1 Introduction

Elliptic curve cryptography (ECC) has risen to the forefront in the past decade as an alternate approach of public key cryptography. Regular cryptography (e.g. RSA) relies on the difficulty of inverting certain mathematic operations over a regular Galois field. ECC is based on the algebraic structure of elliptic curves over finite fields, constructed such that it can offer the same level of security at only a fraction of the key size.

In order to implement such a scheme, the cryptosystem must be able to run large numbers of arithmetic operations modulo some large prime with cryptographically secure guarantees. This presents large opportunities for low-level optimizations of arithmetic over the underlying finite field, not just for performance, but also to prevent side-channel timing attacks, among other benefits accrued from essentially constant time operations. In particular, Dan Bernstein’s seminal work on Curve25519 [1], and more recently Curve41417 [2], demonstrates these methodologies.

Unfortunately, these hand-tuned optimizations only exist for for curves built over fields with specific moduli. Our work attempts to remedy that shortcoming.

2 Project Description

Most techniques for such modulus-specific finite field optimizations are fairly systematic and well-known, making them prime targets for automatic code generation.
Even though we may not achieve as dramatic speedups as demonstrated by [1] or [2], this tool would nonetheless be indispensable for anyone wanting to use a slightly less standard elliptic curve.

In particular, we seek to implement, given the input of any general prime, the following set of standard arithmetic operations:

1. Converting integers between standard binary representations into a format that allows for easier arithmetic by breaking 32- or 64-bit integers into a representation that fits in smaller, e.g. 24- to 26-bit, chunks. This simplifies arithmetic significantly by allowing carry-over logic to be delayed.

2. Addition modulo the prime in question.

3. Multiplication modulo the prime in question.

4. Squaring as a special case of multiplication. As this operation is commonly used and presents further optimization opportunities, it is deserving of extra consideration.

5. Multiple-and-Add as another special case (for the same reasons as above).


Having determined generalized optimization methods for any particular prime, it then becomes possible to automatically generate routines that implement optimized modular arithmetic without the need to hand code optimizations for every new prime.

3 Implementation

3.1 High-level description

The proposed implementation is as follows:

1. Take as input the modulus, the number and size of machine-words for the specialized internal operations (e.g. 10 32-bit words in the Go implementation of curve25519), and the number of significant bits to store.

2. Build an internal abstract-syntax-tree expression that implements the operations described above for our given parameters.
3. Apply standard compiler-expression optimization techniques, like common subexpression eliminations, to get the final output as close to a manual implementation as possible.

The proposed implementation would use Haskell and output code in Go.

3.2 Toy implementation of multiplication

In this toy implementation I will demonstrate multiplication of 2 50-bit integers using 2 32-bit machine words modulo $2^{50} - d$. The internal representation will be 25-25 bit integers.

The key observation is that splitting the integers into 25-25 bit representation allows us to multiply each block separately, and since the results are at most 50 bits, we can use native 64-bit machine words to hold these intermediary steps.

Let the first integer be $a = [a_1, a_0]$, and the second be $b = [b_1, b_0]$. To convert between these two representations we have

$$a_0 = a \& (2^{26} - 1)$$
$$a_1 = a \& ((2^{26} - 1) \ll 25)$$
$$b_0 = b \& (2^{26} - 1)$$
$$b_1 = b \& ((2^{26} - 1) \ll 25),$$

and conversely,

$$a = a_0 + (a_1 \ll 25)$$
$$b = b_0 + (b_1 \ll 25),$$

assuming that all the 25-bit representations are zero in bits 26-32.

Now to multiply $a$ and $b$ notice that we are actually just calculating

$$(a_0 + 2^{25}a_1)(b_0 + 2^{25}b_1) = a_0b_0$$
$$+ 2^{25}a_0b_1$$
$$+ 2^{25}a_1b_0$$
$$+ 2^{50}a_1b_1.$$
before we can add component-wise to get back the 25-25-25-25 bit result. This time we’ll use bars to denote the top 25 bits, and underscores for the bottom 25 bits:

\[
\overline{a_0 b_0} = a_0 b_0 & ((2^{26} - 1) << 25) \\
\overline{a_0 b_0} = a_0 b_0 & (2^{26} - 1).
\]

Similarly, we obtain \(a_1 b_0, a_1 b_0, a_0 b_1, a_0 b_1, a_1 b_1, \) and \(a_1 b_1.\)

Notice that now, our result \(c = [c_3, c_2, c_1, c_0]\) can be calculated by directly adding components, since we have

\[
c = a \ast b \\
= [a_1, a_0] \ast [b_1, b_0] \\
= a_0 b_0 + 2^{25} a_0 b_1 + 2^{25} a_1 b_0 + 2^{50} a_1 b_1 \\
= \overline{a_0 b_0} + 2^{25} \overline{a_0 b_0} + 2^{25} \overline{a_1 b_0} + 2^{50} \overline{a_1 b_0} \\
+ 2^{25} a_0 b_1 + 2^{50} a_0 b_1 + 2^{50} a_1 b_1 + 2^{75} a_1 b_1,
\]

yielding

\[
c_0 = \overline{a_0 b_0} \\
c_1 = a_0 b_0 + a_1 b_0 + a_0 b_1 \\
c_2 = a_1 b_0 + a_0 b_1 + a_1 b_1 \\
c_3 = a_1 b_1.
\]

Finally, to take the modulus of a 50-50 bit representation number, notice that

\[
N = (c_0 + 2^{50} c_1) \mod (2^{50} - d) \\
= c_0 \mod (2^{50} - d) + 2^{50} c_1 \mod (2^{50} - d) \\
= c_0 \mod (2^{50} - d) + 2^{50} \mod (2^{50} - d) \ast c_1 \mod (2^{50} - d) \\
= c_0 \mod (2^{50} - d) + d \mod (2^{50} - d) \ast c_1 \mod (2^{50} - d) \\
= c_0 + d \ast c_1 \mod (2^{50} - d)
\]

Thus, we merely multiply the upper 50 bits by the difference \(d\) and add them back into the lower 50 bits.

Notice that the process outlined above can be optimized by allowing non-zero bits past the first 25 bits in the 25-25 bit representations to reduce the number of necessary operations. While this leads to multiple representations of the same number, we are now allowed to perform canonicalization once, at the very end of the process, as opposed to every step of the way - so long as none of the intermediary steps run the risk of carry overflows, the results should be the same.
4 Deliverables

The project at its conclusion will consist of two deliverables:

1. A Haskell package fulfilling the requirements outlined above. The package will be freely available.

2. A short paper detailing the methodology, to be maintained throughout the project and distributed along with the package.

5 Collaboration

Lining Wang will also be working on this same project, essentially in full. Our collaboration policy will be to work as closely in parallel as possible, exchanging progress and ideas every 2 weeks. However, the deliverables will be entirely our own - only ideas will be exchanged. Our hope is that this policy will allow us to combine efforts to get a better approach than if we had been working alone.

6 Conclusion

It is my hope that the completion of this project will be immediately useful in the DeDiS group’s current and future efforts by providing good-enough modular arithmetic over any field of interest, particularly in Dissent. In addition, optimization is an art, and writing good optimizations is heavily system-dependent (e.g. CPU pipelining, which compiler and flags the user uses for the final Go code). Ideally, we would output several versions and allow the user to run tests on his/her target systems, but even building benchmarks is a tricky matter, since caching oftentimes artificially speeds up certain types of operations in no predictable manner. These extensions are outside the scope of the current project but present interesting followup material.

References