Optimized Cryptography with LLVM
Non-Standard Bitwidth Types

Michael McLoughlin
mcloughlin@cmu.edu

https://mmcloughlin.com/15745

Contents

1 Introduction

An intriguing feature of LLVM IR is its arbitrary bit-width integer types. One might think this is just a curiosity, but in fact in *Formal-Methods-Based Bugfinding for LLVM's AArch64 Backend*, they observe:

> Does anyone actually care if the LLVM backends can deal with non-power-of-2 bitwidths? Turns out yes: in an optimized compile of LLVM itself, using LLVM, every integer width from 1 through 64 can be found. The largest integer that occurs when compiling LLVM using LLVM is 320 bits wide.

These types are about to become even more compelling following the inclusion of an N-bit type in the upcoming C23 standard [18]. Until now these types had been reserved for dedicated users, only recently exposed via the Clang 11 `_ExtInt` extension [13] and prior to that only available when directly producing LLVM IR. As this niche feature graduates to a `_BitInt(N)` type in standard C, its use will no doubt proliferate and developers will have raised expectations for compiler performance.

This invites the question of how we might leverage arbitrary bitwidth types for specialist applications? Does LLVM perform well on these use cases, and what optimization opportunities are there? In *Adding a Fundamental Type for N-bit integers* several motivating use cases are outlined: "using 256-bit integer values in various cryptographic symmetric ciphers like AES, when calculating SHA-256 hashes, representing a 24-bit color space, or when describing the layout of network or serial protocols." Low-level hardware applications are also suggested, for example FPGA designs can support extremely wide integers. More broadly, the appeal of these types is to allow the programmer to more precisely specify their intent thus improving the clarity and correctness of domain-specific code, while relying on the compiler for safety and performance guarantees. This project will probe this claim in the specific case of cryptography, a domain where both correctness and performance are paramount.

1.1 Opportunity

Our target application will be high-performance cryptography. Given an implementation of a critical cryptographic algorithm written as simply as possible using arbitrary bitwidth types, how close can the compiler get to hand-tuned performance?

Arbitrary bitwidth types enable cryptography engineers to write concise implementations of certain cryptographic algorithms, instead of using hand-tuned implementations in which machine-specific optimizations and hand-written assembly are pervasive and may lead to bugs. Can we provide compiler opti-

mizations that would get the concise implementation close to the hand-tuned performance?

Our project scope will focus more narrowly on elliptic curve cryptography over finite fields, which rely efficient multi-precision arithmetic modulo fixed large prime numbers. For example, one widely deployed elliptic curve is Curve25519 [3], which reduces to operations on 255-bit values modulo the prime $2^{255} - 19$. Implementations of such curves are famously complex, but they could be quite easily expressed using arithmetic operations on the `_BitInt(255)` type. Hand-tuned implementations leverage a variety of tricks for fast reduction modulo primes with special mathematical structure, as well as micro-optimizations for multi-precision arithmetic and carry chains. How many of these tricks can we exploit in LLVM, and therefore produce safer high-performance cryptography?

1.2 Approach

Our goal is to assess the feasiblity of optimized cryptography in LLVM nonstandard bitwidth types by establishing and focussing on a high-value target benchmark. Our hope is to get as close as possible to state of the art performance of hand-tuned implementations, and to explore what challenges arise on the way.

Our target benchmark will be the X25519 cryptographic algorithm implemented as simply as possible in C with the `_BitInt(255)` type. This is a critical widely deployed algorithm, and success optimizing it with LLVM would therefore be of potential interest.

The scope of possible optimizations is broad. For project feasiblity we will focus on LLVM mid-level optimization which can be applied at the IR level. In addition, we limit attention to a particular class of finite fields that are cryptographically important, namely those with a prime modulus that has the Crandall prime form. Nonetheless, we believe that this restricted scope is still broad enough to meaningfully assess the feasiblity question for optimized cryptography in LLVM. At the end of this report, we'll cover a pleathora of additional directions that could be pursued.

1.3 Related Work

Optimization of elliptic curve and finite field operations on CPUs is a thorougly studied domain. Just a small selection of material on this topic are [6, 11, 8, 10, 17]. This material has mostly focussed on optimizing specific elliptic curves. The techniques are broadly applicable to classes of curves, but have typically not been packaged up into compilers.

Code generation for cryptography has also been pursued, for example the `fiat-crypto` [9] project outputs verified finite field optimizations, though they are known to have sub-optimal performance. Building on top of this, the `ecckiila` [1] project generates entire elliptic curve implementations. Projects of this kind are better seen as code generators rather than optimizing compilers. That said, verification offered by `fiat-crypto` is a massive source of complexity not addressed in our project.

The use of LLVM wide types for cryptography has been pursued before by the `constantine` [20] project, where one backend produces LLVM IR directly. The fact this has been tried before is some vindication for the project direction. We also note that they typically prefer to use their custom JIT compiler instead, which suggests that there are performance improvements to be had in LLVM.

Therefore the broad topics have seen related work, but the specific niche of optimized cryptography in LLVM is still open for improvements. The fact that it's now possible with standard C also makes the direction even more compelling.

1.4 Contributions

The contributions of this project are:

- Demonstrate how newly standardized `_BitInt(N)` types can be used to produce simple high-level implementations of complex cryptography.
- Provide a suite of LLVM analysis and optimization passes that collectively optimize our target benchmark by 80.1x, and within a factor of 1.6 of a popular optimized external library. This prototype demonstrates the feasiblity of optimized cryptography in LLVM.
- Implement Crandall Reduction optimization for fast modular reduction of cryptographically relevant Crandall primes.
- Design and implement a Reduction Analysis which identifies self-contained expression graphs of finite field computation.
- Implement an Incomplete Reduction optimization which strips out redundant modular reductions and maintains intermediate values in partially reduced states.
- Identify future optimization opportunities which we believe would approach performace parity with hand-tuned code for this class of cryptographic applications.

