condition that determines which case is used must remain secret, both cases need to be computed – otherwise an attacker could infer the secret condition by measuring the time taken by the addition computation. This is quite common: when multiplying a point by a secret scalar, the choice of which points to add (and thus which cases of the addition procedure are triggered) depends directly on the bits of the secret key. This means that the work required for an addition formula with multiple cases is the sum of the amounts of work for each case, resulting in a strong incentive to minimize the number of cases.

It is not obvious that a couple of footnote-sized formulas would require formal verification: the existence of fast addition formulas is well-known; the formulas themselves are relatively simple, and appear in peer-reviewed publications. However, the editors of the explicit formulas database discovered 2 significant errors in the literature when they checked 123 formulas using the Magma computer algebra system[3], and to my best understanding, there was no systematic checking of side-conditions of the formulas.

Applicability of Optimizations

While the formulas themselves can be simple, the conditions for their applicability are not necessarily so. Yet there is a strong drive towards using faster, more specialized formulas, and some very desirable formulas have tricky side

1. Daniel J. Bernstein and Tanja Lange. *Explicit-formulas database*. URL: https://hyperelliptic.org/EFD/.

2. For example, on a complete twisted Edwards curve $(-1)x^2 + y^2 = 1 + dx^2y^2$ represented using (X, Y, Z, T) s.t. $x = X/Z$, $y = Y/Z$, and $xy = T/Z$, point addition requires 8 multiplications, 8 additions and 2 doublings:
$A = (Y_1 - X_1)(Y_2 - X_2)$
$B = (Y_1 + X_1)(Y_2 + X_2)$
$C = T_1(2d)T_2$
$D = Z_1 2Z_2$
$E = B - A$
$F = D - C$
$G = D + C$
$H = B + A$
$X3 = EF$
$Y3 = GH$
$T3 = EH$
$Z3 = FG$

3. Daniel J. Bernstein and Tanja Lange. *The Explicit-Formulas database*. Sept. 5, 2007. URL: https://cr.yp.to/talks/2007.09.05/slides.pdf, page 4.

conditions. It is a pattern that the fastest procedures just barely cover all cases – if there was a case that could be left out in a way that improved performance, we would have an even faster procedure.

For example, Joost Renes, Craig Costello, and Lejla Batina recommend an addition procedure consisting of a single 40-step formula for use with elliptic curves in Weierstrass form. The formula in question does not work correctly for all points for possible parameters of the Weierstrass curve: in particular, it computes an incorrect value for $P + Q$ if $P \neq Q$ but $2P = 2Q$ (i.e., $P - Q$ has order exactly 2) [4]. Nevertheless, this formula is considered very desirable even for use on curves that do contain points of order 2 because many computations never produce those points. However, code using that formula with a curve that does have points of order 2 needs to be carefully vetted.

The optimized addition laws I chose to implement for Edwards and Montgomery curves also have non-trivial side conditions but the former only requires information about static parameters and the latter is accompanied with an alternative proof that the output is *safe* in the sense that no secret information is leaked even when the preconditions are not met.

Library-wide Invariants

Even more commonly, elliptic curve code requires that the points that it operates on actually satisfy the curve equation. Otherwise, the result is meaningless but may leak any secret inputs to the computation[5].

Therefore, the function that converts a point from a space-efficient representation used for transmission to the internal representation must never output a point that does not satisfy the curve equation. Again, the point decoding functions for commonly used representations admit sophisticated optimizations, leaving room for design and implementation error.

4. Joost Renes, Craig Costello, and Lejla Batina. "Complete addition formulas for prime order elliptic curves". In: *Proceedings of the 35th Annual International Conference on Advances in Cryptology, New York, USA*. Springer-Verlag, May 8, 2016. URL: https://eprint.iacr.org/2015/1060.pdf, section 5.2.

5. A point that is not on the desired curve may be on some other curve. If the point is multiplied by the secret key and sent over the network as one might in a Diffie-Hellman handshake, the attacker would learn the value of the secret key modulo the order of a (small) curve of their choice by computing a discrete logarithm on that curve. Multiple such measurements can be used to reconstruct the secret key using the Chinese Remainder Theorem. Tibor Jager, Jorg Schwenk, and Juraj Somorovsky. "Practical Invalid Curve Attacks on TLS-ECDH". in: (2015). URL: http://euklid.org/pdf/ECC_Invalid_Curve.pdf.

Constraints of Higher-Level Protocols

Implementing multiple kinds of cryptographic functionality can create even more tension between simplicity and cost. For example, the best-known elliptic curve protocol for verifiable random functions requires a Diffie-Hellman group of prime order but the exception-free Montgomery ladder strategy for Diffie-Hellman key agreement requires the curve order to be divisible by 4. Adding a new cryptographic object to an application would require implementations of its operations for all platforms, additional key management, and in case of an existing deployment, a backwards compatibility layer. In many cases, these costs often outweigh the benefits of simply using a protocol "as it says on the label". Instead, the example conflict in this paragraph can be resolved through clever engineering by choosing a curve of order $4p$ where p is a prime, and having the verifiable random function protocol operate on a specific order-p subgroup. Yet the required additional code (for testing subgroup membership) is far from obvious, and any omission could easily remain undetected as during normal operation the checks (if implemented) would silently pass.

5.2 Specifying Elliptic Curves

To verify either representations of elliptic curves or code that uses elliptic curves, we first need a specification of the elliptic curves, their operations, and proofs of basic properties of these operations. This section will briefly review elliptic curves of Edwards, Montgomery and short Weierstrass form and explain the way these definitions are written in Coq. The proofs of basic properties of group structure will be covered in the next section to the extent they are specific to the elliptic curve in question – most of the heavy lifting is done by the field equation prover described in the next chapter.

Twisted Edwards Curves and Invariants in Types

A twisted Edwards curve over a field F (of characteristic at least 3) with parameters a and d is defined to contain the points (x, y) s.t. $ax^2 + y^2 = 1 + dx^2y^2$. Addition of points $(x_1, y_1) + (x_2, y_2)$ is defined similarly to addition of sines and cosines of angles: $\left(\frac{x_1y_2+x_2y_2}{1+dx_1x_2y_1y_2}, \frac{y_1y_2-ax_1x_2}{1-dx_1x_2y_1y_2}\right)$. If F contains a square root of a but no square root of d, the denominators are never zero. I am not aware of cryptographic use of parameter choices that do not rule out division by zero; alternative formulas for these cases are not covered here. The identity element is $(0, 1)$ and scalar multiplication nP is defined as addition of P repeated n times.

A straightforward transcription of this specification to Coq is possible, but getting the relevant notations set up in the first place requires a fair bit of boilerplate. The only semantically significant difference comes from Coq defaulting to constructive logic: we need to specify as a precondition that it is possible to algorithmically decide whether two field elements are equal or not, while study of elliptic curves under the assumptions of classical mathematics treats this as an axiom.

The representation I chose (figure 5.1) enforces that all `points` are in fact on the curve by typing. However, as it would be unreasonable to expect the Coq type checker to verify that the addition formula produces valid points when checking `add : point -> point -> point`, a proof of point validity is included as an additional field, next to x and y. This way, every function computing a pair of field elements and returning a `point` needs to be accompanied with an explicit proof that the output is always on the curve. Of course, we do not actually compute with the proofs: Coq's default evaluation strategy does not materialize proofs when the overall expression has a simple type, extraction of OCaml or Haskell code ignores the proof fields, and so does our own synthesis of low-level code. Nevertheless, every time I (unintentionally) type `Eval vm_compute in P`

instead of `Eval vm_compute in (coordinates P)`, Coq happily exhausts tens of gigabytes of RAM trying to (amongst other things) enumerate all relevant cases of a proof originally given by induction.

The consequences of using expressive types in specifications reach far beyond evaluation. A major benefit is that algebraic laws can be proven in their natural form, for example `forall P Q R, P+(Q+R) = (P+Q)+R` instead of `forall P Q R, onCurve P -> onCurve Q -> onCurve R -> P+(Q+R) = (P+Q)+R` as one would use for simply typed points. This is particularly important when interfacing with pre-existing higher-order functions that expect an associative operation as an argument but have no support for separately threading through the fact that this operation preserves some property of the objects that it is operating on. For example, we were able to reuse the standard library implementation of binary exponentiation (and its correctness proof) as a stepping stone towards our own constant-time scalar multiplication procedure. On the other hand, putting an invariant in the types makes the code depend on the proofs that this invariant is preserved. Worse, mixing code and types without a plan has so far brought me only confusion and triggered limitations of standard library proof scripts. Nevertheless, we were able to find a set of conventions that seem to have kept us removed from the horror stories of dependently typed programming. In short: write code and proofs separately, wrap them together at the module interface. Every “public” function should as the very first (innermost) step separate the input values from the input invariants and as the very last (outermost) step package the output value with the output invariants.

```
Section TwistedEdwardsCurves.
  Context
    `{field: @Algebra.field
         F Feq Fzero Fone Fopp Fadd Fsub Fmul Finv Fdiv}
     {char_ge_3: @Ring.char_ge
         F Feq Fzero Fone Fopp Fadd Fsub Fmul 3}
     {Feq_dec: DecidableRel Feq}.

  Local Notation "0" := Fzero.
Local Notation "1" := Fone.
  Local Infix "+" := Fadd.      Local Infix "*" := Fmul.
  Local Infix "-" := Fsub.      Local Infix "/" := Fdiv.
  Local Notation "x ^ 2" := (x*x).
  Local Infix "=" := Feq : type_scope.
  Local Notation "a <> b" := (not (a = b)) : type_scope.

  Context {a d : F}
          {nonzero_a : a <> 0}
          {square_a : exists sqrt_a, sqrt_a^2 = a}
          {nonsquare_d : forall x, x^2 <> d}.

  Definition point : Type :=
    { xy : F*F
    | let '(x,y) := xy in a*x^2 + y^2 = 1+d*x^2*y^2 }.

  Definition coordinates (P:point) : F*F :=
    let (xy, xy_onCurve_proof) := P in xy.
  Program Definition zero : point := (0, 1).

  Program Definition add (P1 P2:point) : point :=
    match coordinates P1, coordinates P2 return F*F with
    | (x1, y1), (x2, y2)
      => (((x1*y2  +  y1*x2)/(1 + d*x1*x2*y1*y2)),
          ((y1*y2 - a*x1*x2)/(1 - d*x1*x2*y1*y2)))
    end.
  Next Obligation.
    destruct P1 as [[??]?], P2 as [[??]?];
      eapply Pre.onCurve_add; trivial.
  Qed.

  Fixpoint mul (n:nat) (P : point) : point :=
    match n with
    | O => zero
    | S n' => add P (mul n' P)
    end.
End TwistedEdwardsCurves.
```

Figure 5.1: The first code block is boilerplate, existing solely to assign the obvious notations to field operations. In principle, this could be automated using either typeclass inference or module functors but I consider superfluous 10 lines of simple code to be much more acceptable than involving one of the two hairiest features of the Coq system in the specifications.

