1 Introduction

The goal of this project was to create a code generator that produced optimized code for arithmetic operations on finite fields of arbitrary (prime) size. Interest in this topic originally stemmed from djb’s work on curve25519, an elliptic curve over the field $2^{255} - 19$ which supports extremely fast addition, multiplication, and squaring of its field elements. Note that modulo arithmetic is not at all equivalent to elliptic curve operations, but are a critical component of those operations as cryptographic elliptic curves are usually defined over finite fields.

Since those operations are crucial in all cryptoschemes relying on elliptic curve cryptography, it seemed natural to create a tool that allowed cryptographers to explore other primes of various representation by generating optimized code specific to that prime, thus enabling the cryptographer to experiment with higher level design choices and testing instead of wrangling with and being excessively encumbered by low level code.

Because the prime is a usually large number of bits, we represent field elements using an array of machine words (fixed at 32 bits each). This intermediate representation requires the user to specify how many bits each machine word in the array represents (summing up to, of course, the number of bits of the prime). When given this “canonical” representation and the prime as input, the code generator outputs optimized code for performing the operations specified above (multiplication, squaring, addition, etc) on the field.

There exists several implementations of curve25519 in many languages, most notably golang’s crypto package implementation and SUPERCOP’s ref10 implementation (in C). I chose the ref10 implementation to serve as a check for correctness of the generator’s logic.
2 Progress summary

The code generator is able to reproduce almost all of the ref10 implementation’s functions, including \texttt{fe_mul} and \texttt{fe_sq}. The entire output can be viewed through the \texttt{lining_output.txt} file in the git repository. Here curve25519 field elements are represented by ten machine words of 32 bits each, using 25 or 26 bits per word as the canonical representation of each element.

One of the main features of the hand-optimized ref10 implementation is common subexpression elimination, which results in a significant speed improvement over the naive schoolbook multiplication method. This is discussed further in the multiplication and squaring section.

However, its correctness is untested on different primes and canonical representations (see section 4, "Limitations and future work").

3 Implementation

3.1 Code generation process

1. Ask the user to specify the parameters of the field: the canonical word representation, the number of words per field element, and the size of the underlying field expressed via the base and the offset (where prime = \(2^\text{base} - \text{offset}\)) (main.hs). Convert user input into Param datatype as declared in params.hs. (Note: currently untested for primes other than \(2^{255} - 19\). To be expanded in the future.)

2. Generate abstract syntax tree (as defined in ast.hs) representing optimized code that performs basic operations over this prime field using gen.hs.

3. Print out optimized C code from AST, properly formatted, indented, and with the correct syntax, using export.hs.

3.2 Abstract syntax

My abstract syntax was derived from Charles’ ”skeleton” implementation, with a few differences: there are no for-loops in the ref10 implementation, as they are substituted by an equivalent set of assignment statements; I created additional ”wrapper” datatype syntax for variables and assignments to reduce redundant code; and I created the datatype AllVars covering both variables and parameters, which previously were not recognized as variables despite being used as such in the ref10 implementation.

Overall, though it represents a limited subset of the C language (with no support for loops or control statements), the abstract syntax was adequate in representing the rigidly structured code of curve25519.
3.3 Function generation

Currently, automatic generation of the following functions are supported:

1. \texttt{fe\_zero} and \texttt{fe\_one}: sets field element to 0 and 1, respectively
2. \texttt{fe\_add} and \texttt{fe\_sub}: adds/subtracts two field elements, component word-wise
3. \texttt{fe\_copy}: sets one field element to another (both passed in as parameters), component word-wise
4. \texttt{fe\_cswap}: same as \texttt{fe\_copy}, but conditional on the value of the integer parameter \( b \) (0 or 1)
5. \texttt{load3}, \texttt{load4}: stores the first 3 and 4 bytes, respectively, of a byte array into an unsigned int64
6. \texttt{fe\_frombytes}: converts byte array into canonical representation of a field element
7. \texttt{fe\_tobytes}: converts from canonical representation of field element to byte array
8. \texttt{fe\_mul}: multiples two field elements, stores result in canonical word representation of field element
9. \texttt{fe\_sq}: squares field element, stores result in canonical word representation of field element

Support functions  In lieu of for-loops, the ref10 implementation assigned variables by setting each component word individually. This occurred often enough in the optimized code that I wrote the functions \texttt{genSimpleAssign} and \texttt{genOper}, which implicitly generated the appropriate assignment statements depending on if the input variables were local variables, with a type, or parameter variables.

