CryptOpt: Compiling Cryptographic Primitives with High Performance and Formal Assurance

JOEL KUEPPER, University of Adelaide, Australia
ANDRES ERBSEN, Massachusetts Institute of Technology, USA
JASON GROSS, Massachusetts Institute of Technology, USA
CHUYUE SUN, Massachusetts Institute of Technology, USA
DAVID WU, University of Adelaide, Australia
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
CHITCHANOK CHUENGSATIANSUP, University of Adelaide, Australia
DANIEL GENKIN, Georgia Institute of Technology, USA
MARKUS WAGNER, University of Adelaide, Australia
YUVAL YAROM, University of Adelaide, Australia

Most software domains rely on compilers to translate high-level code to multiple different machine languages, with performance not too much worse than what developers would have the patience to write directly in assembly language. However, cryptography has been an exception, where many performance-critical routines have been written directly in assembly (sometimes through metaprogramming layers). Some past work has shown how to do formal verification of that assembly, and other work has shown how to generate C code automatically along with formal proof, but with consequent performance penalties vs. the best-known assembly. We present CryptOpt, the first compilation pipeline that specializes high-level cryptographic functional programs into assembly code significantly faster than what GCC or Clang produce, with mechanized proof (in Coq) whose final theorem statement mentions little beyond the input functional program and the operational semantics of x86-64 assembly. On the optimization side, we apply randomized search through the space of assembly programs, with repeated automatic benchmarking on target CPUs. On the formal-verification side, we connect to the Fiat Cryptography framework (which translates functional programs into C-like IR code) and extend it with a new formally verified program-equivalence checker, incorporating a modest subset of known features of SMT solvers and symbolic-execution engines. We found both final solutions pleasantly simple in the end, and our empirical evaluation shows that they nonetheless produce good results. For instance, for P-256 scalar multiplication, a workhorse routine in Internet standards like TLS, our generated code is 28–38% faster than what GCC or Clang produce from best-known C code, at the same time as our formal guarantees are much stronger.

Additional Key Words and Phrases: elliptic-curve cryptography, public-key cryptography, assembly, performance optimization, search-based software engineering, formal verification

1 INTRODUCTION

Being a foundational pillar of computer security, cryptographic software needs to achieve three often-competing aims. First, being security-critical, the software needs to be correct and protected from implementation attacks. Second, because it is used frequently, it needs to be efficient. Third, for migration to new architectures, the software needs to be portable. Implementations of cryptographic code, therefore, must strike a trade-off between these needs. Implementations that aim for

Authors' addresses: Joel Kuepper, University of Adelaide, Australia, joel.kuepper@adelaide.edu.au; Andres Erbsen, Massachusetts Institute of Technology, USA; Jason Gross, Massachusetts Institute of Technology, USA; Chuyue Sun, Massachusetts Institute of Technology, USA; David Wu, University of Adelaide, Australia; Adam Chlipala, Massachusetts Institute of Technology, USA; Chitchanok Chuengsatiansup, University of Adelaide, Australia; Daniel Genkin, Georgia Institute of Technology, USA; Markus Wagner, University of Adelaide, Australia; Yuval Yarom, University of Adelaide, Australia.

2022. XXXX-XXXX/2022/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

portability tend to use high-level languages, such as Java or C. These allow for easy maintenance and are essentially platform-independent, assuming the existence of suitable development tools like compilers and assemblers.

At the same time, compiler-based code generation can be a double-edged sword. First, compiler-produced cryptographic code tends to underperform when compared to hand-optimized code [Bernstein et al. 2013, 2014a,b, 2017; Chou 2015, 2016; Chuengsatiansup et al. 2013; Chuengsatiansup and Stehlé 2019; Kannwischer et al. 2019], typically written directly in the platform's assembly language. Beyond slower performance, compilers are typically not designed for maintaining security properties. In particular, compilation bugs could result in incorrect code [Erbsen et al. 2019, Appendix A], while overly aggressive optimizations may even strip side-channel protections [Barthe et al. 2018; D'Silva et al. 2015; Kaufmann et al. 2016].

We note that the difficulties compilers have when operating over cryptographic code are not caused by high code complexity. In fact, cryptographic code tends to be simpler than a typical program code, due to its avoidance of data-dependent control flows and memory-access patterns for reasons of side-channel resistance. Ironically, compiler optimization passes often focus on optimizing control flow, as it offers a bigger impact than fine-tuning straight-line code [Aho et al. 1986].

Instead, the cause is that such code tends to be simpler than a typical program code, and this simplicity deprives the compiler of optimization opportunities. At the same time, we observe that such simplicity may offer opportunities for using strategies not commonly exploited by compilers, such as reordering arithmetic operations within a basic block or exchanging machine instructions with semantically equivalent machine instructions. Thus, in this work we ask the following question:

How can we exploit the simplicity of cryptographic primitives in order to generate efficient and provably correct implementations of cryptographic functions?

Our Contribution

We present CryptOpt, a new code generator that produces highly efficient code tailored to the architecture it runs on. The task is split between *finding* performant program variants and *checking* that they have preserved program behavior. The former works via random search, and the latter works via a formally verified program-equivalence checker that should be applicable even with other optimization strategies. As a result, the random-search procedures need not be trusted, and, when we compose them with the Fiat Cryptography Coq-verified compiler [Erbsen et al. 2019] and our new equivalence checker, we get end-to-end functional-correctness proofs for fast assembly code – the fastest yet demonstrated for the cryptographic algorithms we study, when compiling automatically from high-level programs as we do.

To *find* performant machine-code variants, instead of relying on human heuristics, CryptOpt represents the code-generation task as a combinatorial optimization problem. That is, CryptOpt considers a solution space that consists of machine-code implementations of the target function and uses techniques from the domain of randomized search heuristics to search for the best-performing implementation.