The "set" notation `{ x : T | P x }` actually stands for a type with two fields where the type of the second field depends on the value of the first. Here, the first field contains the coordinates and the second field a proof that the coordinates are on the curve.

The `Program Definition` machinery accepts a superset of the code `Definition` accepts, creating proof obligations for missing details. Here, the `match` is marked to return `F*F`, but that value is then used as a `point`: the missing proof of the coordinates being on the curve is left as an obligation.

While `Program Definition` can be very convenient for composing code with invariants in the types, I find its behavior on `match` without explicit `return` type annotations annoying: it would create a proof of `(x1, y1) = coordinates P1`, pass it into the `match`, and then construct the output invariant proof inside the match rather than at the end of the function, complicating proofs.

Montgomery Curves

In almost all cases, a point of a Montgomery curve over a field F (of characteristic at least 3) with parameters a and $b \neq 0$ has coordinates (x, y) s.t. $by^2 = x^3 + ax^2 + x$. Similarly, almost all pairs of points can be added using the formula $k = (y_2 - y_1)/(x_2 - x_1)$; $(x_1, y_1) + (x_2, y_2) = \big(bk^2 - a - x_1 - x_2,\ (2x_1 + x_2 + a)k - bk^3 - y_1\big)$. But if this was it, we wouldn't need a separate subsection for defining Montgomery curves.

The above addition formula contains a division, and this time there is no reasonable way to engineer away the possibility of the denominator being zero if $x_1 = x_2$. Furthermore, we want Montgomery curve points to form a group, but no two-coordinate point acts as a zero for the addition. Based on the geometric interpretation of k as a slope, adding a point to itself is defined by instead setting k equal to the slope of a line tangent to the curve at that point. The remaining case $(x, y) + (x, -y)$ would have an infinite slope; these points are algebraically the opposites of each other and their sum is the zero element, denoted ∞. Including cases for adding ∞, we have a total of 6 cases in figure 5.2.

Short Weierstrass Curves

The definition of short Weierstrass curves (figure 5.3) is structured and motivated similarly to that of Montgomery curves, but the family of curves it encompasses is more general. With parameters a and b, the points are ∞ and (x, y) s.t. $y^2 = x^3 + ax + b$. Weierstrass curves do not necessarily have points that are their own opposites (i.e., $P + P = P - P = \infty$): $(x, y) = (x, -y)$ if $y = 0$, but $x^3 + ax + b$ may not have a solution in the underlying field. On the other hand, every Montgomery curve contains $(0, 0)$ (with $(0, 0) + (0, 0) = \infty$) and every Edwards curve contains $(0, -1)$ (with $(0, -1) + (0, -1) = (0, 1)$ as for angles). A similar

```
Local Notation "' ∞ '" := (inr tt).
Local Notation "' ∞ '" := unit : type_scope.
Local Notation "( x , y )" := (inl (pair x y)).

Context {a b : F} {b_nonzero : b <> 0}.
Definition point :=
  { P : F*F + ∞ | match P with
                  | (x, y) => b*y^2 = x^3 + a*x^2 + x
                  | ∞ => True
                  end }.
Definition coordinates (P:point) : (F*F + ∞) :=
  let (coords, pf) := P in coords.

Program Definition zero : point := ∞.

Program Definition add (P1 P2:point) : point :=
  match coordinates P1, coordinates P2 return F*F + ∞
with
  | (x1, y1), (x2, y2) =>
    if dec (x1 = x2)
    then if dec (y1 = - y2)
         then ∞
         else let k := (3*x1^2 + 2*a*x1 + 1)/(2*b*y1) in
              let x := b*k^2 - a - x1 - x1 in
              let y := (2*x1 + x1 + a)*k - b*k^3 - y1 in
              (x, y)
    else let k := (y2 - y1)/(x2-x1) in
         let x := b*k^2 - a - x1 - x2 in
         let y := (2*x1 + x2 + a)*k - b*k^3 - y1 in
         (x, y)
  | ∞, ∞ => ∞
  | ∞, _ => coordinates P2
  | _, ∞ => coordinates P1
  end.
Next Obligation. Proof. (* next section *) Qed.
```

Figure 5.2: The encoding of "either (x, y) or ∞" in Coq uses the type `sum A B` with constructors `inl (a:A) : sum A B` and `inr (b:B) : sum A B`, and the `unit` type with one constructor `tt`. We use notations $F \times F + \infty$ and ∞. The core definitions are given by case analysis on the `sum` type, even just to define whether a point is on the curve.

This code snippet defining Montgomery curves looks similar in length to that defining twisted Edwards curves only because I omitted the common field notation boilerplate and `mul` definition from this one to fit this one on one page. The increased complexity due to multiple cases shows up in all subsequent developments about Montgomery curves specifically, either as human effort in writing the proof or simply longer execution times of proof scripts that work case-by-case.

All considerations about richly typed representations described when defining twisted Edwards curves apply.

separation applies to points such that $4P = \infty$. This makes short Weierstrass curves strictly more general than Montgomery and Edwards curves and it can be shown that every Edwards or Montgomery curve over a field of characteristic at least 4 corresponds to a Weierstrass curve through a change of variables. As a consequence, a definition of Weierstrass curves is very useful to capture curves not covered by the previous definitions (e.g. curves

of odd order) but the optimized algorithms that apply to the more restrictive definitions may not be adaptable to the general case.

Figure 5.3: Again, I omit all boilerplate already explained in the previous definition.

```
Context {a b : F}.
Definition point := { P | match P with
                          | (x, y) => y^2 = x^3 + a*x + b
                          | ∞ => True
                          end }.
Definition coordinates (P:point) : F*F + ∞ :=
    let (coords, _) := P in coords.

Program Definition zero : point := ∞.

Program Definition add (P1 P2:point) : point :=
  match coordinates P1, coordinates P2 return F*F + ∞
with
  | (x1, y1), (x2, y2) =>
    if dec (x1 = x2)
    then
      if dec (y2 = - y1)
      then ∞
      else let k := (3*x1^2+a)/(2*y1) in
           let x3 := k^2 - x1 - x1 in
           let y3 := k*(x1 - x3) - y1 in
           (x3, y3)
    else let k := (y2 - y1)/(x2 - x1) in
         let x3 := k^2 - x1 - x2 in
         let y3 := k*(x1 - x3) - y1 in
         (x3, y3)
  | ∞, ∞ => ∞
  | ∞, _ => coordinates P2
  | _, ∞ => coordinates P1
  end.
Next Obligation. Proof. (* next section *) Qed.
```

5.3 Proving Basic Properties, Group Structure

The definitions in the previous section forward-referenced or outright omitted several important proofs that make the definitions make sense (and type check in Coq). The reason for this choice is rooted in the structure and ultimately the proposed procedure for auditing of the codebase: everyone with relevant domain knowledge should be able to check that the functionality of our library is specified correctly by

reading the files in `Spec/` and nothing else. Even though I chose to write richly typed specifications whose well-typedness depends on proofs of some basic properties, these proofs are completely irrelevant to what is being specified and should not clutter the specification. Even an adversarially crafted proof couldn't change the meaning of the specification when referenced: even though it is possible to mix proofs and code, Coq does not allow any information flow from proofs of `Prop`s to code.

Closure

As the definition of a group requires, the output of point addition and point negation need to be on the curve. The proof goals expressing this requirement are generated automatically by `Program Definition` when it is trying to get a definition to type-check. Each one is straightforward composition of the validity invariant declared with the `point` type and the operation being defined. See figure 5.4 for a goal and figure 5.5 for the proof.

```
match
  match coordinates P1 with
  | (x1, y1) =>
      match coordinates P2 with
      | (x2, y2) =>
          if dec (x1 = x2)
          then
           if dec (y1 = - y2)
           then ∞
           else (* omitted: doubling case ... *)
          else (* omitted: addition case ... *)
      | ∞ => coordinates P1
      end
  | ∞ => match coordinates P2 with
         | (_, _) => coordinates P2
         | ∞ => ∞
         end
  end
with
| (x, y) => b * y ^ 2 = x ^ 3 + a * x ^ 2 + x
| ∞%core => True
end
```

Figure 5.4: Proof obligation generated by `Program Definition` for definition `add` in figure 5.2. The outermost `match` construct originates from the definition of `point`; the definition of adding `P1` and `P2` is inlined into it. The two inner `match`es correspond to the `match` on two points in `add`, which is just convenient syntax for the form shown here.

Even though this expression looks like code, it returns a proposition. Looking at the `with` cases of the outermost match, we can see that (depending on the outcome of the addition) the proposition to be proven is either an equality or the trivial proposition `True`.