2 Experimental Setup

Our primary optimization target for experimentation is the X25519 cryptographic function [15] implemented as simply as possible in C with the `_BitInt(255)` type. The X25519 function is a scalar multiplication on the Curve25519 elliptic curve, which ultimately reduces to arithmetic in the finite field of integers modulo the prime $p = 2^{255} - 19$. X25519 was selected due to its importance as a real-world cryptographic algorithm, and because the implementation strategies required to optimize its finite field operations could be transferable to other cryptographic primitives. Finally, our C implementation with 255-bit types is as simple as reasonably possible to prompt exploration of how much optimization can be automated by the compiler.

2.1 X25519

In this section we will see snippets from the implementation to give a sense of the optimization target. Please refer to the full source code for details.

2.1.1 Finite Field

At its core, X25519 operates on the finite field of integers modulo the prime $2^{255} - 19$. Elements of the field are represented with the unsigned `_BitInt(255)` type.

```
#define FP25519_BITWIDTH (255)
#define FP25519_BYTES ((FP25519_BITWIDTH + 7) / 8)

typedef unsigned _BitInt(FP25519_BITWIDTH) elt_t;
```

The prime modulus may be defined as a regular C literal:

```
// `uwb` suffix proposed in N2775 "Literal suffixes for bit-precise integers".
#define FP25519_LITERAL(X) ((elt_t)X##uwb)
#define FP25519_P    \
    FP25519_LITERAL( \
        0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed)
```

Finite field operations are then implemented with standard C operators followed by modulo operations, ensuring we use a large enough type to contain the intermeditate result. For example, finite field addition is as follows.

```
inline __attribute__((always_inline)) elt_t fp25519_add(elt_t x, elt_t y) {
    typedef unsigned _BitInt(FP25519_BITWIDTH + 1) add_elt_t;
    const add_elt_t s = (add_elt_t)(x) + (add_elt_t)(y);
    return (elt_t)(s % FP25519_P);
}
```

Multiplication is implemented similarly, instead using a double-wide intermediate type. Note we also make use of inlining attributes to ensure that these computations are inlined into more complex expression graphs used in elliptic curve operations.

2.1.2 Scalar Multiplication

Having defined finite-field operations, the core of the X25519 function is a scalar multiplication on the Curve25519 elliptic curve, which is typically performed with the Montgommery Ladder technique. The mathematical details are not imporant from our perspective. From a computational point of view, it boils down to a loop over the bits of a 255-bit scalar, and each iteration does some conditional swaps and computes a straight-line arithmetic expression using finite field operations. The most performance critical part of the code is small enough to present here.

```
uint8_t swap = 0;
for (int pos = 254; pos >= 0; --pos) {
    const uint8_t b = 1 & (k[pos / 8] >> (pos & 7));
    swap ^= b;
    fp25519_cswap(&x_2, &x_3, swap);
    fp25519_cswap(&z_2, &z_3, swap);
    swap = b;

    // https://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m
    const elt_t A = fp25519_add(x_2, z_2); // A = x_2 + z_2
    const elt_t AA = fp25519_sqr(A); // AA = A^2
    const elt_t B = fp25519_sub(x_2, z_2); // B = x_2 - z_2
    const elt_t BB = fp25519_sqr(B); // BB = B^2
    const elt_t E = fp25519_sub(AA, BB); // E = AA - BB
    const elt_t C = fp25519_add(x_3, z_3); // C = x_3 + z_3
    const elt_t D = fp25519_sub(x_3, z_3); // D = x_3 - z_3
    const elt_t DA = fp25519_mul(D, A); // DA = D * A
    const elt_t CB = fp25519_mul(C, B); // CB = C * B
    // x_3 = (DA + CB)^2
    x_3 = fp25519_sqr(fp25519_add(DA, CB));
    // z_3 = x_1 * (DA - CB)^2
    z_3 = fp25519_mul(x_1, fp25519_sqr(fp25519_sub(DA, CB)));
    // x_2 = AA * BB
    x_2 = fp25519_mul(AA, BB);
    // z_2 = E * (AA + a24 * E)
    z_2 = fp25519_mul(
        E, fp25519_add(AA, fp25519_mul(FP25519_LITERAL(121665), E)));
}
```

Note that the formula in the inner loop is derived from the Explicit Formulas Database [5, 4], a compendium of the best-known formulae to perform various elliptic curve operations. While this computation is specific to X25519, this type of construct is representative of an entire class of cryptographic code.

An important caveat to note at this point is that our Montgomery ladder is *not constant-time* in one imporant respect, namely the conditional swap uses a branch rather than the typical XOR trick. This simplification was made to enable the possibility of optimizations crossing the conditional swap. This is likely not a blocker for the project's techniques in the long run, but the simplification was made subject to the limited time allowed for the project. We note in future work (Section 6) that handling constant time crytography would be critical if these techniques were ever to be used in real-world settings.

2.2 Testing

Alongside the target X25519 implementation, we have a test suite and benchmark against a popular hand-tuned external implementation.

2.2.1 Test Suite

The test suite checks both the low-level finite field operations, and the X25519 function against test vectors [15, Section 5.2]. The test suite is valuable to provide confidence in our target implementation, to confirm our benchmarks are representative, and to validate that our custom optimizations preserve correctness.

2.3 Benchmark

The benchmark compares the target implementation against the hand-tuned version in `libsodium` [16], a widely deployed and well-optimized cryptographic provider, which we expect to have good performance, although perhaps not absolute peak. OpenSSL and EverCrypt [19] were also considered for inclusion, but `libsodium` was considered sufficient for this stage of the project. Expanding the benchmark suite would be valuable in future work (Section 6).

Benchmarks are orchestrated with Google Benchmark compiled with `libpfm4` for performance counter support. The target functions are our `x25519_scalar_mult` described in Section 2.1 and `crypto_scalarmult_curve25519` from `libsodium`, both operating on the same synthentic inputs. Benchmarking was performed on an Intel Core i7-13700K Processor tuned for low-variance by setting performance scaling governor and disabling Intel Turbo and Hyperthreading. Benchmarks were executed for 30 seconds with cycle and instruction performance counters collected alongside timing.