To optimize, CryptOpt first chooses an arbitrary correct implementation of the target function. It then mutates the implementation by either changing the instruction(s) that implements a certain operation or changing the order of operations. If the mutated implementation outperforms the original, the mutated implementation is kept; otherwise it is discarded. Repeating this process iteratively improves the retained implementation, allowing CryptOpt to find an efficient implementation.

An enabling feature of CryptOpt is the simplicity of cryptographic code. Instead of supporting a wide array of programming constructs, CryptOpt only optimizes straight-line code that does not contain any control flow. In such code, ordering constraints are solely determined from the data flow. While restricted, we note that code meeting the required format is quite common in cryptographic implementations (e.g., the IR of Fiat Cryptography enforces this restriction syntactically).

Instead of trying to predict code performance, CryptOpt measures the actual execution time. This approach is important because it avoids the inaccuracies inherent in hardware models and allows CryptOpt to tailor the produced code to the target processor while treating the processor itself as an opaque unit.

To *check* that generated code is correct, CryptOpt integrates with Fiat Cryptography [Erbsen et al. 2019]. That framework, implemented in Coq, already generates low-level IR programs proven to preserve behavior of high-level functional programs, and it has been adopted by all major web browsers and mobile platforms for small but important parts of their TLS implementations, so there is high potential for impact improving performance further without sacrificing formal rigor. CryptOpt begins with Fiat Cryptography IR programs and generates x86-64 assembly code. To avoid needing to model the random-search process, we instead built and proved an equivalence checker, which can compare programs across Fiat Cryptography IR and x86-64 assembly. Its two main pieces implement modest subsets of features known from SMT solvers and symbolic-execution engines. From SMT solvers, we take an E-graph data structure to canonicalize logical expressions via rewrite rules, in a compact data structure with significant sharing. From symbolic-execution engines, we take maintenance of symbolic states that tie registers and symbolic memory locations to logical expressions known to the solver. Thanks to their combined proof in Coq, none of these details need to be trusted to believe compilation is sound.

We evaluate CryptOpt in two case studies for cryptographic arithmetic:

- In the first, we feed CryptOpt from Fiat Cryptography. Using nine different primes as input to Fiat Cryptography, we obtain a speedup of 1.63 on geometric means compared to GCC 11 (speedup 1.28 against Clang 13), across eight different x86-64 platforms (three AMD, five Intel). This x86-64 assembly code has end-to-end Coq proof.
- In the second case study, we create an input function from the C code of `libsecp256k1` [Bitcoin Core 2022], feed it into CryptOpt, and with the generated x86-64 assembly obtain an average speedup of 1.05 against the hand-optimized assembly code in `libsecp256k1` (and a speedup of 9.03 if against OpenSSL's prime-agnostic implementation). For this second case study, the verified equivalence checker lags behind in functionality and is not used to validate the results of random search.

Summary of Contributions. In summary, we make the following contributions in this paper:

- We present CryptOpt, a code generator that relies on combinatorial optimization instead of compiler heuristics for producing highly efficient code (Section 3).
- We demonstrate that a relatively modest Fiat Cryptography extension (Section 4) suffices to enable integration with arbitrary backend compiler heuristics. We implemented and verified Coq functional programs with a minimal subset of well-known features from SMT solvers and symbolic-execution engines, leading to a single extractable compiler that checks assembly files for semantic equivalence with high-level functional programs. To our knowledge, this is the first such equivalence checker with mechanized proof from first principles, and our empirical results show which features of SMT solvers were essential and which could be omitted.
- We generate formally verified high-performance cryptographic code optimized for eight CPU architectures, obtaining considerable speedups over GCC and Clang (Section 5).

At the level of detail we have presented so far, there are important similarities with the work of Bosamiya et al. [2020], who also combine random search through assembly programs with formal methods for cryptography. However, their starting point is handwritten assembly code within a relatively shallow metaprogramming framework – the inputs to their tooling literally dictate expected assembly code in performance-critical inner loops. Genetic search then goes on to find superior alternatives for some of those instructions. We show instead that random search can be used as part of a fully automated pipeline that starts from high-level functional programs, allowing us to generate fast code for multiple elliptic curves, not just multiple target architectures from a single algorithm for Bosamiya et al. We also deal with elliptic-curve arithmetic, raising issues of automatic register allocation, spilling, instruction selection (e.g., multiple addition instructions with different consequences for processor pipelines), and reasoning modulo associativity and commutativity, none of which show up in their AES-GCM case study (which naturally leans more on bitwise operations). Finally, our approach has a smaller trusted code base, being built within a proof assistant (Coq) rather than more specialized formal-methods tools (F^* and Z3).

We intend to release our implementation as open source, and it is attached to this submission as a code supplement.

2 OVERVIEW

Let us give the basic background on the problem we solve and sketch our solution, before devoting the following few sections to more detail.

2.1 Finite-Field Arithmetic for Cryptography

We focus on elliptic-curve cryptography (ECC), which is used widely in Internet standards like TLS. It involves certain geometric aspects that are orthogonal to our tooling, which supports arithmetic modulo large prime numbers, otherwise known as finite-field arithmetic (FFA).

Besides its use in ECC, FFA is also used in message-authentication schemes like Poly1305 [Bernstein 2005] and new schemes like Supersingular Isogeny Key Encapsulation (SIKE) [Azarderakhsh et al. 2019] used in post-quantum cryptography. Because the FFA is a performance-critical component in the implementations, many tend to implement it by hand. E.g. targeting different architectures, the ubiquitous cryptographic library OpenSSL [OpenSSL 2022] has many hand-optimized implementations for Curve25519 and NIST P-256, which are both well-known instantiations of ECC. The Bitcoin blockchain uses yet another Curve called secp256k1 for their block signatures. Its core library `libsecp256k1` contains hand-optimized code for the field arithmetic as well as a C implementation used as a fallback in architectures for which no optimized version exists.