Figure 5.5: Proof of figure 5.4.

```
Proof. cbv [coordinates]; break_match; trivial; fsatz. Qed.
```

At this point, a reader who is actually trying to follow the reasoning might object that the five words in figure 5.5 do not constitute a proof by their standards, i.e., that I cheated. It is not up to me to criticize the reader's taste in proofs, but I would rather describe the above as honest trickery: the script shown does generate proofs for each case laid out in the auto-generated goal in figure 5.4, and each of these proofs is checked by the Coq kernel, which knows nothing about this thesis or even elliptic curves. However, I do not claim I would be able to explain this proof in front of a blackboard, or that reading this document would grant anyone else this power: Coq generated the goals, and Coq proved the goals, so may the Coq explain. Or as a mathematician would put it, these proofs are simply too boring. After all, the above proof script captures everything that one would remember about the proof even if they had seen it: each case proceeds by algebraic manipulation (`fsatz`) of the `coordinates` of the points being `match`-ed on (and `trivial` proves `True`).

The corresponding closure proof for twisted Edwards curves includes a non-trivial argument[6] to show that the denominators are nonzero (Montgomery and Edwards addition laws explicitly test for and handle zero denominators). Again, the Coq translation (figure 5.6) of the proof does not show any algebraic manipulation and is shorter than the pencil-and-paper version for $a = 1$ originally published in a conference paper. Further, while the paper version was optimized to better communicate understanding to humans and this one eschews readability in favor of a troubleshooting-oriented coding style, both have the same essential elements: explicit case analysis, explicit construction of $\sqrt{d}$, and an appeal to out-of-band algebraic manipulation to show that the latter is correct. Similar arguments are used inline for the optimized addition

6. Theorem 3.3, Daniel J. Bernstein and Tanja Lange. "Faster Addition and Doubling on Elliptic Curves". In: *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*. ASIACRYPT'07. Kuching, Malaysia: Springer-Verlag, 2007, pp. 29–50. ISBN: 3-540-76899-8, 978-3-540-76899-9. URL: https://eprint.iacr.org/2007/286.

formula proofs described in section 5.4, but I decided to explicitly state and prove just this step here in hope that it is instructive for translating algebraic proofs to Coq.

```
Lemma denominator_nonzero
  (a:F) (a_sq: exists sqrt_a, sqrt_a^2 = a) (a_nz: a<>0)
  (d:F) (d_nsq: forall sqrt_d, sqrt_d^2 <> d)
  (x1 y1:F) (P1ok: a*x1^2 + y1^2 = 1 + d*x1^2*y1^2)
  (x2 y2:F) (P2ok: a*x2^2 + y2^2 = 1 + d*x2^2*y2^2)
  : (d*x1*x2*y1*y2)^2 <> 1.
Proof.
  destruct a_sq as [sqrt_a];
    destruct (dec (sqrt_a*x2+y2 = 0));
    destruct (dec (sqrt_a*x2-y2 = 0));
    try match goal with [H: ?f (sqrt_a * x2) y2 <> 0 |-_]
      => pose proof
          (d_nsq ((f (sqrt_a * x1) (d * x1*x2*y1*y2*y1))
                 /(f (sqrt_a * x2) (y2)   *   x1 * y1)))
    end; fsatz.
Qed.
```

Figure 5.6: Proof that the denominators in figure 5.1 are never zero.

Here `match goal` with `?f` in the pattern binds to `f` the function of `sqrt_a * x2` and `y2` which hypothesis `H` claims to be nonzero – either `+` or `-`, depending on the outcome of the earlier case analysis. Either way, a square root of the non-square d is constructed, and `fsatz` derives contradiction.

In the case the `match goal` does not find suitable `H` and `f`, the `try` in front of it lets the proof script continue – both case analyses must have come out "`=0`", so $x_2 = y_2 = 0$, but that contradicts `P2ok` and `a_nz` by `fsatz`.

Associativity, Commutativity, Invesrses...

Proofs of other properties of elliptic curves required for them to be commutative groups follow the same overall pattern as the closure proofs required for type checking. In particular, the proof scripts only deal with the code structure of the elliptic curve and group definitions; all algebraic manipulation is delegated to the `fsatz` tactic.

More generally, I have tried to keep definition-specific reasoning as simple and syntax-directed as possible and separate non-trivial reasoning to tactics based on problem domain, rather than the lemma or definition that uses the tactic. Having a clear specification of what kinds of goals each tactic should solve, and what kinds of goals it is allowed leave behind is immediately useful to resolve

situations where a bug fix in a tactic breaks a proof that uses this tactic. Perhaps even more importantly, this enables modular reasoning about how much work it would be to prove one or another property.

Of course, this kind of modularity has its limits. In particular, it does not capture resource constraints on the execution of the proof script. A perfectly sound strategy for decomposing a goal into a finite number of known-to-be-solvable subgoals can still take unreasonably long and two proof steps that can be generated separately just within the available RAM can resulting in an out-of-memory error when combined. Nevertheless, decoupling proof methodology from the final proof generated by the script is unambiguously worth it, even if it means that the size and generation time of the resulting proof need to be estimated and controlled separately. Performance issues also further emphasize the need for predictable tactics: there is a huge difference between needing to adjust and re-run a proof script every 10 minutes and writing it once, even if it needs to be run overnight.

Figure 5.7: A proof that every Weierstrass curve forms a group.

The proof script follows the usual pattern of opening up definitions, simplifying the goal in ad-hoc ways using a `repeat match …progress`, and then passing each subgoal through a sequence of tactics that each either solve it or leave it unchanged.

`contradiction` from the Coq standard library solves all goals with structurally nonsensical hypotheses such as `False` or `∞ = (_, _)`.

```
Program Definition opp (P:point) : point :=
  match coordinates P return F*F+ ∞ with
  | (x1, y1) => (x1, - y1)
  | ∞ => ∞
  end.
Next Obligation.
  cbv [coordinates]; break_match trivial; fsatz.
Qed.

Lemma commutative_group (discr_nz: 4*a^3 + 27*b^2 <> 0)
  : Algebra.commutative_group
      (eq:=eq) (op:=add) (id:=zero) (inv:=opp).
Proof.
  cbv [opp eq zero add coordinates];
    repeat match goal with
    | [H:_ /\ _ |- _] => destruct H
    | _ => progress intro
    | _ => progress Algebra.split
    | _ => progress break_match
    | _ => progress break_match_hyps
    end; try contradiction; trivial; fsatz.
Qed.
```

A Coq proof that every short Weierstrass curve forms a group is shown in figure 5.7; the one for Edwards curves is even simpler due to the lack of case analysis. While a similar direct proof would presumably work for Montgomery curves, I instead prove that each Montgomery curve is isomorphic to some Weierstrass curve using the very same tactic machinery.

5.4 Optimized Representations

To allow for fast computations, I implemented and verified inversion-free addition formulas for each family of curves. While transcribing the formulas themselves was trivial, coming up with precise specifications involved a surprising amount of guesswork – the formulas themselves are much better known than the invariants they maintain about the values they operate on.

For example, XYZT coordinates for Edwards curves represent the x and y coordinates of each point as fractions X/Z and Y/Z, carrying around precomputed T=XY/Z. In this case, the complete invariant is rather obvious: the curve equation needs to hold on $(\mathsf{X/Z}, \mathsf{Y/Z})$, T=XY/Z needs to hold, and Z needs to be nonzero. The natural way to encode this in Coq is the same as for the specification of the elliptic curve: a 4-tuple of field elements carrying an invariant proof. The correctness of operations on this representation is easily stated as an isomorphism: operating on the efficient representation and then converting to the spec representation should be the same as first converting to the spec representation and operating there. Again, all proof obligations are naturally decomposed into field equalities which can be handled by `fsatz`.

The optimized single-case addition formulas for Montgomery and Weierstrass curves are significantly trickier to reason about because they are not complete: for some inputs, the formula does not produce the correct answer.

Furthermore, the Montgomery ladderstep formula[7] is used for x-coordinate scalar multiplication without checking that the input point is on the curve at all, this obviously ruling out the possibility to encode that fact in the type. Furthermore, computing on the x coordinate alone prevents us from converting from the optimized representation back to the specification representation, forcing the use of a custom equivalence relation throughout proofs. To complicate the situation even further, the fraction $x/0$ is allowed for nonzero x; it represents the point ∞.

7. Daniel J. Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *Public Key Cryptography - PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26*. Springer-Verlag, Feb. 9, 2006. URL: http://cr.yp.to/papers.html#curve25519 (visited on 08/14/2016), appendix B.

```
Definition projective (P:F*F) : Prop :=
  if dec (snd P = 0) then fst P <> 0 else True.
Definition peq(P Q:F*F) := fst P * snd Q = fst Q * snd P.

Lemma to_xz_add (x1:F) (xz x'z':F*F)
  (Hxz:projective xz) (Hz'z':projective x'z')
  (Q Q':point)
  (HQ:peq xz (to_xz Q)) (HQ':peq x'z' (to_xz Q'))
  (difference_correct:match coordinates (Q - Q') with
                      | ∞ => False
                      | (x,y) => x = x1 /\ x1 <> 0
                      end)
  :  peq (to_xz (Q+Q )) (fst (xzladderstep x1 xz x'z'))
  /\ peq (to_xz (Q+Q')) (snd (xzladderstep x1 xz x'z')).
```

Figure 5.8: Statement of main lemma about `xzladderstep`, which efficiently computes the x coordinates of $Q + Q$ and $Q + Q'$ given the x coordinates of $Q - Q', Q, Q'$. The last two inputs and the outputs use a projective representation as fraction x/z; the difference input implicitly uses $z = 1$. `+` and `-` refer to the specification of Montgomery curves (figure 5.2).

The multitude of preconditions is not noteworthy – in other cases, the same information was required as a part of the `point` type. The `difference_correct` precondition, however, encodes the inherent limitation of this formula: adding a point to itself does not work, as doesn't adding points whose difference is $(0, 0)$. To safely use this formula, we separately prove that the difference of two points is preserved when `xzladderstep` adds Q to both of them. This is proven by reasoning about the elliptic curve as a group, *without* breaking each point into coordinates. The caller never starts with $Q - Q' \in \{\infty, (0, 0)\}$.