2.4 Baseline

Our benchmark establishes a baseline to work from, and a performance goal to match. The baseline will be out-of-the-box performance from Clang 17.0.3with `O3`. Initial results show that the baseline is a massive 127.3 times slower than `libsodium`.

Implementation	Time (us)	Iterations	Cycles
Baseline	4682.75	8962	15771906.96
`libsodium`	36.80	1141472	124794.13

3 Implementation

Subject to project time constraints, our technical solution focused on a subset of the problem:

Mid-level optimization. Our optimizations operate at the LLVM IR level. There are certainly significant gains to be had in the LLVM backend also, which we would expect to be complementary to this work.

Crandall primes. Our optimization focussed on finite field operations for Crandall primes, of which the X25519 finite field is a special case. Nonetheless, we will see that a significant portion of the optimization work is agnostic to the form of the prime modulus, and we would expect it to be feasible to extend it to support other specialized cryptographic prime forms.

The optimizations and analyses we have implemented are:

Crandall Reduction. Crandall reduction detects modular reduction `urem` operations that appear in finite field operations and rewrites them to use specialized alternatives when the modulus has the Crandall prime form. This optimization is performed by the `CrandallReductionPass`.

Reduction Analysis. Reduction Analysis identifies self-contained arithmetic expression graphs which are guaranteed to be reduced modulo the same modulus. The purpose of this stage is to detect when intermediate reductions may not be necessary, and could be safely rewritten to use partially non-reduced forms. This analysis is provided by the `ReductionAnalysis` pass, which provides a `ReductionInfo` result to consuming passes.

Incomplete Reduction. Incomplete reduction utilizes the prior techniques to eliminate unnecessary `urem` instructions. Reduction analysis results allow us to determine when `urem` instructions are candidates for elimination, and instead Crandall reduction techniques can be used to leave it in an incompletely reduced form instead. Incomplete reduction was implemented as an evolution of the `CrandallReductionPass`.

3.1 Crandall Reduction

Crandall reduction is a technique for fast modular reduction by constants with the *Crandall* form. These reductions appear frequently in cryptographic finite field implementations. *Crandall primes* are those of the form $p = 2^n - c$ for a small constant c, but the optimization would actually apply for any modulus of that form whether prime or not. The optimization leverages the simple fact that for a modulus $p = 2^n - c$ we have $2^n \equiv c \pmod{p}$. Therefore, given a value x to be reduced, if we represent it as its low n bits l and remaining high bits h, then:

$$x = 2^n h + l \equiv ch + l \pmod{p},$$

and $ch + l$ will in most circumstances be much smaller than x. Therefore, this does not complete the reduction by p but it cheaply produces a result equivalent modulo p.

The `CrandallReductionPass` detects and applies this optimization technique to LLVM IR. The initial form of the pass was a local optimization:

1. **Detection.** First we use LLVM pattern matching to locate `urem` instructions with a constant integer modulus. Given the constant modulus we need to determine whether it has the Crandall form $2^n - c$. We deduce n

from the number of active bits in the constant, subtract 2^n to get a candidate for c and then accept it if the candidate is less than a configured threshold for c.

2. **Reduction Generation.** Having identified a candidate reduction, generate LLVM IR instructions to perform a single Crandall reduction step. That is, to compute $ch + l$ for the high and low bits h and l. This is done with the `IRBuilder`: `and` and `lshr` instructions are used to separate the low and high bits, and then `add` and `mul` instructions compute the reduced result. We refer to this as a single *Crandall step*. In the initial version of the pass, only a single step was performed and the original `urem` instruction is left intact, with its input replaced by the output of the Crandall step. For correctness we need the `urem` to compute the fully reduced output.

Despite its simplicity, the results of this pass alone are shockingly good. Compiling with the `CrandallReductionPass` that implements single-step Crandall reduction only, we observe a 24.0 speedup over baseline.

Implementation	Time (us)	Iterations	Cycles
Baseline	4682.75	8962	15771906.96
Crandall Single Step	195.29	215216	662345.84

These results are likely so good becuase of how code generation works for large multi-precision `urem` instructions in LLVM. Specifically, this is handled by the `ExpandLargeDivRemLegacyPass` which generates looping code to perform large remainder operations, and it's a correct but inefficent way of doing it for these special cases. Even though the Crandall step does not actually remove the `urem` instruction, the insertion of the step prior to the instruciton allows LLVM to in some cases replace the reduction with a simpler conditional subtraction. Or at least when the inefficent construct remains, there is still dynamically less work to be done. The results are therefore effective, but there's still a long way to go to reach hand tuned performance.

Typical optimized implementations of finite-field reductions will apply Crandall reduction steps multiple times, as long as each step is guaranteed to produce an output with a smaller upper bound. This is a natural extension of the `CrandallReductionPass` also, with a goal to eventually replacing at least some of the `urem` instructions. The question is what upper bound are we aiming for, and a common trick applied to Crandall moduli is reduction *aligned to the machine word boundary*. In practice we know that our 255-bit quantities will occupy 256-bits in registers, typically either 4x64 or 8x32. Our target for reduction steps is for the result to once again fit in 256 bits, not necessarily to get the completely reduced 255-bit value. For an implementation of X25519 this means we will be applying Crandall reduction with the identity $2^{256} \equiv 38$ at the 256-bit boundary, instead of $2^{255} \equiv 19$ at the non-aligned boundary. In the general case of a Crandall modulus $2^n - c$ we are interested in the next bit

width $m \geq n$ that's a multiple of the machine word size, and the constant c is adjusted accordingly.