FFA is not trivial to implement. The size of an element in a finite field suitable for cryptographic schemes is typically larger than the common size of a computer register. While this consequence follows from the need for security of cryptographic schemes, it implies that field arithmetic cannot be performed directly by the underlying hardware and is thus typically handled in software. In particular, a field element is typically represented using several CPU registers, and thus every field operation requires multiple CPU cycles.

In particular, we note that those implementations tend to be straight-light code, heavy on arithmetic rather than control flow. As typical compilers tend to optimize control flow rather than straight-line code, they usually fail to optimize FFA very well, compared to what experts know how to write by hand in assembly. Specifically, those experts are doing clever simultaneous instruction selection, instruction scheduling, and register allocation, which, going well beyond capabilities of off-the-shelf C compilers, should take into account microarchitecture details such as macro-op fusion [Celio et al. 2016; Ronen et al. 2004], cache prediction [Hooker and Eddy 2013; Lepak and Lipasti 2000; Subramaniam and Loh 2006], cache-replacement policies [Vila et al. 2020], and other (potentially undocumented) microarchitectural choices.

```
Definition wwmm_step A B k S ret :=
  divmod_r_cps A (λ '(A, T1),
  @scmul_cps r _ T1 B _ (λ aB,
  @add_cps r _ S aB _ (λ S,
  divmod_r_cps S (λ '(_, s),
  mul_split_cps' r s k (λ '(q, _),
  @scmul_cps r _ q M _ (λ qM,
  @add_longer_cps r _ S qM _ (λ S,
  divmod_r_cps S (λ '(S, _),
  @drop_high_cps (S R_numlimbs) S _ (λ S,
  ret (A, S) ))))))))).
Fixpoint wwmm_loop A B k len_A ret :=
  match len_A with
  | O => ret
  | S len_A' => λ '(A, S),
    wwmm_step A B k S (wwmm_loop A B k len_A' ret)
  end.
```

(a) Input functional program

```
void wwmm_p256(u64 out[4], u64 x[4], u64 y[4]) {
  u64 x17, x18 = mulx_u64(x[0], y[0]);
  u64 x20, x21 = mulx_u64(x[0], y[1]);
  u64 x23, x24 = mulx_u64(x[0], y[2]);
  u64 x26, x27 = mulx_u64(x[0], y[3]);
  u64 x29, u8 x30 = addcarryx_u64(0x0, x18, x20);
  u64 x32, u8 x33 = addcarryx_u64(x30, x21, x23);
  u64 x35, u8 x36 = addcarryx_u64(x33, x24, x26);
  u64 x38, u8 _ = addcarryx_u64(0x0, x36, x27);
  u64 x41, x42 = mulx_u64(x17, 0xffffffffffffffff);
  u64 x44, x45 = mulx_u64(x17, 0xffffffff);
  u64 x47, x48 = mulx_u64(x17, 0xffffffff00000001);
  u64 x50, u8 x51 = addcarryx_u64(0x0, x42, x44);
  // 100 more lines...
```

(b) Output C code

Fig. 1. Word-by-word Montgomery Multiplication, output specialized to P-256 ($2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$)

2.2 Our Starting Point: Fiat Cryptography

Erbsen et al. [2019] present their Fiat Cryptography framework, which translates descriptions of field arithmetic into code with a Coq proof of functional correctness. The starting point is a library of functional programs giving generic implementations of common implementation strategies (word-by-word Montgomery and Solinas) for field arithmetic. These functional programs are proved correct once and for all. A specific code-generation run selects one of the functional programs and specializes its parameters (e.g. field size) for a particular prime, resulting in an intermediate representation (Fiat IR). Fiat Cryptography then generates provably correct code in various languages (C, Java, Zig) from Fiat IR.

Figure 1, taken from Erbsen et al. [2019], demonstrates the end-to-end behavior of Fiat Cryptography. Figure 1a shows a representative input, a high-level functional program in Coq's language Gallina. It implements an arithmetic algorithm, parametric in the prime modulus of arithmetic. An invocation of the Fiat Cryptography compiler specifies which prime to specialize the code to. A variety of formally verified phases then compose to produce code in output languages like C, as shown in Figure 1b. Off-the-shelf compilers were then applied to this code, generating assembly with a noteworthy performance gap vs. what experts know how to write by hand.

The earlier, verified compilation goes via the Fiat IR, which has translators to various backend languages. Our goal with CryptOpt was to follow the same philosophy of automated compilation alongside formal guarantees, so we chose to build CryptOpt to accept Fiat IR programs as inputs.

We will demonstrate how, compared to the released Fiat Cryptography tooling, we both improved performance and decreased the trusted code base, with theorems bottoming out in the operational semantics of x86-64 assembly instead of Fiat IR, removing compilers like GCC from the picture altogether (specifically not requiring us to trust in their correctness).

2.3 Our Solution

Section 2.3 diagrams CryptOpt, with connections to preexisting tools. CryptOpt ingests the code of a function in the Fiat Cryptography IR and produces x86-64 assembly code. The CryptOpt optimizer then mutates the current x86-64 assembly code, measuring execution time of both the original and the mutated version. It then discards the slower one and iteratively continues this process until a predefined computational budget is used up (depicted in the middle part of Section 2.3).

