1 Overview

Public key cryptosystems depend on high-security operations performed over certain finite fields modulo large primes, e.g., Schnorr groups and elliptic curves. These operations present a variety of opportunities for optimization; in particular, Dan Bernstein’s hyperfast Curve25519 implementation [1] takes advantage of optimization opportunities offered on both the curve representation and modular arithmetic levels. However, his implementation is highly specific to a single prime, i.e., a single set of curve parameters, which heavily limits our exploration of other, potentially interesting primes. Because low-level optimization techniques are well-known and are readily generalized, and the number of operations needed for typical use is fairly small, we propose an automatic code generator that takes a parameter configuration as input and returns optimized code in Go, thus enabling future exploration of elliptic curves (e.g., Curve41417) for other researchers without the pain of manual implementation. Furthermore, we would be able to complete the implementation of Ed25519 by optimizing scalar arithmetic modulo the curve’s prime order, as well as arithmetic on the underlying field.

2 Project Description

2.1 Automated Code Generation

Our automatic code generator will be completed in three main steps:

1. Using Haskell and a set machine-word representation, generate an internal abstract syntax tree expression representing non-modular multiplication. The output of the operation should held in variables large enough to contain the answer (allowing for slop).

   For example, using two 32-bit words to represent a 64-bit integer, and multiplying them using the schoolbook method without regard to the modulo.

2. Obtaining the correct total bit-width of the output by incorporating a modulo-reduction operation, and implementing the carry logic to obtain two machine words in the correct range. This may produce slop where the carries are stored, which is amenable to future canonicalization.
For example, returning the result of multiplying two such integers as a 64-bit word, with respect to the modulo and appropriate slop for carries, and then separating the result back into two 32-bit words w/ 1-2 bits in each reserved for carries.

3. Applying compiler optimization techniques such as common subexpression elimination. Finally, by traversing the AST, we output code in the appropriate syntax of the target language (Go).

2.2 Example Configuration

This example, using a 50-bit modulus (specifically, $2^{50} - d$) where the size of each machine word is 32, and each integer is split into two words of 25 bits each, illustrates the main ideas of the process above.

Let $a = [a_1, a_0]$ and $b = [b_1, b_0]$ be two 50-bit integers, where $a_1$ represents the lower 32-bits of $a$ (26-32 bits zero), etc. We convert into our internal representation as follows. Let $m$ be the mask 0x1fffffff, or $2^{29} - 1$, since we require at least 25 bits:

$$a_0 = (a >> 25) \& m$$
$$a_1 = a \& m$$
$$b_0 = (b >> 25) \& m$$
$$b_1 = b \& m$$

For steps (1) and the beginning of (2), we use the schoolbook multiplication method to obtain a result consistent with our representation:

```
  a1 a0
x b1 b0
______
c1  c0
c3  c2
c5  c4
c7  c6
_____  
f3  f2  f1  f0
```

c0 = lower 25 bits of $a_0 \ast b_0$
c1 = upper 25 bits of $a_0 \ast b_0$
c2 = lower 25 bits of $a_1 \ast b_0$
, etc... except replace $b_0$ w/ $b_1$
in static single assignment form, this looks like

\[
\begin{align*}
g_0 &= a_0 \cdot b_0 \\
c_0 &= g_0 \& m \\
c_1 &= g_0 >>= 25 \\
g_1 &= a_1 \cdot b_0 \\
c_2 &= g_1 \& m \\
c_3 &= g_1 >>= 25 \\
g_2 &= a_0 \cdot b_1 \\
c_4 &= g_2 \& m \\
c_5 &= g_2 >>= 25 \\
g_3 &= a_1 \cdot b_1 \\
c_6 &= g_3 \& m \\
c_7 &= g_3 >>= 25
\end{align*}
\]

For the product \( f = [f_3, f_2, f_1, f_0] \), we obtain four 25-bit numbers (including carries):

\[
\begin{align*}
f_0 &= c_0 \\
f_1 &= c_1 + c_2 + c_4 \\
f_2 &= (f_1 >>= 25) + c_3 + c_5 + c_6 \\
f_3 &= (f_2 >>= 25) + c_7
\end{align*}
\]

To take the modulus of \( f \), we cannot merely discard the last two 25-bit numbers; we must also take into account that the modulus is not a precise power of two, without the use of if statements or other control structures that would decrease optimization and additionally complicate the AST generator.

Let \( h_0 \) and \( h_1 \) be the lower and upper 50 bits of the result \( f \). Let \( N \) be the result mod \( M \), where \( M = 2^{50} - d \). Then we have:

\[
\begin{align*}
N &= (h_1 \cdot 2^{50} + h_0) \mod M \\
    &= h_1 \cdot 2^{50} \mod M + h_0 \mod M \\
    &= (2^{50} \mod M) \cdot (h_1 \mod M) + h_0 \mod M \\
    &= d \cdot h_1 \mod M + h_0 \mod M \\
    &= (d \cdot h_1 + h_0) \mod M
\end{align*}
\]

This logic may give us multiple representations, for example of the value 0, but this can be resolved with canonicalization. We can also allow flexibility in the storage of the 25-bit words in 32-bit machine words by allowing one or two bits to be nonzero, so as to save some operations. Specifically, we do not need to trim intermediate results of the computation to be exactly 25 significant bits; we can canonicalize at the end instead of after every step.
3 Collaboration

I will be collaborating with Charles Jin on this project in parallel, exchanging ideas but arriving at separate deliverables. In particular, for the last phase (optimization), we expect to take different approaches and evaluate the overall speedup of each.

4 Deliverables

We expect to attain the following goals:

1. Weekly updates (approx. a page), posted on a public blog and/or submitted to the project adviser, detailing implementation progress.

2. A report/write-up of the project at the end of the semester, as well as related and future work.

3. A Python package that includes the automated code generator, available for use by other researchers.

References