Having established a target upper bound, we need to know the possible range of values following each Crandall reduction step. At this point we hoped to rely on LLVM infrastructure, in particular `LazyValueInfo` which provides deduced `ConstantRange` bounds for each value in a function. However, the built-in analysis is only able to deduce the trivial bound provided by a Crandall step, not the tighter bound that becomes more relevant as multiple steps are applied. Therefore it was necessarily to implement a custom `getReductionBound` method to compute the possible range of values output from a Crandall reduction step. With this in place, multi-step Crandall reduction works as follows:

1. **Determine Target Reduction.** Given a detected Crandall modulus $2^n - c$ we round n up to the next multiple of a machine word m. Now m and $d = 2^{m-n}c$ take the place of n and c in the original procedure.
2. **Determine Input Bound.** Use `LazyValueInfo` to initialize a known upper bound for the input to the reduction.
3. **Iterate Until Reduced.** At each iteration, compute how much this step would reduce the upper bound by. Bail if the step would not decrease the bound. Otherwise proceeed to generate code for the Crandall reduction, just as we did for single-step.

Note at this stage the final `urem` instruction is still preserved, as we do not know whether it is safe to remove. That will be the priority for upcoming optimizations. Results for the multi-step reduction are underwhelming at this stage.

Implementation	Time (us)	Iterations	Cycles
Crandall Single Step	195.29	215216	662345.84
Crandall Multi Step	194.95	215445	661183.93

The change here is marginal at best, though it's surprising is was so small. In many cases only one Crandall step is added anyway. But what we're likely seeing here is that performance is still dominated by the remaining `urem` instructions. Most of the change in the upper bound of the value under reduction comes in the first Crandall step, and subsequent ones do not significantly reduce the work required in the `urem`.

However, the importance of multiple Crandall steps isn't in an optimization in itself, rather that they allow us to place a hard bound on the result, which is critical for correctness. Therefore with this capability in place, we have unlocked the possibility of removing the reduction entirely.

3.2 Reduction Analysis

Thus far we have seen how Crandall reduction allows for substantial performance improvements for modular reduction by primes of special forms. In addition,

when we apply these techniques to a computation of $x \bmod p$ it is most convenient to arrive at a partially reduced result that's equivalent to the complete reduction modulo p but not necessarily equal to it, for example by stopping reduction steps when we have a result that fits in a convenient multiple of machine words. We would like to be able to stop at this point and elide the final `urem` instruction, but when would it be safe to do so? The purpose of Reduction Analysis is to answer this question, and tell us when we have self-contained expression graphs that are all operating on arithmetic modulo the same prime p, and in that case we can potentially defer complete reduction until later.

Let's consider a simple example first, drawn right from the inner loop of X25519. The variable `DA` is computed as:

$$D * A = (x_2 + z_2) * (x_3 - z_3)$$

However, since these are not plain arithmetic operations, there are implicit modular reductions by p generated in the code. The expression graph we are actually processing is:

$$(D * A) \bmod p = (((x_2 + z_2) \bmod p) * ((x_3 - z_3) \bmod p)) \bmod p$$

Mathematically, the interior modular operations are not necessary. The computation is correct if we replace $(x_2 + z_2) \bmod p$ with any value *equivalent* to $x_2 + z_2$ modulo p. The key observation is that the full reduction is not necessary because we know that there is a complete reduction modulo p later in the expression graph.

The Reduction Analysis pass computes this property. A value is *reduced modulo p* if all uses of the value are also reduced modulo p, and only involve operations that *preserve modular arithmetic*. Critical examples of modular arithmetic preserving operations handled by the first iteration of the reduction analysis pass are:

Non-wrapping Arithmetic. Plain arithmetic instructions such as addition, subtraction and multiplication propogate the reduction property as long as we know they will not wrap. Wrapping at machine word boundaries means the operation is no longer a mathematically pure arithmetic operation.

Casts. Cast operations such as zero extension or truncation can also propagate the reduction property, as long as we can prove they are pure type transformations and an effective no-op on the value.

A notable additional class of instructions that can propagate reduction property is is ϕ nodes, which will be important later.

The `ReductionAnalysis` pass is an LLVM function analysis pass providing the `ReductionInfo` result to consumers. Internally, it uses a worklist-based algorithm which iteratively updates the state of each value. The possible states of each value are reminiscent of lattice states for classic constant propagation. Possible states are:

`UNDEF` Undefined state, meaning we have no information about this value.

`REDUCED(`M`)` Reduced state. All observed uses of the variable are also reduced modulo the same modulus M, and only involve preserving instructions.

`NOTREDUCED` Not reduced. This can mean the value is used in non-preserving instructions, or is used by instructions that are reduced modulo different moduli.

Lattice states can be merged with other lattice states with the expected behavior: imporantly the reduction state is only preserved if the two moduli are the same, and meeting with `NOTREDUCED` always produces `NOTREDUCED`. The worklist algorithm works as follows:

1. **Initialization.** All instructions are enqueued to be visited.
2. **Modular Reductions.** `urem` instructions by constant moduli *generate* the reduction property for their operand. In this case, the operand is marked reduced in the lattice and enqueued to be visited if its state has changed.
3. **Propagating Instructions.** Propagating instructions merge their current state with their operands. If this leads to any state changes for operands, they are added to the worklist.
4. **Non-Propagating Instructions.** Non-propagating instructions are handled the same, but they merge the `NOTREDUCED` state with their operands and enque anything changed.

The end result is lattice states for every value in a function which is exposed with the `ReductionInfo` type. Clients can use this to determine which values have the reduced property, and if so with which modulus. As we'll see, the most interesting case is `urem` instructions that have the reduced property, as these are candidates for removal.

3.3 Incomplete Reduction

The goal of incomplete reduction is to actually remove modular reductions that can be deferred until later. Given results of reduction analysis, we can determine which reductions are candidates for removal, and transform the expression graph to remove it. You might think this would be as simple as pruning some `urem` instructions from the function, but complexity arises because an incomplete reduction causes larger output values. In turn, larger output sizes require wider types to hold their possible value ranges. For example, in our original X25519 example we would have a complete reduction to 255-bits followed by a truncation to the `i255` type, which in turn feeds into an `i510` multiply. Eliding the final reduction means the output is now 256-bit, the trucation needs to be `i256` and the multiply `i512`, and so on. In this manner, the choice to use an incomplete

reduction cascades into type resizing across the expression graph. Therefore the extension of the `CrandallReductionPass` to support incomplete reduction transitioned it from a relatively simple local optimization into a delicate global optimization.