I was also able to verify that the single-case addition formula[8] for Weierstrass curves of odd order behaves as expected. Again, the precondition looks rather gnarly when written in terms of coordinates of the points being added.

8. Joost Renes, Craig Costello, and Lejla Batina. "Complete addition formulas for prime order elliptic curves". In: *Proceedings of the 35th Annual International Conference on Advances in Cryptology, New York, USA*. Springer-Verlag, May 8, 2016. URL: https://eprint.iacr.org/2015/1060.pdf.

This condition can also be specified purely in terms of group operations on the elliptic curve, without revealing any internal details (see figure 5.9).

```
Definition projpoint := { P : F*F*F | let '(X,Y,Z):= P in
  Y^2*Z = X^3 + a*X*Z^2 + b*Z^3 /\ (Z = 0 -> Y <> 0)}.

Definition not_exceptional_y (P Q:projpoint) :=
  match coordinates (to_affine P - to_affine Q) with
  | ∞ => True
  | (_, y) => y <> 0
  end.
Definition not_exceptional P Q :=
  let p := to_affine P in
  let q := to_affine Q in
  (W.eq (p+p) (q+q) -> W.eq p q).
```

Figure 5.9: The projective representation invariant and preconditions for the single-case addition formula for Weierstrass curves. While the precondition is again stated as a `match` on the difference of inputs, the requirement is different from before. Again `+` and `-` refer to the affine representation (figure 5.3, figure 5.7).

Obviously, the latter form is more suitable for further reasoning, even if it is concluded with an argument how the group structure of the particular elliptic curve does not include any point that could trigger the exceptional cases. The proof of equivalence between these two statements consists of two key lemmas: all exceptional cases have $2P = 2Q$, and $P + P$ is never exceptional.

Overall, the main challenge in verifying optimized representations of elliptic curves is getting the invariants and preconditions exactly right – while the formulas presented in publications can often be translated to Coq verbatim, the preconditions are given in prose, or as a part of the proof, or omitted entirely.

5.5 EdDSA

To check that the definitions of elliptic curves are right, I coded up a specification of the EdDSA[9] digital signature scheme. The definition in figure 5.10 is parametrized over the underlying elliptic curve, hash function, base point and other dependencies. The original paper specification inlines the definition of twisted Edwards curves, and I only instantiate it with Edwards curves, but using parameters in a

9. Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *EdDSA for more curves*. 2015. URL: http://cryptojedi.org/papers/#eddsa.

definition is a good way to separate out proofs about different abstraction layers.

```
Program Definition curveKey (sk:word b) : nat :=
  let x := wfirstn n (H sk) in
  let x := x - (x mod (2^c)) in
           setbit x n.
Definition prngKey (sk:word b) : word b :=
  split2 b b (H sk).
Definition public (sk:word b) : word b :=
  Eenc (curveKey sk*B).

Program Definition sign (A_:word b) sk {n}(m:word n):=
  let r : nat := H (prngKey sk ++ m) in
  let R : E := r * B in
  let s : nat := curveKey sk in
  let S : F l := F.nat_mod l (r + H(Eenc R++A_++m)*s) in
      Eenc R ++ Senc S.

Definition valid {n} (m : Word.word n) pk sig : Prop :=
  exists A S R,
    Eenc A = pk /\ Eenc R ++ Senc S = sig /\
    F.to_nat S * B == R + (H(Eenc R++Eenc A++m) mod l)*A.
```

Figure 5.10: Specification of EdDSA, without the parameter declarations.

+, *, and == refer to point addition, scalar-point multiplication, and equality on the underlying elliptic curve. `Eenc` and `Senc` are encodings of points and scalars as b-bit strings given as parameters to the specification.

The `word b` represents bit vectors of length b, `++` is concatenation and `split2 b b` takes the last b bits. H outputs `word (b+b)` and takes as input a `word n` for any n.

The specification does not prescribe a verification algorithm, it just defines which signatures are considered valid. To derive an implementation, each optimization is verified individually: for example,the equation on the last line can be checked by solving it for R and encoding it, without ever decoding the R received as a part of the signature. We instantiated this specification with the parameters for ed25519 and extracted an implementation that uses binary exponentiation, which passed all tests on run #2 – after fixing encoding order in the spec.

Chapter 6

Automated Proofs of Field Equations

To complete the proofs in the high-level elliptic curve library described in the previous chapter, I created a custom Coq tactic `fsatz` (building on the standard library `nsatz`). This chapter will given an overview of its functionality, design, and implementation.

The elliptic curve formulas can contain addition, multiplication, division, and powers of coordinates. The specifications can be reasonably translated into this formulation if they do not already fit – for example, two points are equal if and only if their coordinates are equal. Furthermore, the correctness of the addition formulas depends only loosely on the properties of the field that the coordinates are a member of: it is often required that $1 + 1 \neq 0$ (or similar), but there is never need to explicitly examine the behavior of the field operations on some variables. However, since division is undefined if the denominator is zero, the system will need to be able to handle inequalities. Based on this information, I decided to build a proof script to handle arbitrary implications between rational equations and inequalities over finite fields[1]. More formally: the givens and goal must be of the form $e = e$ or $e \neq e$ where $e ::= 1 \,|\, x \,|\, e + e \,|\, -e \,|\, e \cdot e \,|\, e^{-1}$ for

1. Here is an example equation that one might find in a high school textbook. It is proved completely automatically as a part of the test suite of the proof script described in this chapter: given $\frac{9}{x^2+x-2} = \frac{3}{x+2} + 7\frac{1}{x-1}$ and appropriate assumptions about the coefficients and denominators being nonzero, we have $x = -\frac{1}{5}$.

(potentially multiple) symbolic variables x.

6.1 The `nsatz` Tactic

The Coq standard library contains a tactic `nsatz`[2] (named after Hilbert's Nullstensatz) for solving implications between polynomial equalities. It is my impression that `nsatz` is little-known among Coq users, which I think does not do justice to its usefulness. For example, a single `nsatz` invocation automatically completes a notoriously long proof by algebraic manipulation in figure 6.1.

This section will give an overview of the capabilities of `nsatz`, both to give the reader the background necessary for understanding the following sections and to more generally explain where else nsatz might be of use.

The prerequisite for using `nsatz` is that the algebraic structure in the goal must be an integral domain, that is it must be a ring where products of nonzero elements are nonzero. This includes the integers, the rationals, the reals, and all fields. Unlike the better-known `ring` tactic, `nsatz` does not work on semirings such as `nat`. For `nsatz` to work, the goal must be an equality. The tactic understands addition, subtraction, multiplication, and powers. Outputs of unrecognized operations are treated as variables: even though `nsatz` can prove $\frac{a}{b} + c = c + \frac{a}{b}$, it is just because `nsatz` can prove $x + c = c + x$.

There are two potential downsides to using `nsatz`. A successful invocation of `nsatz` does not necessarily solve the goal. However, the residual goal will by default not mention any variables and can thus be checked by computation on all concrete algebraic structures. More significantly, `nsatz` provides no performance guarantees. While most goals are solved within a fraction of a second, I encountered cases where a single succeeding `nsatz` call takes several minutes. In some cases, the running time fluctuated heavily with minor changes in the goal such as

2. Loïc Pottier. "Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics". In: *CoRR* abs/1007.3615 (2010). URL: `http://arxiv.org/abs/1007.3615`.

```
Require Import Reals Nsatz. Local Open Scope R_scope.
Local Notation "x ^ 2" := (x*x)
(only parsing, at level 30).
Local Notation "x ^ 3" := (x*x*x) (only parsing, at level 30).

Lemma weierstrass_associativity_main_case (* many givens: *)
 (a b x1 y1 x2 y2 x4 y4 :R)   (A :y1^2-x1^3-a*x1-b = 0)
 (B :y2^2-x2^3-a*x2-b = 0)    (C :y4^2-x4^3-a*x4-b = 0)
 (i3:R) (Hi3:i3*(x2-x1)=1)  (k3:R) (Hk3:k3=(y2-y1)*i3)
 (x3:R) (Hx3:x3=k3^2-x1-x2) (y3:R) (Hy3:y3=k3*(x1-x3)-y1)
 (i7:R) (Hi7:i7*(x4-x3)=1)  (k7:R) (Hk7:k7=(y4-y3)*i7)
 (x7:R) (Hx7:x7=k7^2-x3-x4) (y7:R) (Hy7:y7=k7*(x3-x7)-y3)
 (i6:R) (Hi6:i6*(x4-x2)=1)  (k6:R) (Hk6:k6=(y4-y2)*i6)
 (x6:R) (Hx6:x6=k6^2-x2-x4) (y6:R) (Hy6:y6=k6*(x2-x6)-y2)
 (i9:R) (Hi9:i9*(x6-x1)=1)  (k9:R) (Hk9:k9=(y6-y1)*i9)
 (x9:R) (Hx9:x9=k9^2-x1-x6) (y9:R) (Hy9:y9=k9*(x1-x9)-y1)
:x9 = x7 /\ y9 = y7. (* the claim *)
Proof. split; nsatz. Qed.
```

Figure 6.1: Example goal solved by `nsatz`. The lemma states that addition of two different nonzero non-opposite points on an elliptic curve in Weierstrass form is associative. This lemma is notoriously difficult. I am not aware of any direct proof that would be feasible to check by hand. Indirect proofs rely on significantly more complicated theory (for example, the Picard group of divisors) and direct proofs rely on modern computer-algebra systems.

re-ordering the givens. Worse, failing goals can take much longer – on several occasions I ran `nsatz` overnight and found it still running when I got back to work, only to discover that the lemma it was trying to prove was missing a hypothesis and thus probably untrue. It should be noted that these limitations are an integral part of the algorithm, rather than specific to Coq: I have had similar experiences with the implementations of the same decision procedure provided in the Sage and Magma computer algebra systems.