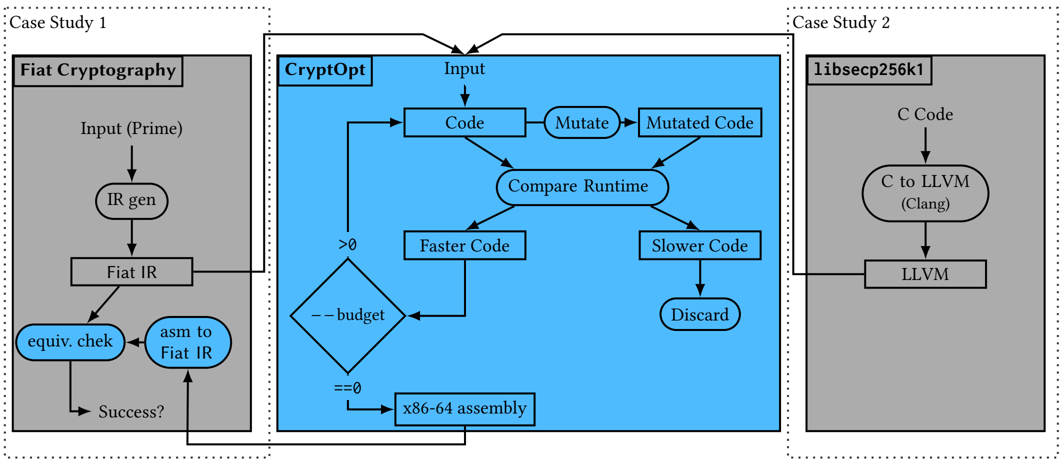


Fig. 2. CryptOpt system architecture

The random optimizer was codesigned with a separate program-equivalence checker, which, given an original IR program and its optimized assembly, is able to verify behavior preservation with no further hints. Thanks to this checker, the optimizer itself need not be trusted. The checker itself is implemented and verified in Coq, and we compose it with the original Fiat Cryptography tooling to extract a command-line tool that takes as input a functional program, the parameters to specialize it to, and a claimed assembly implementation. The classic Fiat Cryptography compiler is run, and its resulting IR is equivalence-checked against the provided assembly. This command-line tool has an end-to-end Coq theorem establishing that the assembly truly implements the original functional program (when a command-line invocation outputs "success"). This usage mode forms "Case Study 1" in Section 2.3.

To show CryptOpt's flexibility, we conducted another case study ("Case Study 2" in Section 2.3). `libsecp256k1` provides two versions of the field arithmetic: a hand-optimized x86-64 assembly version and a C fallback. We transformed the C fallback to LLVM using Clang. Then, we created a simple LLVM-to-CryptOpt script to transform the necessary operations in LLVM to the input format for CryptOpt. Note that neither the hand-optimized x86-64 assembly nor the C fallback is formally verified, and thus this case study skips the formal-verification aspect, focusing just on further evaluation of the random optimizer.

Now we are ready to fill in the details of CryptOpt (random search in Section 3 and equivalence checking in Section 4) and how we evaluated it (Section 5).

3 RANDOM SEARCH FOR ASSEMBLY PROGRAMS

The first half of our approach is random search for improvements to assembly programs that implement Fiat IR programs. We draw on ideas from the mature field of random search, perhaps surprisingly requiring only some of the simpler and more intuitive elements of that toolbox, which we introduce as they arise in this section.

3.1 Input Format

Recall (Section 2.3) that CryptOpt takes Fiat IR as an input. Available operations include arithmetic (`add`, `compare`, `multiply`, `negate`, `subtract`), Boolean logic (`and`, `not`, `or`, `rotate`, `shift`, `xor`),

Algorithm 1: An Example Function

```
input  : $X, Y, Z$ such that $0 \leq X, Y, Z < 2^{63}$
output : $O = 2^{64}O_1 + O_0 = X \cdot (Y + Z) + Z^2 + Y$
function example(X, Y, Z)
begin
    $t_2,\ t_1 \leftarrow \textsc{Mul}_1(Z, Z)$
    $c_\varnothing,\ t_0 \leftarrow \textsc{Add}_1(Y,\ Z,\ 0)$
    $t_4,\ t_3 \leftarrow \textsc{Mul}_2(t_0, X)$
    $c_0,\ t_5 \leftarrow \textsc{Add}_2(t_3,\ t_1,\ 0)$
    $c_1, O_0 \leftarrow \textsc{Add}_3(t_5,\ Y,\ 0)$
    $c_\varnothing,\ t_6 \leftarrow \textsc{Add}_4(t_4,\ t_2, c_0)$
    $c_\varnothing, O_1 \leftarrow \textsc{Add}_5(t_6,\ 0, c_1)$
    return $O_1,\ O_0$
end
```

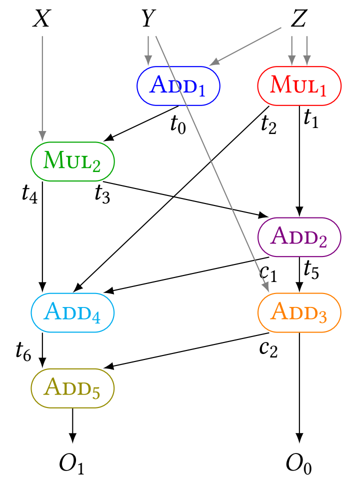


Fig. 3. Data flow of the running example in Algorithm 1

and assignment (`conditional_move`, `move`). The list of supported operations is restricted so that CryptOpt only supports straight-line code with no branches. Input programs contain no explicit memory accesses, just accesses of local variables. As a result, generated assembly programs will not be too much more complex, avoiding all memory aliasing and restricting pointer expressions to be constant offsets of either function parameters (for data structures passed by reference) or the stack pointer (for spilled variables). Restricting the code we optimize as stated above enables an efficient search of the solution space. Moreover, these restrictions facilitate the formal verification of CryptOpt's output in terms of both correctness and side-channel resistance.