The implementation graduated through two main versions: firstly operating in basic blocks, and the second handling ϕ nodes. Substantial changes were required between them due to the presence of cycles once ϕ are allowed in the expression graph. We will only describe the design of the second iteration, but present results from both iterations. The extended `CrandallReductionPass` works in three phases:

1. **Plan Reductions.** Determine how to handle each modular reduction in the function: what bound to reduce it to, which Crandall parameters to apply, and whether the reduction should be left incomplete.

2. **Plan Ranges.** Given a set of reductions which will be left incomplete, determine the possible range of values that every value in the transformed expression graph will now have. These ranges are used to determine new types.

3. **Rewriting.** Given reductions to be rewritten, and new types for the expression graph, actually replace the expression graph with a parallel rewritten version. Delete the old expression graph and fix up ϕ nodes.

3.3.1 Plan Reductions

Reduction planning determines how each `urem` by a constant will be handled by the pass. It detects `urem` instructions by Crandall moduli and for each one computes:

- **Crandall Parameters.** Given $p = 2^n - c$ it aligns n to the next machine word boundary m and computes the adjusted constant d we'll apply in Crandall steps.

- **Incomplete?** We determine whether the reduction will be left incomplete. This consumes the output of the Reduction Analysis. Specifically, a `urem` instruction with a modulus p will be left incomplete if reduction analysis reports the instruction has state `REDUCED(`p`)`.

3.3.2 Plan Ranges

The plan ranges phase aims to determine which types we need to use to safely preserve the value of the computation through the expression graph. Recall that in Reduction Analysis we insisted that arithmetic was non-wrapping and casts were value no-ops. This must still be true in the transformed graph even with larger outputs of incomplete reductions.

The end result of this phase is a set of deduced LLVM `ConstantRange` ranges that each value in the current expression will have in the transformed version.

Constant ranges for values are initialized to their known range according to LLVM's `LazyValueInfo`. We use a worklist again, which is originally populated with any reduction instructions planned to be incomplete in the prior planning phase, since these now have different sizes than the original graph. We iteratively process the worklist and update instruction ranges depending on their operands, enqueing their users to the worklist if there has been any change. Instructions are handled as follows:

- **Reductions.** If the reduction is incomplete, its output range is set to the full range $[0, 2^m)$ where m is the bitwidth we've planned to use for Crandall reduction.
- **Addition.** Given input ranges for the two operands, we compute the complete possible range of values of the unsigned addition. Note care has to be taken here since default mathematical operations on LLVM `ConstantRange` will wrap at the bit size of the inputs. Therefore, we compute the maximum bitwidth of the output given the inputs, which is the max of the two inputs plus an extra bit for a carry. Then we resize inputs to the maximum possible bit width, compute the sum on the resized ranges. Finally, we truncate the result output range to the minimal number of bits required to represent it.
- **Multiplication.** Multiplication is handled in the same way as addition. The only difference is the maximum bitwidth is now the sum of the bitwidths of the inputs.
- **Subtraction.** Subtraction is more delicate. Since this subtraction is being rewritten, we know from reduction analysis that the original subtraction would not wrap. The new version of the instruction must not either, however the value being subtracted could now be larger due to incomplete reduction. That is, given an instruction $x - y$, the input y could now be larger. We address this by *offsetting* the x operand by adding multiples of the modulus. That is, our subtraction will be transformed to:

  $$(ip + x) - y \tag{1}$$

  for a suitable number of offsets i such that we know the result will not wrap. Note that adding offsets here can also change the number of bits required to represent $ip+x$, and therefore of the entire computation. Therefore, range planning for a subtraction adjusts the offsets and bitwidths until it can prove the result would not overflow. The number of offsets must also be preserved for the code rewriting phase that follows.
- **Casts.** Casts are treated as *passthroughs* for the purposes of range planning. Recall again that reduction analysis confirmed that cast operations are value no-ops. Therefore at this stage we just see through cast operations and assume the value has the same range as its input. Cast operations will be inserted as necessary in rewriting.

- ϕ **Node.** The range of a ϕ node is the union of the ranges of all incoming values.

3.3.3 Rewriting

Given the results of range planning, we now have to rewrite the expression graph to implement incomplete reduction and resize any intermediate values. Rewriting is performed recursively, with operands rewritten prior to the instruction itself. Some points on the handling of specific instruction types:

- **Reductions.** Reductions are rewritten as described in Section 3.1 on Crandall reduction. Multiple crandall steps are generated until the value has been reduced to its planned width. If the reduction is planned incomplete, no final `urem` is produced. Care is required when when the reduction is complete, since we can think of it as an exit point from the rewritten expression graph. A complete reduction requires the final `urem` as well as potentially an integer cast to ensure the output has the same type as it did before.
- **Binary Operators.** The result of range planning is used to determine the integer type required for the operation such that it will not wrap, and integer casts are applied to the inputs to ensure they're all of the same type. In the special case of subtraction, we also need to apply modulus offsets to the left-hand-side.
- **Casts.** Casts are not *explicitly* rewritten, and we passthrough to rewriting the operand. Casts do end up implicitly regenerated when values are used as operands to other resized insructions.
- ϕ **Node.** Handling of ϕ nodes must be done with care to avoid a recursive cycle. In the first rewriting phase, ϕ nodes are replaced with a new *placeholder* ϕ, which will be populated at the final stage of rewriting. Unlike all other instruction types, we do not recurse into operands when processing a phi node, as this could induce a cycle.