Both limitations can be understood by thinking about the high-level strategy behind `nsatz`: to show $A = 0 \to B = 0 \to P = 0$, it is sufficient to find a, b, c, r such that $aA + bB = cP^r$. The residual goal left behind by `nsatz` is precisely $c \neq 0$. The algorithm proceeds by trying increasing integer values of r until P can be reduced to 0 by a Gröbner basis equivalent to (A, B). The coefficients of the hypotheses (here a and b) are allowed to be arbitrary polynomials in the relevant variables, but c is required to be an integer by default. It is possible to relax this constraint and speed up the search by marking some variables as *parameter*s in the `nsatz` invocation, but then proving $c \neq 0$ is no longer trivial (I have not used this option).

6.2 Eliminating Divisions

While it is seemingly trivial to eliminate division by bringing all fractions to a common denominator, this approach can significantly increase the size of the goal. Somebody else working on this project implemented a common-denominator-based approach to dealing with division and attempted to prove that the standard Weierstrass addition procedure is associative but instead achieved a stack overflow in the `nsatz` polynomial implication prover. The original three-line goal with fractions in it had expanded to a 750-line polynomial equation. While the stack overflow has since been fixed, "the computation still takes ages, because it computes a HUGE polynomial"[3].

3. https://coq.inria.fr/bugs/show_bug.cgi?id=5085

It is better to introduce new variables for the inverses of the denominators and specify their status as an inverse using an addition equation. As the changes to the original equation are purely local, its size does not blow up. For example, the goal $a = b \to x \neq 0 \to \qquad a/x = b/x$ is transformed to $a = b \to x \neq 0 \to ix = 1 \to ai = bi$, which is solved by `nsatz`. Importantly, the transformation is reversible: any field element that gives 1 when multiplied by x must be the unique inverse of x. However, after transforming explicit inverses to equations specifying inverses, all information about inverses is in a format accepted by `nsatz`.

Figure 6.2: Unlike common denominator finding, the turning inverses into equations is a breeze to implement.

```
Ltac divisions_to_inverses :=
  rewrite ?(field_div_definition) in *.

Ltac inverses_to_conditional_equations :=
  repeat match goal with
  | |- context[inv ?d] =>
    unique pose proof constr:(right_multiplicative_inverse d)
  | H: context[inv ?d] |- _ =>
    unique pose proof constr:(right_multiplicative_inverse d)
  end.
```

6.3 Eliminating Inequalities

First, let's examine two cases where a goal with inequalities can be transformed into an equivalent goal stated purely in terms of equalities:

- When trying to prove $e_1 \neq e_2$, it is equivalent to prove $e_1 = e_2 \to 0 = 1$.
- When trying to prove $a_1 \neq a_2 \to e_1 \neq e_2$, it is equivalent to prove $e_1 = e_2 \to a_1 = a_2$.

In both cases, hypotheses stating that two expressions are equal carry over transparently. However, slightly more care is needed to make use of multiple inequality hypotheses. Naively, we tried to merge inequality hypotheses by multiplying them: $b \neq 0 \to a \neq 0 \to e_1 \neq e_2$ would become $ab \neq 0 \to e_1 \neq e_2$. However, proving the goals $e_1 = e_2 \to ab = 0$ derived through this method was very slow – we did not investigate the precise reasons for this, but since the products of all nonzero-denominator hypotheses were very large in several cases, the slowness is rather unsurprising.

Instead, the fact that an expression is nonzero can be encoded as a polynomial equation using inverses: $x \neq 0$ iff there exists i such that $xi = 1$, and the goal $bi = 1 \to aj = 1 \to e_1 \neq e_2$ can be solved using the first simple case shown above[4]. In our experience, the polynomial equality goals achieved through this mechanism can be solved without noticeable wait times even in cases where a "true by inspection" goal transformed using the previous method took several minutes. More importantly, the second method manages to make use of the inequality hypothesis even if the goal itself is not an inequality: for example, $x^2 = y^2 \to x \neq -y \to x = y$ is proved automatically, even though the first version of inequality elimination would not have applied.

4. While my implementation relies on field structure to introduce the inverse, this method could be generalized to algebraic domains that are not fields by allowing fractional inverses. Doing so is equivalent to injecting the goal and all hypotheses to the "field of quotients" (also known as "field of partial fractions") of the original algebraic structure.

Eliminating Disjunctions

The basic intuition behind the first attempt at eliminating inequalities is instead useful for dealing with disjunctions in goals or hypotheses. In particular, $x = 0 \vee y = 0$ iff $xy = 0$. Adding this rule is sufficient to automatically prove $x^2 = y^2 \to x = y \vee x = -y$.

6.4 Multiple Inequalities

If a goal contains multiple denominators, all of them may need to be proven to be nonzero in order to prove the goal itself. The following is completely non-obvious: the nonzero-denominator proofs are not independent of each other. The naive algorithm "prove that each denominator is nonzero, one at a time" fails at the following example:

1. $x + y = 1$ (so $x \neq 0 \vee \neq 0$)
2. $y/x = x/x$ (so $x \neq 0 \to y = 1$)
3. $z/y = 1/y$ (so $y \neq 0 \to z = 1$)

These hypotheses do not imply that $x \neq 0$ ($z = y = 1$ is a satisfying assignment). Similarly, trying to prove $y \neq 0$ from the first hypothesis alone would not be successful: without considering the second hypothesis, it might very well be the case that $x = 1$ and $y = 0$. However, if x is nonzero then the second hypothesis would force $y = 1$, so y cannot be zero. Importantly, this can come up if the original goal did not involve y directly, for example when trying to prove $z = 1$.

The fix is simple: `fsatz` should consider all other hypotheses when trying to prove that a denominator is nonzero, including those that have denominators which might need to be proven nonzero in turn (figure 6.3. While in theory this introduces yet another way `fsatz` can take non-polynomial time, in practice the slowdown was immeasurably small, and was in fact offset by the

compilation time saved due to the removal of sub-lemmas working around this issue from two different elliptic developments.

```
Ltac forward_nonzero solver_tac :=
  repeat match goal with
  | [H: (?x <> zero) -> _ |- _ ]
    => let H' := fresh in
       assert (H' : (x <> zero)) by
         (clear_hypotheses_with_nonzero_requirements;
            solver_tac);
       specialize (H H')
  | [H: (?x <> zero) -> _ |- _ ] (* <-- new case *)
    => let H' := fresh in
       assert (H' : (x <> zero)) by
         (clear H; solver_tac);
       specialize (H H')
  end.
```

Figure 6.3: Implementation of the fix. The old behavior is still tried speculatively because it is usually faster; if the first case of the `match` fails the second one is tried automatically.

While it was relatively straightforward to work around it in both cases, and the slowdown introduced by the workarounds was tolerable, I would like to argue that the recursive `fsatz` is much better than the early non-recursive version. The difference is in what can be concluded from a failure to prove a goal using `fsatz`. For example, if `omega`, the linear arithmetic tactic in Coq, fails to prove a goal, then the linear hypotheses must insufficient to reach the conclusion, that is, if all non-linear operations and hypotheses were replaced with arbitrary variables, the goal wouldn't be true. A slightly weaker fact is true when the correct implementation of `fsatz` fails to solve a goal: the goal cannot be proven by algebraically combining the hypotheses. The non-recursive `fsatz`, does not have this property. Worse, the relation "`fsatz` can prove A from B" wouldn't even be transitive, so the user would need to think whether stating an intermediate lemma and proving it by `fsatz` would solve the goal. For example, the lemma in figure 6.4 could be proven by `assert (d*y^2 <> a) by fsatz; fsatz`. Performance and simplicity of proof scripts are valuable, but predictability of success and interpretability of failure are much more important.

Figure 6.4: A proof about Edwards curve point decompression that would require an intermediate lemma if `fsatz` didn't solve nonzero-denominator goals recursively.

```
Lemma edwards_solve_for_x2 (a sqrt_a d x y : F)
  (Hc : a * x^2 + y^2 = 1 + d * x^2 * y^2)
  (Ha : sqrt_a^2 = a) (Hd : (sqrt_a / y)^2 <> d)
: x^2 = (y^2 - 1) / (d * y^2 - a).
Proof. fsatz. Qed.
```

6.5 Fields of Unknown But Large Characteristic

As mentioned earlier, the correctness of most elliptic curve formulas relies on assumptions about the field characteristic. In particular, the theory of elliptic curves is significantly different in binary fields, that is when $1 + 1 = 0$, or "characteristic 2". Yet it is desirable to prove the correctness of the formulas once, quantifying over all fields of suitable characteristic.