Algorithm 1 shows a running example of a Fiat IR program, which we will use throughout this paper. The program takes three inputs (X, Y, Z) and outputs $X \cdot (Y + Z) + Z^2 + Y$ using two types of operations: Add and Mul. The operation Add adds two 64-bit numbers and one 1-bit carry, then returns the sum as one 64-bit number and one 1-bit carry. The operation Mul multiplies two 64-bit numbers, then returns the 128-bit product as two 64-bit words. For simplicity, we assume that the function arguments are in the range $0 \leq X,\ Y,\ Z < 2^{63}$, allowing us to ignore some carries known to be 0. We mark these carries with $c_\varnothing$.

In the absence of memory aliasing and control flow, the restrictions on operation evaluation order in the programs we optimize are captured by the data flow. Figure 3 shows the data flow graph of the code in Algorithm 1. Nodes represent the operations executed in the code, whereas arcs correspond to data dependencies between the operations.

3.2 Optimization Strategy

A unique feature of CryptOpt is that instead of relying on heuristics for generating the code, CryptOpt explicitly casts the problem as a combinatorial optimization problem. That is, CryptOpt searches the space of machine-code sequences that implement the input function, looking for the best-performing implementation simply by utilizing the execution time as the objective function. For optimization, we employ the random local search (RLS) strategy [Auger and Doerr 2011; Doerr and Neumann 2019]. RLS is often an efficient and effective baseline for finding and analyzing local optima. Each run of RLS aims to find a local optimum by iteratively moving from one solution to a random neighbor (i.e. mutation) given a defined variation operator. This move happens if that new neighbor performs at least as well.

RLS is often highly sensitive to the initial conditions. To address such erratic behavior, the simple Bet-and-Run heuristic [Fischetti and Monaci 2014; Weise et al. 2019] turns the sensitivity to the initial conditions into an advantage by employing multiple runs. A typical use of Bet-and-Run starts with multiple independent runs of RLS, each optimizing for a predefined number of mutations. After this initialization step, the algorithm selects the best run and lets this run continue optimizing from that step, stopping when a total number of mutations is explored. We adopted Bet-and-Run in CryptOpt, i.e. search begins in parallel on several random assembly realizations of the input Fiat IR program, switching at some point to focus on the most promising one.

3.3 Code Generation

As for the process whereby those assembly candidates are generated, it can be understood as split between instruction scheduling, instruction selection, and register allocation, in that order. Each phase makes certain arbitrary decisions that may be reversed by later random mutation. We summarize the phases here before returning to treat in detail how they operate and which random mutations are relevant to their decisions.

First, for instruction scheduling (see Section 3.4), dataflow analysis determines dependency structure across instructions, at which a random topological sort of that graph is chosen to use in the assembly. Then, for instruction selection (see Section 3.5), for each high-level operation, we choose among the compatible x86-64 assembly instructions. Finally, in our setting, register allocation (see Section 3.6) arises mostly in ensuring compatibility with operand restrictions of the instructions that were selected. For example, consider the objective to *multiply two values* (one single instruction can at most read from one memory location), in a context where both operands reside in memory. The relevant dimension of freedom is which value to load into a register explicitly and into which register. After the decisions of the three phases have been recorded, it is easy to read off the chosen assembly program sequentially. Recall that all of these decisions may be revisited later in random mutations.

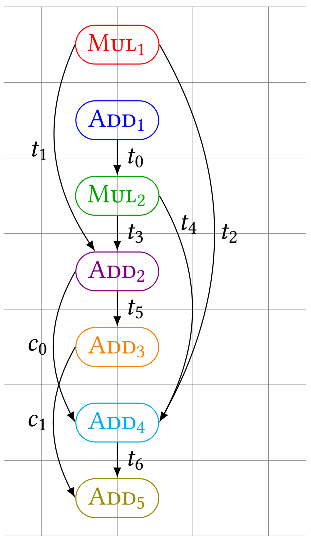
To present details of the three phases, we focus on the example program from Algorithm 1. Figure 4 shows how operations are initially ordered and potential scheduling mutations; the effects of mutations are shown in Figure 5; until finally we see examples of the emitted code in Figure 6. We will reference those illustrations later as needed.

3.4 Instruction Scheduling

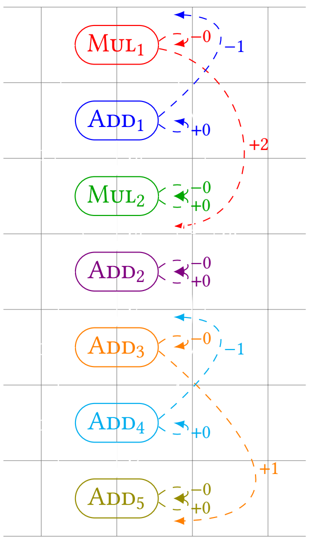
Thanks to the simplicity of Fiat IR, every variable is assigned exactly once in the straight-line programs that CryptOpt takes as input. Consequently, any operation can be evaluated whenever all its inputs have been computed. In the example, $\textsc{Mul}_2$ can be evaluated when t_0 and X are computed (X, being the input, is always in memory and therefore available).

Initial Ordering. To create an initial ordering, CryptOpt simply computes a topological sort of the dataflow graph. Figure 4a shows an example of an initial ordering that could be produced for our running example. We would like to emphasize that the selection does not rely on any heuristic. While this initial selection does affect the subsequent mutated ordering, our mutation strategy guarantees that any possible ordering can be reached from any initial random starting point via a sequence of mutation steps. Hence, we ensure that, at least theoretically, we can achieve the optimal ordering irrespective of the initial ordering.

Mutation Step. An instruction-scheduling mutation step of CryptOpt randomly selects one operation in the current ordering. This operation is moved to a randomly chosen location within



(a) Black arrows indicate creation and consumption of intermediate values. (X, Y, Z and O_1, O_2 omitted for clarity.)