Once the transformed expression graph has been written we fixup the function:

1. Populate placeholder ϕ nodes by inserting the rewritten incoming values. Integer casts may be required here too, and these have to be written into the source basic blocks.
2. Finally, delete the old expression graph. Replace any uses of rewritten values that need to be consumed by the preserved portion of the function. Specifically, this means any complete reductions remaining.

3.3.4 Results

Results for incomplete reduction are very encouraging. In comparison to local optimizations that leave complete reductions intact, we see a 2.1 speedup in the initial version of incomplete reduction that did not handle ϕ nodes.

Implementation	Time (us)	Iterations	Cycles
Crandall Multi Step	194.95	215445	661183.93
Incomplete Reduction	93.11	451088	315782.34

However, it was clear from inspecting generated code that remaining complete reductions in the inner loop were a substantial drain on performance. This motivated a push to adapt the incomplete reduction algorithm to support ϕ nodes, and therefore allowing the entire loop to operate over non-reduced values. This change actually required substantial surgery on the implementation: initially everything was handled in the recursive rewrite function, but ϕ node handling prompted the split into distinct range planning and rewrite phases. The effort paid off in performance improvements, where we see another 1.6 speedup that takes us to a factor of 1.6 away from hand-optimized `libsodium`.

Implementation	Time (us)	Iterations	Cycles
Incomplete Reduction	93.11	451088	315782.34
Incomplete Reduction over ϕ	58.46	718515	198282.05
`libsodium`	36.80	1141472	124794.13

3.4 Orchestration

Finally, we note that orchestration of custom passes was delicate due to how they interact with existing optimizations. The LLVM passes are most easily able to deduce the semantics of the computation after some light optimization passes have run, but before the entire suite of `O3` passes. The optimization pipeline actually applied is the following comma-separated sequence of pass specifications to the LLVM `opt` tool:

1. `module(sroa),module(sccp),module(early-cse)`: These stock LLVM optimizations effectively cleanup the input IR in the X25519 inner loop to be a sequence of arithmetic operations and type casts on plain IR values.

2. `module(function(loop(loop-unroll-full)))`: Loop unrolling is applied most notably for the finite field inversion code which involves a number of small constant trip count loops annotated with `#pragma unroll`.

3. `function(crandall-reduction)`: At this point we invoke our pass. Dependence on the `ReductionAnalysis` is specified in code and handled by the pass manager.

4. `module(verify)`: Verify post pass output, just to be sure we produced valid IR.

5. `default<O3>`: Lastly, invoke the barrage of standard LLVM optimization.

4 Experimental Evaluation

In Section 3 we presented our incremental results in developing targetted optimizations for finite field cryptography in arbitrary bitwidth types. Please refer back for detailed discussion of the individual steps. The table below summarizes the performace progression as our optimizations matured.

Implementation	Time (us)	Iterations	Cycles
Baseline	4682.75	8962	15771906.96
Crandall Single Step	195.29	215216	662345.84
Crandall Multi Step	194.95	215445	661183.93
Incomplete Reduction	93.11	451088	315782.34
Incomplete Reduction over ϕ	58.46	718515	198282.05
`libsodium`	36.80	1141472	124794.13

Starting from a simple implementation of X25519 in Section 2.1, we have achieved a masive 80.1x speedup therefore reaching performance within 1.6 of a real-world optimized library. That's an exciting result!

Nonetheless, from inspecting the generated code there are still clear deficiencies that could be addressed to close the 1.6x gap.

Back-end Improvements. Mid-end optimization can only get so far. In fact, our optimizations have done remarkably well. However, the generated code for multi-precision arithmetic in LLVM's x86 backend is clearly not as good as it could be. Notably it falls back to lowest common denominator instructions. For example LLVM uses the `MUL` instruction rather than `MULX` from the BMI2 extension, which takes arbitrary inputs and does not affect flags. LLVM also does not leverage ADX extensions which permit maintence of two distinct addition carry chains, and were specifically crafted for this kind of cryptographic multi-precision code.

Register Allocation. It appears by inspection that LLVM is spilling registers more than may be required. A plausible hypothesis is that this is a downstream consequence of poor instruction selection and scheduling for the multi-precision arithmetic, therefore spilling may be reduced if we could address the code generation issues.

Implementation Tricks. There are a pleathora of implementation tricks used in hand-tuned optimizations and it appears by inspection that LLVM may not be taking full advantage of them. For example, hand-tuned implementations carefully implement 257-bit additions by "folding the carry" bit

into the next Crandall reduction step using `CMOVC`. Squares are also code generated differently from multiplies. While LLVM is able to observe these tricks to some degree, the results are not as tight as hand-tuned optimizations, leaving a gap that could potentially be closed.

5 Surprises and Lessons Learned

The first surprise of the project came quickly, namely the *poor Clang/LLVM baseline performance*. I expected better performance from Clang/LLVM out-of the box. In retrospect, this is arguably a consequence of a deliberatly simplified implementation of the finite field underlying X25519. This was intentional to produce a cryptographic implementation that was as close to "obviously correct" as you can get with C. However it produces unusual code patterns that trigger fallback code generation in LLVM that is correct but has pathological performance. In this case the `ExpandLargeDivRemLegacyPass` was producing extremely poor looping reduction algorithms. Ultimately the poor baseline performance was not a problem per se, and in fact suggested there would be multiple avenues for performance gains. At the same time it quickly suggested there would be a lot of work to match hand-tuned implementations, and dashed my hopes of beating hand-optimized libraries in the timescale of the project.

My biggest surprise of the project was the *effectiveness of mid-end optimization passes alone* at allowing us to substantially close the gap to hand-optimized code. As noted in Section 4 there are likely substantial remaining gains to be had in the LLVM backend, but my incoming intuition told me much more of the gains would be found there. That said, another motivation for this project was to explore the possibility that LLVM may uncover optimizations missed by hand-tuning, especially at the level of the modular arithmetic expresion graphs targetted by our passes. I think this still remains an open question, but the effectiveness of the mid-end optimizations alone is yet another testament to the design of LLVM's IR and maturity of its optimizer.