Requirements on the field characteristic show up in our proofs as explicit side conditions generated by the `nsatz` tactic. Indeed, `nsatz` works on $x = 0 \to y = 0 \to e = 0$ by finding a constant c and polynomials a, b such that $ce = ax + by = 0$, leaving it up to the caller of the tactic to show that $c \neq 0$. While proving this side condition for a specific c by hand is trivial, doing so would have been prohibitively time-consuming – for example, the proof that Weierstrass curve addition forms a group involves around 200 cases, all of which are otherwise handled automatically. Figure 6.5 shows the core of the automated reflective procedure I wrote to solve these side conditions.

```
Ltac reify x :=
 match x with
 |one => constr:(Coef_one)
 |opp ?a =>
  let a' := reify a in
  constr:(Coef_opp a')
 |add ?a ?b =>
  let a' := reify a in
  let b' := reify b in
  constr:(Coef_add a' b')
 |mul ?a ?b =>
  let a' := reify a in
  let b' := reify b in
  constr:(Coef_mul a' b')
 end.
```

```
Fixpoint denote (c:coef):R:=
match c with
|Coef_one => one
|Coef_opp  => opp (denote c)
|Coef_add c1 c2 =>
  add (denote c1) (denote c2)
|Coef_mul c1 c2 =>
  mul (denote c1) (denote c2)
end.
```

```
Fixpoint is_nonzero (c:coef) : bool :=
  match c with
  | Coef_one => true
  | Coef_opp c => is_nonzero c
  | Coef_mul c1 c2=> andb (is_nonzero c1) (is_nonzero c2)
  | _ => is_constant_nonzero (CtoZ c) (* factoring *)
  end.

Lemma is_nonzero_correct' c (checked:is_nonzero c = true)
 : denote c <> zero.
Proof.
  induction c;
  repeat match goal with
  | [ H : _ |- _ ]
   => progress rewrite Bool.andb_true_iff in H;destruct H
  | [ H : _ |- _ ]
   => progress apply is_constant_nonzero_correct in H
  | _ => progress (change (denote (Coef_one))
                     with (of_Z 1) in * )
  | _ => progress (change (denote (Coef_opp c))
                     with (opp (denote c)) in * )
  | _ => progress (change (denote (Coef_mul c1 c2))
                     with (denote c1 * denote c2) in * )
  | _ => progress (change (is_nonzero (Coef_mul c1 c2))
             with (is_nonzero c1 && is_nonzero c2) in * )
  | |- (opp _ <> zero)
   => progress rewrite opp_zero_iff
  | |- (_*_<>zero)
   => eapply nonzero_product_iff_nonzero_factor
  | _ => solve [eauto
               using is_constant_nonzero_correct, le_1_l]
  | _ => progress rewrite <-CtoZ_correct
  end.
Qed.
```

Figure 6.5: The `reify` tactic script constructs an explicit syntax tree of $c ::= 1 \mid c \cdot c \mid -c \mid c + c$ by inspecting the goal, after which a wrapper tactic then calls the function `is_nonzero` to determine whether the expression (now referenced as the `denote` of the explicit syntax tree) is nonzero under the assumed characteristic.

The cases for 1, multiplication and negation are trivial: these operations cant turn a nonzero constant into a zero constant. However, addition of two values smaller than the field characteristic may produce a value that is not smaller than the field characteristic. To prove that the coefficient is nonzero for a specific characteristic K, it would be sufficient to reduce it modulo K and check that the result is nonzero. Without knowing the value of K, but assuming that $K > C$, it is still possible to show that a constant $c > C$ is nonzero by factoring c and checking that all factors are $\leq C$. Conveniently, this can be done without proving the correctness of the factoring algorithm – my code simply computes the factors and then checks that multiplying them indeed returns the original coefficient.

The choice to use a verified reflective procedure instead of a tactic script was motivated by ease of coding, not performance concerns.

6.6 Independent Use

`fsatz`, the tactic I created for solving implications between field equalities and inequalities, does not depend on any theory specific to elliptic curves or cryptography. The dependencies of the file implementing it are contained in two subdirectories (for algebra definitions and general tactics), and the properties required of a field are designed so that they can be satisfied by a field implemented without any knowledge of these definitions.

Anecdotally, I can report that the following workflow for manipulating arbitrary algebraic equations is distraction-free and seamless enough that the loss of efficiency is outweighed by not having to check for (and correct) errors:

1. Enter the known equations as hypotheses for any goal: `Lemma wip x y (H:x^2 + y*x + 2=0) : False`
2. `assert (` *...derived equality ...* `) by fsatz.`, for example `assert (-x^2 = 2 + x*y) by fsatz.`
3. Edit (or copy) and re-execute the last line as desired.
4. Replace the original goal `False` with the derived equation. All intermediate steps can be removed.

6.7 Future (Grunt) Work

I cannot but prescribe a single action item towards improving the field algebra automation: fix bugs in the underlying `nsatz` tactic. While perfectly usable for "sane" goals written by a human, the unverified computation that tries to construct a proof certificate produces stack overflows in corner-cases that can arise from goals left behind by `fsatz`. For example, Coq bug #5359 describes a situation where *removing* a hypothesis makes `nsatz` able to solve a goal where it previously terminated due to stack overflow[5].

5. `https://coq.inria.fr/bugs/show_bug.cgi?id=5359`

Chapter 7

Deployment Considerations

7.1 Avoiding Historical Bugs

This section will review common issues with correctness of implementations of arithmetic-based cryptography and recount which ones are ruled out by the correctness proofs described in this thesis. The guarantees provided by other verification efforts described in chapter 2 will be highlighted if different.

I analyzed deployed cryptographic software for a sample of functional-correctness bugs specific to the functions being implemented. This means that generic issues such as memory mismanagement and side channels were excluded. On the other hand, I included bugs that could reasonably be attributed either to the cryptographic security of the protocol as implemented, or a failure to implement a known secure protocol.

The sample in table 7.1 includes the first 25 bugs I foundthat fit the specified criteria. I did not intentionally exclude bugs for any other reason. A more thorough search would inevitably uncover more bugs.

We observed that while the mistakes were often “small” in the sense that the difference between the original and corrected versions was minimal (in one case, a single character), understanding why one is correct and the other

is not requires significant contextual information, sometimes across multiple abstraction layers. In sharp contrast with the top-level specification, it is nontrivial to write the specification of a subroutine that captures all required behaviors and yet allows for important optimizations.

Arbitrary-Precision Arithmetic Bugs

Of the 25 bugs, 19 had to do with low-level multi-precision arithmetic. In this category, it is nontrivial to distinguish between design errors (code correctly implements the programmer's flawed understanding) and coding mistakes (typos, missed low-level details) based on the code itself, so I referred to relevant bug-tracker discussion if available. Similarly, it is extremely difficult to estimate the security impact of these bugs: Brumley, Barbosa, Page, and Vercauteren demonstrate[1] a sophisticated exploit against OpenSSL bug 1593, and Bernstein and Schwabe estimate[2] that a well-equipped attacker would be able to exploit OpenSSL CVE-2015-3193, but we do not know about all of the bugs. A detailed analysis of exploitability is outside the scope of this project, and I choose not to speculate.

Here are three example bugs for which an explanation was available:

- The TweetNaCl paper[3] describes a typo in `ed25519-amd64-64-24k`: `r1 += 0 + carry` should have been `r2 += 0 + carry` instead. Authors noted that the issue would not have been caught by random tests.

- OpenSSL issue 1593 was traced back to confusion between the postconditions of exact division with remainder and an operation like our `split` (section 3.6) that produces a q and r s.t. $x = qm + r$, but does not guarantee that r is the smallest possible. The

1. Billy Brumley, Manuel B. M. Barbosa, Daniel Page, and Frederik R G Vercauteren. "Practical realisation and elimination of an ECC-related software bug attack". In: (2011). URL: https://eprint.iacr.org/2011/633.pdf.

2. Daniel J. Bernstein and Peter Schwabe. *gfverif*. Jan. 1, 2016. URL: http://gfverif.cryptojedi.org/.

3. Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "TweetNaCl: A crypto library in 100 tweets". In: *Progress in Cryptology – LATINCRYPT 2014*. Ed. by Diego Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2015, pp. 64–83. URL: http://cryptojedi.org/papers/#tweetnacl.

Table 7.1: A sample of crypto-specific implementation bugs.

Reference	Specification	Implementation	Defect
openssl#3607	sqrmod p256	64-bit Montgomery form, AMD64	limb overflow
go#13515	expmod	uintptr-sized Montgomery form, Go	carry handling
NaCl ed25519	F25519 mul, square	64-bit pseudo-Mersenne, AMD64	carry handling
openssl#0c687d7e	poly1305	32-bit pseudo-Mersenne; x86, ARM	bad truncation
openssl#ef5c9b11	expmod	64-bit Montgomery form, AMD64	carry handling
nettle#09e3ce4d	mod secp-256r1		carry handling
socat#7	DH in Z_p^*	irrelevant	non-prime p
invalid-curve	NIST ECDH	irrelevant	not onCurve
donna#8edc799f	F25519 output	32-bit pseudo-Mersenne, C	non-canonical
CVE-2006-4339	RSA-PKCS-1 check	irrelevant	padding check
CVE-2014-3570	bignum squaring		
ic#237002094	p256 Barrett	only one cond. subtraction	unkown if ok
openssl#1593	mod P384	carry handling	exploitable
go#fa09811d	poly1305 reduction	AMD64 asm, -=3	found quickly
jose-adobe	ECDH-ES	5 libraries	not onCurve
tweetnacl-m[15]	F25519 output	bit-twiddly C	bounds? typo?
tweetnacl-U32	irrelevant	bit-twiddly C	sizeof(long) != 4
CVE-2017-3732	x^2 mod m	Montgomery form, AMD64 asm	carry, exploitable
openssl#c2633b8f	a + b mod p256	Montgomery form, AMD64 asm	non-canonical
openssl#59dfcabf	Weier. Jacobian	Montgomery form, AMD64 and C	∞ confusion
openssl#a970db05	poly1305	Lazy reduction in x86 asm	lost bit 59
openssl#6825d74b	poly1305	AVX2 addition and reduction	bounds?
openssl#74acf42c	poly1305	multiple implementations	carry handling
ed25519.py	ed25519 reference	accepts signatures others reject	missing h mod l
CryptoNote bug	anti-double-spending	additive curve25519 point	need order(P) = l

probability of a random test triggering this bug was bounded to $10 \cdot 2^{-29}$.

- One of the two bugs uncovered in OpenSSL issue 3607 was summarized by its author as “Got math wrong :-(”, which I think referred to an erroneous manual execution of the range analysis algorithm described in section 4.3. The discussion was concluded when the patched version was found to be “good for ~6B random tests” and the reviewer saw that “there aren’t any low-hanging bugs left.”

Higher-Level Bugs

While field arithmetic indeed accounts for the nastiest crypto bugs I have seen, confusion about operations with

elliptic curve points is not obviously any less dangerous. In particular, I do not agree with the notion that elliptic-curve-level code is inherently simpler or less error-prone.

Jager, Schwenk, and Somorovsky demonstrate[4] how, because an elliptic-curve point-decoding function failed to establish that the point indeed lies on the elliptic curve, devastating remote attacks were enabled against TLS/HTTPS security implemented by the Oracle Java and Bouncy Castle libraries. Years later, essentially the same issue was re-discovered in 5 libraries implementing JSON Web Encryption standard.