(b) Dashed arrows indicate the possible movements of each operation. −1 and +2 mean that an operation can be moved one slot backwards and two slots forwards respectively, relative to its current position.

Fig. 4. One ordering. Round rectangles show the operations, which are evaluated top-down.

the interval where it would be valid to move: not before the last assignment responsible for setting a variable that is used here as an operand, nor after the first assignment that reads the variable being set here. Starting with the initial ordering shown in Figure 4a, the different mutation candidates are illustrated in Figure 4b, where we show the bounds of possible movements of each operation. For example, looking at ADD$_4$ we see that it can move up one position, because it does not depend on the output of ADD$_3$. However, ADD$_4$ cannot move down because the very following operation, ADD$_5$, uses its output t_6. Step β of Figure 5 shows the effect of moving ADD$_4$ up one position.

Selecting the position to move an operation to is biased towards larger moves, i.e. further away from the initial position. The rationale for this bias is that such moves reduce the distance between computing a value and using it, which reduces the likelihood that the value will be spilled to memory.

3.5 Instruction Selection

An important property of complex instruction sets such as x86-64 is that there can be multiple alternative implementations for each high-level operation, each with a slightly different semantics and impact on the processor pipeline. To match machine instructions to operations, CryptOpt uses templates describing the possible implementations for each operation.

Consider, for example, the ADD operation. This can be implemented using multiple instructions, such as `add`, `adcx`, or `adc`. While semantically these choices are equivalent, the specific choice depends on the current state of the program. For instance, in the case of no carry, implementing an ADD operation using an `adcx` instruction requires clearing the carry flag. The template of the `adcx` instruction accounts for that and issues a clear-carry instruction (`clc`) before the `adcx` instruction,

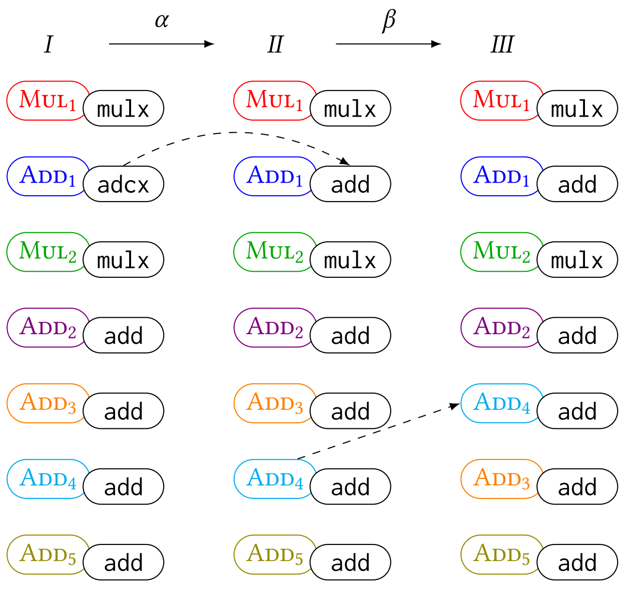


Fig. 5. I: Initial ordering of operations (colored rounded rectangles) with attached templates (black rounded rectangles); II-III: Two mutations α and β: mutation α in instruction-template selection, mutation β in topological ordering. Data-flow arrows omitted; dashed arrows indicate mutations.

unless CryptOpt determines that the carry is already clear. Lines 7 and 8 of Figure 6 show this choice for Add$_1$.

Initial Template Mapping. When creating the initial code, CryptOpt selects a random template for each operation as an initial mapping. Figure 5 (I) shows an example of a possible mapping of templates to the operations of our example function. The figure indicates the operation Mul$_1$ is implemented using the `mulx`, Add$_1$ uses `adcx`, and so forth.

Mutation Step. To mutate the instruction selection, CryptOpt chooses an operation at random and replaces the template for that instruction with one of the possible alternatives. Mutation α in Figure 5 shows an example of replacing the template for Add$_1$ to use the `add` instruction instead of the original `adcx`.

Flag Spills. The series of additions Add$_2, \ldots,$ Add$_5$ show not only the influence of different orderings on the resulting code but also the mechanism of how a template is used to handle flag spills. Consider the code in Figure 6a where Add$_2$ results in `CF`=c_0. Later on, Add$_3$ needs to write its own `CF`=c_1. Therefore, the current `CF` needs to be spilled (into another register). In this case, it does so with `setc dl` (line 18). Similarly, Add$_4$ then needs to spill c_1 (line 23) to avoid overwriting it with its $c_\varnothing$.

At this point (line 25), CryptOpt needs to add the values of three operands, i.e. $t_4 + t_2 + c_0$. As the x86-64 assembly language does not have a single three-operand addition instruction, CryptOpt first adds two operands together then adds the sum to the third one. Note that this changes the evaluation order from $t_4 + t_2 + c_0$ to $(t_2 + c_0) + t_4$. As the Add operation is commutative, any evaluation order maintains correctness. The equivalence checker accounts for this change in evaluation order (see Section 4.4).

```
 1 | ; t_2, t_1 ← Mul_1(Z, Z)
 2 | mov   rdx,   [Z]
 3 | mulx  r8,    r9,   [Z]
 4 |
 5 | ; c_∅, t_0 ← Add_1(Y, Z, 0)
 6 | mov   rdx,   [Y]
 7 | clc
 8 | adcx  rdx,   [Z]
 9 |
10 | ; t_4, t_3 ← Mul_2(t_0, X)
11 | mov   [rsp], r8          ; spill
12 | mulx  r8,    rdx,  [X]
13 |
14 | ; c_0, t_5 ← Add_2(t_3, t_1, 0)
15 | add   r9,    rdx
16 |
17 | ; c_1, O_0 ← Add_3(t_5, Y, 0)
18 | setc  dl                 ; dl ← c_0
19 | add   r9,    [Y]
20 | mov   [O_0], r9
21 |
22 | ; c_∅, t_6 ← Add_4(t_4, t_2, c_0)
23 | setc  r9b                ; r9b ← c_1
24 | movzx rdx,   dl
25 | add   rdx,   [rsp]
26 | add   r8,    rdx
27 |
28 | ; c_∅, O_1 ← Add_5(t_6, 0, c_1)
29 | movzx r9,    r9b
30 | add   r8,    r9
31 | mov   [O_1], r8
```