The implementation phase of the project presented surprises and challenges that were perhaps a failure to account for *Hofstadter's law*. Initial local optimizations were relatively easy to get right, but reduction analysis and incomplete reduction required far more thought and iteration to refine. Notable complexities were handling ϕ nodes in Incomplete Reduction (Section 3.3), which necessitated the splitting of the range planning and rewriting phases. Furthermore, correct handling of subtraction rewrites was a challenge to debug and resolve. Even the more peripheral aspects of the project took slightly more time to get right than budgeted for, for example establishing the target benchmark with tests and benchmark suite, integrating with external cryptographic libraries, compiling against Google Benchmark with performance counters, establishing a build system for compiling the same program with multiple different versions of custom optimization passes, and so on.

On a similar note, *LLVM's complexity is not to be underestimated* and I perhaps approached this with some hubris. I had intended to implement backend

optimizations as well, and spent quite some time understanding how LLVM's selection DAG was lowering multi-precison arithmetic into X86 instructions. In retrospect this work was interesting but in the project time constraints did not bear fruit into concrete results.

Continuing in the vein of LLVM complexity, it was frequently a problem that LLVM itself would apply optimizations that interfere with the ability of our passes to correctly infer the semantics of the original computation. Therefore, a carefully crafted optimization pipeline was required to achieve the desired effect. A substantial part of the project was understanding LLVM internals, how it was optimizing existing code, and how best to interact within them. Implementating *domain-specific transformations in LLVM* may or may not be a good idea depending on how they interact with the existing pipelines.

Despite surprises and challenges along the way, this project reached a pleasing conclusion. Performance within 1.6 of well-optimized libraries is no mean feat.

6 Future Work

While the 1.6 gap to hand-tuned code is satisfying, performance parity feels within grasp here. Moreover, it would be interesting to futher probe the question of whether there are compiler techniques that could beat hand-tuned optimizations. Specific areas for further work:

Mid-end Optimization. As noted in Evaluation (Section 4), there are more implementation tricks remaining that could likely squeeze out more performance from the mid-end optimization. Some concrete ideas that could be pursued:

- *"Folding the carry."* this trick uses conditional move instructions to conditionally apply a Crandall reduction step when a machine-word-aligned add overflows.
- *Square specialization.* Multi-precision should be more efficient than a regular multiply, since some of the word-by-word multiplies are common. Is LLVM correcty taking advantage of this fact?
- *Machine word alignment.* At present, the IR produced by incomplete reduction still contains multiplies of odd bitwidth types. It may be worth checking whether LLVM performs better if these odd types are rounded up to machine word widths.
- *Multiplication by constants.* X25519 operations end up containing multiplies by fixed constants, for example 121665, and 38 also arises from the Crandall steps. Hand-tuned code often specializes assembly just for these cases. Is there anything that can be transferred to LLVM's code generation for fixed constant multiplies?

- *Slothful Reduction.* The Incomplete Reduction pass already implements partial reduction in order to reduce reduction overhead, however the choice of the limit to reduce to is relatively naive. It would be interesting to gradute the techniques to incorporate the so-called "Slothful Reduction" [21], whereby an entire expression is analyzed to deduce the optimal size.

Expand Benchmark and Testing. The test and benchmark suite should be expanded. The target set of programs with implementations of more elliptic curves, not just the X25519 algorithm. Moreover, a refined set of synthetic microbenchmarks would help to drive optimization efforts. For example, the Explicit Formulas Database [5] would provide an excellent source of the kinds of straight line expression graphs we care about. Tracking average performance over all these graphs would be interesting.

On the other side, if we were to approach performance parity, we'd need to expand the external libraries used for performance comparison. `libsodium` is good but we'd expect that OpenSSL, EverCrypt [19] and others may be incrementally better.

Other Cryptographic Moduli. The work of this project focussed on moduli of Crandall form, which occur frequently in cryptographic code. However there are others that are tackled with an alternative form of optimization known as Montgomery reduction. It is our hope that many of the techniques described here would still be agnostic to the modulus form, for example the Reduction Analysis and the core structure of the Incomplete Reduction pass. Ideally it would be possible to refactor these techiques to support a pluggable implementation of the reduction piece, and therefore support other types of cryptographically relevant moduli. Future work should address this question.

Back-end Optimization. Following on from comments in the Evaluation section (Section 4), it is clear that the generated code from LLVM's backend is sub-optimal. Future work should examine how to improve multi-precision code generation, both by leveraging BMI2 and ADX extensions on X86 platforms that support them, and by improving handing of carry chain scheduling. Back-end optimizations are additionally interesting because they are more likely to transfer to other domains using multi-precision arithmetic than the mid-end optimizations we pursued in this project. The complexity of LLVM's backend meant we could not probe this question in this project, but it will almost certainly have to be tackled to meet performance parity.

Multi-architecture Support. With ARM increasingly relavent in consumer and cloud markets, it would be interesting to pay attention to code generation not just for X86 but also for ARM and other architectures. In particular, it would be interesting to get at the question of what the right

intermediate represetnations are for finite field and multi-precision arithmetic if you want to be able to match hand-tuned code on multiple backend architectures.

Peephole Optimization. Most peak performance finite-field implementations rely on low-level machine-specific instruction scheduling optimizations. While we can no doubt make improvements to the backend, we may end up relying on typically approximate machine models for instruction scheduling. Recent work [14, 7] has shown massive performance improvements with automated program search. It would be interesting to see to what extent we can apply these ideas in a finite-field domain specific compiler, ideally attempting to cut down on the search time required to achieve their optimal results.

Constant-Time Support. A significant caveat of this project, and in fact a well-known problem of writing cryptography in higher-level languages, is that we cannot yet guarantee the compiler will produce constant-time code. Any operation whose timing might depend on secret data is a potential side-channel risk, and the typical way this happens is branching, though variable time arithmetic instructions exist also. Any real application of the techniques in this project would have to grapple with this problem. Working within LLVM is one option, but existing work attempting to add annotations or guaranteed constant-time support to LLVM has not borne fruit given the sheer complexity of the existing project. Constant-time support may be a reason on its own to pursue a domain-specific non-LLVM approach, but there are other requirements that might push us in that direction too.