When trying to prove equivalence between the naive and optimized formulas for ed25519 signature verification, I discovered a discrepancy between the specification (and the Python reference implementation) of ed25519 and all serious implementations I tested (C "ref", ed25519-donna, ed25519-amd64-51-30k). In particular, a malicious actor would be able to create a signature that is considered good by any implementation that follows the specification to the letter but considered invalid by the optimized implementation. The difference comes from the fact that the point $(h \bmod l)P$ is equal to hP only if P has order l, but Curve25519 also contains points of other orders, even though a good signer never generates them. The optimization is sound in the sense that it makes no new signatures valid, and it significantly simplifies implementation. I changed our specification to match the current practice. However, in some settings it is critical that all parties in a protocol agree precisely on which signatures are valid and which are not (e.g., a disagreement might cause a crypto-currency to fork). The latest IETF draft[5], due to an independent fix, contains this updated specification.

Not all confusion about composite-order points ends up this lucky: the CryptoNote cryptocurrency framework also used a composite-order curve with a protocol that is secure

4. Tibor Jager, Jorg Schwenk, and Juraj Somorovsky. "Practical Invalid Curve Attacks on TLS-ECDH". in: (2015). URL: http://euklid.org/pdf/ECC_Invalid_Curve.pdf.

5. Simon Josefsson and Ilari Liusvaara. *Edwards-curve Digital Signature Algorithm (EdDSA)*. Internet-Draft draft-irtf-cfrg-eddsa-08. IETF Secretariat, Aug. 2016. URL: http://tools.ietf.org/internet-drafts/draft-irtf-cfrg-eddsa-08.txt.

on prime-order curves, leading to a double-spending vulnerability. The fix was simple: the code now rejects all points not in the prime-order subgroup of the elliptic-curve group. I find this case particularly interesting from a verification standpoint: writing a specification based on the existing code in CryptoNote would have most likely resulted in an insecure specification that matches the code, however, a functional specification extracted from the security argument of the protocol would have caught the bug.

Elliptic curves are not the only high-level structure whose implementations have suffered dangerous bugs: CVE-2006-4339 involves improper validation during parsing that allowed for bogus RSA signatures to pass as valid. Even more embarrassingly, Socat security advisory 7 admits unbounded loss of security due to a hardcoded constant that was required to be prime actually being composite (and of unknown origin).

Verification Scope

All verification efforts described in this paper rule out the multiprecision arithmetic bugs that result where the output is plainly incorrect. gfverif, hacl-star and this project also rule out bugs where the output represents the correct number, but in a non-canonical representative.

No other that I am aware of rules out realistic elliptic-curve-level bugs. While gfverif and hacl-star use Sage-verified elliptic curve formulas from EFD, there is not a rigorous connection between the preconditions of the formulas and the code calling them. I am not aware of any other project making any attempt at showing that the implemented elliptic curve satisfies the requirements of the higher-level cryptographic primitives.

All described projects include an unverified language translation phase:

1. hacl-star uses the F* extraction mechanism to

generate code.

2. gfverif replaces the integer types in the C code under test.

3. verify25519 translates qhasm code into SMT-solver formulas.

4. This project contains a formal semantics for a straight-line subset of C and pretty-prints the code in that language as C.

Unlike other projects, our pretty-printing does not require name mangling or structural changes: the constructs in our formalized output languages map one-to-one to constructs in standard C. In principle, we could actually prove that the formal language we use for output code maps directly to the C language as formalized, for example, in the CompCert C compiler frontend specification[6]. This has not been a priority because our output language is very simple; I do not expect that it would be difficult.

6. Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C Semantics: CompCert and the C Standard". In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 543–548. ISBN: 978-3-319-08970-6. DOI: 10.1007/978-3-319-08970-6_36. URL: http://gallium.inria.fr/~xleroy/publi/Formal-C-CompCert.pdf.

7.2 Engineering

Integration Effort

As the code synthesized for unsaturated arithmetic is output as plain C, it is rather straightforward to include it verbatim in all developments that can call C code. Unfortunately, the output code is even less susceptible to manual review than hand-written optimized C code: for example, the variable names are completely uninformative. The C code we have synthesized so far is still (subjectively) easier to read than hand-optimized assembly code.

Alternatively, a project seeking to use synthesized crypto code could include the synthesis pipeline as a compile-time dependency. In that case, the high-level specifications could be reviewed, relying completely on the Coq proof checker for correctness of the output code. The most obvious

downside of this approach is that it might drastically increase the resource requirements for compiling the application: currently, the Coq continuous integration build service takes more than an hour to build Coq and this project. Less obviously, the Coq version required for building most of this library (8.6) is not available in the repositories of any mainstream free software distribution.

Creating Implementations for New Parameters

The effort required for creating a cryptographic implementation in the style presented here depends on whether the required high-level algorithms are already present in our library.

If all required algorithms are present, specializing them to a particular cryptographic primitive is trivial.

```
Definition s : Z := 2^255.
Definition c : list limb := [(1, 19)].
(* modulus = 2^s - eval c = 2^255 - 1*19 *)

Definition numlimbs := 5%nat.
Definition bitwidth := 64.
```

Figure 7.1: Required parameters for synthesizing an implementation of arithmetic modulo $2^{255} - 19$. Only the last two parameters differ between target architectures.

Verifying New Algorithms on Existing Representations

In general, estimating the effort required for a new Coq development required good knowledge of the algorithm and its correctness proof – there is no general rule of thumb, and the measures of simplicity common in software engineering do not necessarily apply. Instead of trying to completely characterize the flexibility of this approach, I will give a couple of examples.

Adding a new elliptic curve formula for an existing coordinate system took me 20 minutes the last time I did it: write it down, find the formula that it should be equivalent to, state the equivalence, call `fsatz` to prove it.

Adding a new algebraic optimization to the field element representation is similarly simple, with the caveat that it

may not be obvious which representation to use or whether a new one is required. For example, factoring out multiplications by 2 and 19 in the formula synthesized in section 3.6 can be done by printing the current code, editing it into the desired form, and calling `ring` to prove it equivalent to the synthesized code. A major upside of the synthesis approach is that it minimizes sunk cost: optimizations that change large amounts of code can be introduced in a way that leverages the less-optimized code for their correctness proof.

On the other hand, optimizing multiplication using Karatsuba's trick is deceptive. It could be easily written down (either for a concrete number of limbs or in terms of split). However, I think that range analysis would fail to establish that $(a+b)(c+d)-(ac+bd)$ is always non-negative. For example, given $0 \le a,b,c,d \le 1$, range analysis would infer $0 \le (a+b)(c+d) \le 4$, and $0 \le (ac+bd) \le 2$, and then naively $-2 \le (a+b)(c+d)-(ac+bd) \le 4$ without noticing that $(ab+cd)$ cancels out. As the current back-end does not include finite integer types that can take both negative and positive values, the translation to C would fail.

Completely New Representations

Adding a new elliptic curve coordinate system requires understanding its complete representation invariant – which, for example, is *not* included in the Explicit Formulas Database. Equipped with this knowledge, creating a new coordinate system for an elliptic curve shape that has already been formalized should be simple in principle. However, every time I have done it, I have encountered new bugs in the `nsatz` tactic in the Coq standard library and spent several hours working around them.

Introducing a new representation for multi-precision integers would require first figuring out the general form of the desired representation (potentially very difficult) and

then formalizing it as in chapter 3 (rather straightforward). Every time we have done this, the work needed to go back to the drawing board at least once because we missed something in the first step.

7.3 Performance

To help the reader estimate performance of code synthesized using the methodology presented here, this section presents a case study of implementing the X25519 Diffie-Hellman function on 64-bit processors. I chose this primitive because our library contains correctness proofs of all low-level and high-level optimizations required to achieve decent performance. While the low-level optimizations cover large classes of realistic crypto primitives, the library lacks serious optimizations at the cyclic group level and above. For example, it is possible to synthesize an ed25519 implementation using the same field arithmetic as in this X25519 implementation, but I would have to either use a simplistic binary exponentiation procedure and accept a significant performance degradation or use an unverified fixed-window exponentiation routine. This is not to say that higher-level optimizations are particularly difficult to verify: the reason they are not present in our library is that none of us found them interesting enough to work on.

Table 7.2 shows the performance our synthesized code

Implementtation	CPU cycles	µs at 2.6GHz
amd64-64	148248	57
amd64-51	159384	61
sandy2x	160992	62
donna-c64	164716	63
fiat51d	176796	68
fiat51	196200	75
ref10	371472	143
ref	6278332	2415

Table 7.2: X25519 Diffie-Hellman Key Agreement Performance. The measurements were taken on an Intel Broadwell i7-5600U using the SUPERCOP benchmarking tool. Turbo Boost and Hyper-Threading were disabled.

along with the best X25519 implementations on SUPERCOP. The fastest non-assembly implementation, `donna-c64`, is is 16% faster than the automatically synthesized `fiat51`. As our synthesis pipeline does not include instruction-level optimizations, I paused the synthesis pipeline at the point where field operations are defined in terms of many mathematical-integer limbs and manually rewrote the order of operations to match that of `donna-c64` (see figure 7.2). Nevertheless, `donna-c64` is still 7% faster than the manually optimized `fiat51d` implementation.