Multiplication and squaring  These functions are especially challenging to generate because they require the code generator to recognize and identify precomputations. We give an overview of the generation of the precomputations for the \texttt{fe\_mul} function here (the \texttt{fe\_sq} function works similarly):

1. Accounting for neven representations, referring to the code block starting with

   \begin{verbatim}
   crypto_int32 f1_2 = 2 * f1;
   \end{verbatim}

   We can equalize the result of multiplying two blocks with different prefixes by multiplying the overall product by two, boosting it to the correct prefix of
the resulting product’s block. We pick the minimum set of statements which covers the uneven representation, which in this case is the set of odd variables \( f_1 \) through \( f_9 \).

2. **End-around carry precomputation.** This is the section starting with

\[
\text{crypto_int32 } g0_{19} = 19 \times g0;
\]

Suppose each field element is represented by two machine words (32 bits each) with 25 meaningful bits per word. Let \( h_0 \) and \( h_1 \) be the lower and upper 50 bits of multiplying two such elements together. Let \( N \) be the result mod \( M \), where \( M = 2^{50} - d \). Then we have:

\[
N = (h_1 \cdot 2^{50} + h_0) \mod M \\
= h_1 \cdot 2^{50} \mod M + h_0 \mod M \\
= (2^{50} \mod M) \cdot (h_1 \mod M) + h_0 \mod M \\
= d \cdot h_1 \mod M + h_0 \mod M \\
= (d \cdot h_1 + h_0) \mod M \\
= (d \cdot h_1) \mod M + h_0 \mod M
\]

In terms of curve25519, this means that to fit the product into the field element representation, any bits beyond the 255th (in the example above, any bits above 50) can be multiplied by the offset and added onto the rest of the bits.

I received great assistance from Charles for this part of the project, in particular for \texttt{fe\_sq}. I found his implementation extremely helpful in helping explain the concepts behind the operations, which I used to reconstruct my own version.

### 4 Limitations and future work

Currently, the correctness of the code generator is untested on curve parameters other than those of Curve25519. In particular, work remains to be done in generalizing the carry step of the \texttt{fe\_mul} and \texttt{fe\_sq} functions, determining the optimal load pattern for \texttt{fe\_frombytes}, and canonicalizing the result of arithmetic operations into the user specified representation.

In the future, we hope to add optimizations such as instruction scheduling and conversion to static single assignment form in order to maximize CPU pipeline efficiency and the number of statements that can be executed given the availability of arithmetic registers. There also remains two functions in the ref10 implementation that we have not generated: \texttt{fe\_invert}, which inverts a field element, and \texttt{fe\_mul12166}, which multiples a field element by the curve constant 12166 and is a crucial step in the Montgomery ladder step computation (a fast modular arithmetic multiplication method).
5 Conclusion

Overall, though the current implementation of the code generator falls short of its original goals, it is able to reproduce the hand-optimized code for curve25519 operations almost exactly (bar the order of product pairs in the multiplication and squaring functions). The project was also a valuable learning experience in areas ranging from abstract algebra concepts to cryptography to functional programming and abstract syntax tree generation.

6 Acknowledgments

I would like to thank Professor Ford for his continuous support and helpful explanations, and Charles Jin for his mentorship, moral support, and camaraderie during the semester.

References


https://github.com/floodyberry/supercop/tree/master/crypto_scalarmult/curve25519/ref10