MLIR Dialect. As covered in the discussion of project surprises (Section 5), at times it felt like fighting with LLVM to enable it to see the semantics of the modular arithmetic computation that was being performed. This is especially true given the restricted set of propagating instructions that the Reduction Analysis pass supports. One approach might be to harden the existing approaches to recognize more code patterns in LLVM IR, but another might be to accept the right approach is custom representations where the *semantics are explicit*. An entirely custom compiler is of course an option that might be necessary, but an MLIR dialect is also an intriguing possibility. One could imagine dialects at different levels: an elliptic curve point dialect lowers to finite field dialect modulo primes, which in turn lowers to a multi-precision arithmetic dialect and finally to LLVM IR. It would not be the first project of its kind, the HEIR project is already implementing a Homomorphic Encryption compiler on top of MLIR [12].

Verification. Finally, verification is increasingly viewed as table-stakes for cryptographic implementations, given the ease of writing bugs and the outsized consequences. Similar to constant-time support, a real production finite field cryptography compiler would likely need to tackle verification in order to be a compelling proposition amongst offerings in modern

cryptography engineering. Valuable future work would be to consider how to develop a domain-specific compiler of this type with verification support. It's likely this would again pull us away from the LLVM and MLIR approaches, but a verified compiler with close to hand-tuned performance would be a holy grail goal.

7 Conclusion

This project has assessed the feasibility of optimized cryptography using non-standard bitwidth types. By tackling a target benchmark of an X25519 implementation in simple C `_BitInt(255)` types, we have shown how domain-specific LLVM analyses and passes can give a massive performance boost and take us to within a reasonable 1.6 gap from well-optimized libraries. The optimizations presented applied Crandall modular reduction techniques, together with Reduction Analysis that allowed us to maintain finite field computations in more efficient forms. Their effect compounded to give an 80.1 speedup over the initial baseline.

While this makes a case for feasibility, there is much more that could be done, and fundamental issues that would need to be addressed if it was to be considered for real-world use. Ultimately, LLVM may not be the right vehicle for this kind of domain-specific compilation, but an optimizing compiler for elliptic curve finite field cryptography remains an exciting prospect.

References

[1] Dmitry Belyavsky et al. "Set It and Forget It! Turnkey ECC for Instant Integration". In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. , Austin, USA, Association for Computing Machinery, 2020, pp. 760–771. ISBN: 9781450388580. DOI: 10.1145/3427228.3427291. URL: https://doi.org/10.1145/3427228.3427291.

[2] Ryan Berger et al. *Formal-Methods-Based Bugfinding for LLVM's AArch64 Backend*. URL: https://blog.regehr.org/archives/2265.

[3] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. URL: https://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf.

[4] Daniel J. Bernstein and Tanja Lange. *Analysis and optimization of elliptic-curve single-scalar multiplication*. Cryptology ePrint Archive, Report 2007/455. 2007. URL: https://eprint.iacr.org/2007/455.

[5] Daniel J. Bernstein and Tanja Lange. *Explicit-Formulas Database*. URL: https://hyperelliptic.org/EFD.

[6] Joppe W. Bos et al. *Montgomery Multiplication Using Vector Instructions*. Cryptology ePrint Archive, Paper 2013/519. 2013. URL: https://eprint.iacr.org/2013/519.

[7] Jay Bosamiya et al. "Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language". In: *Software Verification*. Ed. by Maria Christakis et al. Cham: Springer International Publishing, 2020, pp. 106–123. ISBN: 978-3-030-63618-0.

[8] Niek J. Bouman. *Multiprecision Arithmetic for Cryptology in C++ - Compile-Time Computations and Beating the Performance of Hand-Optimized Assembly at Run-Time*. 2018. arXiv: 1804.07236 [cs.CR]. URL: https://arxiv.org/abs/1804.07236.

[9] Andres Erbsen et al. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005.

[10] Shay Gueron and Vlad Krasnov. *Fast Prime Field Elliptic Curve Cryptography with 256 Bit Primes*. Cryptology ePrint Archive, Paper 2013/816. 2013. URL: https://eprint.iacr.org/2013/816.

[11] Shay Gueron and Vlad Krasnov. "Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures". In: *Arithmetic of Finite Fields*. Ed. by Ferruh Özbudak and Francisco Rodríguez-Henríquez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 119–135. ISBN: 978-3-642-31662-3. URL: https://link.springer.com/chapter/10.1007/978-3-642-31662-3_9.

[12] *HEIR: Homomorphic Encryption Intermediate Representation*. URL: https://heir.dev/.

[13] Erich Keane. *The New Clang _ExtInt Feature Provides Exact Bitwidth Integer Types*. Apr. 2020. URL: https://blog.llvm.org/2020/04/the-new-clang-extint-feature-provides.html.

[14] Joel Kuepper et al. "CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591272. URL: https://doi.org/10.1145/3591272.

[15] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. Jan. 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748.

[16] *libsodium: A modern, portable, easy to use crypto library*. URL: https://libsodium.org/.

[17] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. "Efficient Implementation". In: *Handbook of Applied Cryptography*. CRC Press, 1996. Chap. 14. URL: http://cacr.uwaterloo.ca/hac/about/chap14.pdf.

[18] *Adding a Fundamental Type for N-bit integers*. Proposal. June 2021. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2763.pdf.

[19] Jonathan Protzenko et al. "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114.

[20] Mamy Ratsimbazafy. *constantine: Constant time pairing-based or elliptic curve based cryptography and digital signatures*. URL: https://github.com/mratsim/constantine.

[21] Michael Scott. *Slothful reduction*. Cryptology ePrint Archive, Paper 2017/437. 2017. URL: https://eprint.iacr.org/2017/437.