Figure 7.2: One of the two manual optimizations differentiating `fiat51d` from `fiat51`. The other one, for field element multiplication, is very similar to this one.

```
(* Goal: ?square a =                  *)
(* let '(r4, r3, r2, r1, r0) := a in *)
(* (2*r0*r4 + 2*r1*r3 + r2^2,         *)
(*   2*r0*r3 + 2*r1*r2 + 19*r4^2,     *)
(*   2*r0*r2 + r1^2 + 19*(2*r3*r4),   *)
(*   2*r0*r1 + 19*(2*r2*r4 + r3^2),   *)
(*   r0^2 + 19*(2*r1*r4 + 2*r2*r3))   *)

(* fully automatic derivation would do [reflexivity] *)

(* micro-optimized formula from curve2551-donna: *)
  instantiate (1 := fun a =>
    let '(r4, r3, r2, r1, r0) := a in
    let d0 := r0*2 in
    let d1 := r1*2 in
    let d2 := r2*2*19 in
    let d419 := r4*19 in
    let d4 := d419*2 in
    let t0 := r0*r0 + d4*r1 + d2*r3      in
    let t1 := d0*r1 + d4*r2 + r3*(r3*19) in
    let t2 := d0*r2 + r1*r1 + d4*r3      in
    let t3 := d0*r3 + d1*r2 + r4*d419    in
    let t4 := d0*r4 + d1*r3 + r2*r2      in
    (t4, t3, t2, t1, t0)
  ).
(* equivalence proof: *)
  break_match; repeat apply (f_equal2 pair);
    ring_simplify; trivial.
Defined.
```

The primary reason this code is faster than the synthesized code is that `(19*a)*b` is faster than `19*(a*b)`. This in turn is true because the later compilation stages assign `a` and `b` to 32-bit variables, making `a*b` a 64-bit value and thus requiring a 64-bit multiplication when scaling by 19. Instead, the range analysis transformation will determine that `19*a` still fits in a 32-bit variable, and emit a 32-bit multiplication for the scaling. The C compiler further optimizes this to `19*a = a + (a + a*8)*2`, computed using two `lea` instructions without using the multiplier at all.

I do not have a fine-grained explanation for the observed 7% performance difference – both of them use the same radix 2^{51} and the same formulas. For both `fiat` and `donna`, the best timings (shown here) were achieved using `gcc -march=native -mtune=native -O3`

`-fomit-frame-pointer -fwrapv` version `6.3.1 20170306`. However, there is also an unexplained 3% performance difference between `donna-c64` and the `amd64-51` assembly implementation optimized for the very first Intel i7 processors. On the other hand, the difference between the two fastest qualitatively different implementation strategies (`amd64-64` and `amd64-51`) is just 7%, so it would be careless to neglect this difference.

To understand where the extra time is spent, I used `callgrind –dump-instr=yes` to generate instruction-level timing profiles of the `donna-c64` and `fiat-51d` implementations. I then identified the loop where most of the time was spent, disassembled it, and counted instructions as a crude measure of cost. Table 7.3 contains the findings. In short, `donna-c64` has 5% fewer instructions in the inner loop, and 10% fewer moves, but the counts of arithmetic instructions are nearly identical.

Instruction	`donna-c64`	`fiat51d`
`mov`	635	709
`add`	229	231
`mulx`	190	190
`adc`	180	180
`lea`	105	113
`and`	89	91
`xor`	60	60
`shr`	59	50
`shrd`	50	49
`movabs`	24	29
`sub`	20	20
`movq`	0	12
`movzbl`	2	2
`subl`	1	1
`shlb`	1	1
`neg`	1	1
`jne`	1	1
Total	1647	1740

Table 7.3: Inner loop instruction usage breakdown for `fiat51d` and `donna-c64`.

One possible explanation of this is that the use of short arrays for long-lived intermediate results (in `donna-c64`)

guides the C compiler towards better stack and register management, whereas the `fiat51d` implementation uses simple variables for all inner-loop state. However, memory optimizations are outside the scope of this project, and it is unlikely that the observed difference is due to arithmetic.

The choice of which implementation to use out of these depends on the performance constraints and target platforms. If the application only needs to run on `amd64` machines, `amd64-64` is probably the best choice. It is the fastest, and while the verify25519 verification may be seen as less rigorous than the Coq-based strategy here, the `qhasm` compiler used to create `amd64-64` is also much less complicated than C compilers used in the `fiat` and `donna` implementations. For use cases requiring portability across instruction sets, the question comes down to whether a 7% decrease in execution time outweighs the confidence derived from Coq-based correctness proofs. A middle-of-the-road solution could be to use a SMT-solver based tool to check that the `donna-c64` code is equivalent to `fiat51d` – I haven't tried it, but given the extent to which the two are similar, it is reasonable to expect that this task is much simpler than verifying `donna-c64` from scratch.

References

Bernstein, Daniel J. “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, Feb. 9, 2006. URL: http://cr.yp.to/papers.html#curve25519 (visited on 08/14/2016).

Bernstein, Daniel J. *Salsa20 specification*. Apr. 27, 2005. URL: https://cr.yp.to/snuffle/spec.pdf.

Bernstein, Daniel J., Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. “TweetNaCl: A crypto library in 100 tweets”. In: *Progress in Cryptology – LATINCRYPT 2014*. Ed. by Diego Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2015, pp. 64–83. URL: http://cryptojedi.org/papers/#tweetnacl.

Bernstein, Daniel J., Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *EdDSA for more curves*. 2015. URL: http://cryptojedi.org/papers/#eddsa.

Bernstein, Daniel J. and Tanja Lange. *Explicit-formulas database*. URL: https://hyperelliptic.org/EFD/.

Bernstein, Daniel J. and Tanja Lange. “Faster Addition and Doubling on Elliptic Curves”. In: *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*. ASIACRYPT’07. Kuching, Malaysia: Springer-Verlag, 2007, pp. 29–50. ISBN: 3-540-76899-8, 978-3-540-76899-9. URL: https://eprint.iacr.org/2007/286.

Bernstein, Daniel J. and Tanja Lange. *The Explicit-Formulas database*. Sept. 5, 2007. URL: https://cr.yp.to/talks/2007.09.05/slides.pdf.

Bernstein, Daniel J. and Peter Schwabe. *gfverif*. Jan. 1, 2016. URL: http://gfverif.cryptojedi.org/.

Bernstein, Daniel J. and Peter Schwabe. "NEON crypto". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2012, pp. 320–339. URL: http://cryptojedi.org/papers/#neoncrypto.

Brumley, Billy, Manuel B. M. Barbosa, Daniel Page, and Frederik R G Vercauteren. "Practical realisation and elimination of an ECC-related software bug attack". In: (2011). URL: https://eprint.iacr.org/2011/633.pdf.

Chen, Yu-Fang, 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*. ACM, 2014, pp. 299–309. URL: http://cryptojedi.org/papers/#verify25519.

Chlipala, Adam. *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant*. MIT Press, 2013. URL: http://adam.chlipala.net/cpdt/cpdt.pdf.

Chlipala, Adam. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics". In: *Proc. ICFP*. 2008, pp. 143–156. URL: http://adam.chlipala.net/papers/PhoasICFP08/PhoasICFP08.pdf.

Delaware, Benjamin, Clément Pit–Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proc. POPL*. 2015. URL: http://plv.csail.mit.edu/fiat/papers/fiat-popl2015.pdf.

Dockins, Robert, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. "Constructing Semantic Models of Programs with the Software Analysis Workbench". In: *Verified Software. Theories, Tools, and*

Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers. Ed. by Sandrine Blazy and Marsha Chechik. Springer International Publishing, 2016, pp. 56–72. ISBN: 978-3-319-48869-1. DOI: 10.1007/978-3-319-48869-1_5. URL: https://saw.galois.com/files/saw-vstte-final.pdf.

Jager, Tibor, Jorg Schwenk, and Juraj Somorovsky. "Practical Invalid Curve Attacks on TLS-ECDH". In: (2015). URL: http://euklid.org/pdf/ECC_Invalid_Curve.pdf.

Josefsson, Simon and Ilari Liusvaara. *Edwards-curve Digital Signature Algorithm (EdDSA)*. Internet-Draft draft-irtf-cfrg-eddsa-08. IETF Secretariat, Aug. 2016. URL: http://tools.ietf.org/internet-drafts/draft-irtf-cfrg-eddsa-08.txt.

Kaufman, Brett Max. *A Guide to What We Now Know About the NSA's Dragnet Searches of Your Communications*. Aug. 9, 2013. URL: https://www.aclu.org/blog/guide-what-we-now-know-about-nsas-dragnet-searches-your-communications.

Krebbers, Robbert, Xavier Leroy, and Freek Wiedijk. "Formal C Semantics: CompCert and the C Standard". In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 543–548. ISBN: 978-3-319-08970-6. DOI: 10.1007/978-3-319-08970-6_36. URL: http://gallium.inria.fr/~xleroy/publi/Formal-C-CompCert.pdf.

Langley, Adam. *Overclocking SSL*. Google, June 29, 2010. URL: https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html.

Lomont, Chris. *Fast Inverse Square Root*. 2003. URL: https://www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

Moglen, Eben. *Why Political Liberty Depends on Software Freedom More Than Ever*. Feb. 5, 2011. URL: https://www.softwarefreedom.org/events/2011/fosdem/moglen-fosdem-keynote.html.

Nakibly, Gabi, Jaime Schcolnik, and Yossi Rubin. "Website-Targeted False Content Injection by Network Operators". In: *CoRR* abs/1602.07128 (2016). URL: http://arxiv.org/abs/1602.07128.

Pit-Claudel, Clément. "Compilation using Correct-by-Construction Program Synthesis". Master's thesis. Sept. 2016. URL: http://pit-claudel.fr/clement/MSc/FiatToFacade_Pit-Claudel_2016.pdf.

Pottier, Loïc. "Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics". In: *CoRR* abs/1007.3615 (2010). URL: http://arxiv.org/abs/1007.3615.

Renes, Joost, Craig Costello, and Lejla Batina. "Complete addition formulas for prime order elliptic curves". In: *Proceedings of the 35th Annual International Conference on Advances in Cryptology, New York, USA*. Springer-Verlag, May 8, 2016. URL: https://eprint.iacr.org/2015/1060.pdf.

Things that use Ed25519. Nov. 9, 2016. URL: https://ianix.com/pub/ed25519-deployment.html.

Zinzindohoue, Jean Karim, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. "A Verified Extensible Library of Elliptic Curves". In: *IEEE Computer Security Foundations Symposium (CSF)*. 2016.