(a) Code from *I* (initial code)

```
| ; t_2, t_1 ← Mul_1(Z, Z)
| mov   rdx,   [Z]
| mulx  r8,    r9,   [Z]
|
| ; c_∅, t_0 ← Add_1(Y, Z, 0)
| mov   rdx,   [Y]
|
| add   rdx,   [Z]
|
| ; t_4, t_3 ← Mul_2(t_0, X)
| mov   [rsp], r8          ; spill
| mulx  r8,    rdx,  [X]
|
| ; c_0, t_5 ← Add_2(t_3, t_1, 0)
| add   rdx,   r9
|
| ; c_∅, t_6 ← Add_4(t_4, t_2, c_0)
|
| adc   r8,    [rsp]
|
|
|
| ; c_1, O_0 ← Add_3(t_5, Y, 0)
|
| add   rdx,   [Y]
| mov   [O_0], rdx
|
| ; c_∅, O_1 ← Add_5(t_6, 0, c_1)
|
| adc   r8,    0
| mov   [O_1], r8
```

(b) Code from *III* (code after two mutations)

Fig. 6. Emitted assembly code. Highlighted lines (7-8, 18-19, 23-26, 29-30) show the effects from the mutations.

3.6 Register Allocation

The register allocation step of CryptOpt achieves two aims. It first needs to decide which values are assigned to registers, and then determine which registers to spill to memory when running out of registers. CryptOpt uses random search for the former, whereas the latter is determined using a deterministic strategy.

Register Assignment. Registers need to be assigned in two main cases: when computing a new value and when reading a value from memory, either from the input or following a register spill. CryptOpt keeps track of the live registers, allocating a free register if one is available. If none is available, CryptOpt spills the contents of a register to memory, and uses the freed register. To choose the register to spill, CryptOpt scans the future use of all register and splills the register whose next use is furthest based on the current operations order.

For example, lines 11–12 in Figure 6 implement the Mul$_2$ operation. (For this example, we assume that the architecture only has three general purpose registers: `r8`, `r9`, and `rdx`.) At this point, registers `r8`, `r9`, `rdx` have been used for t_2, t_1, and t_0, respectively. Hence, we need to spill a register. Observing that the t_2 is not required until Add$_4$, whereas t_0 and t_1 are used earlier, CryptOpt spills `r8` (Line 11).

Memory Loads. Most arithmetic operations in the x86-64 architecture support instruction formats that take one argument from memory. When an argument of an operation is in memory, CryptOpt tries to use such an instruction format. When this is not possible, e.g. when the values of two arguments are in memory, CryptOpt resorts to loading a value from memory into a register. In the case of associative operations, such as addition and multiplication, CryptOpt initially randomly chooses the value to load. During the optimization, a mutation step may change the decision of which value to load.

Exploiting Simplicity. Finally, we note that the absence of control flow and avoidance of human heuristics are key enablers for memory spill decisions. The former simplifies dependency analysis, allowing CryptOpt to determine the requirements for downstream operations. The latter allows CryptOpt to examine the entire function rather than focusing on instructions within a peephole window, a commonly used technique by off-the-shelf compilers.

3.7 Cost-Function Evaluation

We compare different random program variants by running them on the actual processors of interest. If there is too much random noise in running-time measurement, random search could be driven into unproductive oscillation. We found the details of such measurement surprisingly difficult to get right. Appendix A gives those details, which rely on running a chosen number of repetitions of the two candidate programs, interleaved in a random order, then returning the median timing observed per program.

4 CHECKING PROGRAM EQUIVALENCE

Our goal with CryptOpt was to preserve or even strengthen the formal guarantees of Fiat Cryptography, which provides Coq proof of behavior preservation. One way to achieve that goal would have been to verify the whole random-search process with Coq, but we wanted to find a simpler strategy that would have the side benefit of also potentially supporting automatic verification of various handwritten assembly solutions. Therefore, we decided to write a program-equivalence checker in Coq and verify it. Industrial-strength translation validation as in Alive [Lopes et al. 2021] is now well-established, but again, proving such a tool from first principles would be a substantial undertaking. We were curious, instead, how far we could get implementing (and proving) our checker from scratch, lifting just those features of more conventional checkers that turned out to be important in our domain. This section of the paper may be most interesting for sharing a small feature set that we validated empirically as flexible enough.

4.1 Background

One way to avoid bugs in compiler tooling is to rely on general, reusable frameworks like OpenTuner [Ansel et al. 2014], which abstracts out concerns of smart searching of design spaces. Another useful ingredient is careful validation of ingredients like performance models, as in work by Abel and Reineke [2021]. However, bugs in these tools could easily compromise soundness.

Formally proved compilers are the gold standard to address this concern. For instance, CompCert [Leroy 2009; Leroy et al. 2016] was proved as a correct C compiler using the Coq theorem prover, which we also rely on. However, proving a whole compiler can be very labor-intensive, and thus it is often appealing to prove a *checker* for compiler outputs, known as a translation validator. For instance, CompCert was extended in that way [Tristan and Leroy 2008]. To date, however, the formally proved translation validators have not incorporated reasoning with algebraic properties of arithmetic, as we found we needed in CryptOpt.

In contrast, translation validation with SMT solvers uses rich reasoning in order to prove equivalence between the high-level input program and the obtained low-level output. The Alive project [Lopes et al. 2021] for LLVM is a good example. Compared to work with formally proved translation validators, Alive and similar tools include much larger trusted bases, for instance including a full SMT solver like Z3 [de Moura and Bjørner 2008]. Some SMT solvers have been extended to produce proofs that can be checked in more foundational tools like Coq, as in SMTCoq [Armand et al. 2011].

In our experience, these tools hit performance bottlenecks when working with large bitvectors. Even if we imagine those issues as fixed some day, there are still benefits to creating a customized checker, keeping just the relevant aspects of SMT. A benefit of a slimmed-down custom prover is that it becomes more feasible to prove the prover itself, rather than just a checker for its outputs, which improves performance and reduces surprise from proof-generation bugs.

4.2 Code Verification

The best-performing implementation produced by CryptOpt is assured to be correct through a formally verified equivalence checker. Specifically, we verify the correctness of CryptOpt's output through functional equivalence between programs in the Fiat IR and x86-64 assembly. We developed simple symbolic-execution engines for the relevant subsets of both languages, producing program-state descriptions in a common logical format. The next task is to check that function-output registers and memory locations store provably equivalent values between the two programs. To that end, we developed a simple equivalence theorem prover that borrows from SMT solvers, using a similar (E-graph) data structure but retaining just the complexities needed for this domain of straight-line code. The components of equivalence checking are written in Coq and thus can be composed with the existing Fiat Cryptography compilation pipeline. Furthermore, we prove the correctness of our verifier against Fiat Cryptography's IR semantics and a simple model of straight-line x86-64 assembly.

We built a general equivalence checker for programs in the Fiat IR and x86-64 assembly. The equivalence checker is a functional program in the Coq proof assistant and can be compiled (via standard automatic translation to OCaml), along with the rest of the Fiat Cryptography pipeline, into a new command-line compiler that validates x86-64 assembly files against high-level cryptographic algorithms (specialized to particular parameters).

Our implementation is split into two main pieces: an equivalence checker for purely functional expressions and a pair of symbolic-execution engines that call the expression checker. The expression-level support is inspired by the E-graph data structure of SMT solvers [Detlefs et al. 2005], which in general helps find equalities between different symbolic expressions. For our purposes, its public API is based on the following definition of expressions:

$$\begin{array}{rlll}
\text{Integer constants} & n & & \\
\text{Variables} & x & & \\
\text{Operators} & o & & \\
\text{Expressions} & e & ::= & n \mid x \mid o(e, \ldots, e)
\end{array}$$

The E-graph exposes a function `internalize` that takes in an expression and returns a variable now associated with that expression's value. Importantly, the expression will usually mention variables that came out of previous calls, which lets us work with exponentially more compact representations than if we expanded out all variables. The internal E-graph takes advantage of this sharing for efficiency. Also, crucially, every `internalize` invocation proactively infers equalities between previously considered expressions and the new expression and its subterms. Thus, we may check two expressions for equality simply by verifying that `internalize` maps them to the same variable, which becomes a chosen representative of an equivalence class of expressions.

4.3 Symbolic Execution

We built symbolic-execution engines for the two relevant languages, Fiat IR and x86-64 assembly.

The engine for the IR is simpler than for x86-64 assembly. Programs in this IR are just purely functional sequences of variable assignments with expressions that effectively already match the grammar we just gave. Thus, to evaluate such a program symbolically, we just maintain a dictionary associating program variables to logical variables; the latter are effectively handles into the E-graph. We step through a program in order, looking at each assignment. The righthand-side expression of an assignment is rewritten using the current dictionary, replacing each program variable with its logical counterpart. Then that expression is `internalized`, with the resulting logical variable used to extend the dictionary.

The execution engine for x86-64 assembly is moderately more complicated. Now the symbolic state associates not program variables but *registers and memory addresses* with logical variables. We take advantage of the stylized structure of cryptographic code to simplify the treatment of memory. The only valid pointer expressions are constant offsets from either function parameters (standing for cells within data structures passed to the function) or the stack pointer (standing for spilled temporaries). Thus, it is appropriate to make the symbolic memory a dictionary keyed off of *pairs of logical variables and integers*. The logical variable is the base address of an array in memory, while the integer gives a fixed offset into one of its words.

With this convention fixed, it is fairly straightforward to march through the instructions in a program, updating the register and memory dictionaries with the results of `internalize` calls. The symbolic executor effectively breaks each (possibly complex) x86 instruction down into multiple simpler operations on integers. There are multiple operations because many instructions affect both flags and their explicit destination registers. The explanations of those effects are expressions from the grammar above, using a relatively small vocabulary of bitvector operators, which include arithmetic and bitwise operators.

At the end of symbolic execution of an assembly function, we pull the output values out of calling-convention-designated registers and memory locations. These can then be compared against the explicit return value of a Fiat IR program. Both are expressed as logical variables connected to a common E-graph, so they should be syntactically equal exactly when the E-graph found a proof of equality. Importantly, both symbolic states are initialized with common logical variables standing for function inputs.

Consider a new but simple example of a function that, at a high level, is considered to take in five inputs. However, the first four inputs are actually the four machine words used to represent one big integer, and they are passed by-reference in a single array, so a C version of this function would have a signature like `int f(int *X, int y)`. Symbolic execution of an assembly variant of the function would start in this symbolic state, assuming the two function arguments are respectively passed in registers r_0 and r_1:

- **Registers:** $r_0 \mapsto X, r_1 \mapsto Y$
- **Memory:** $(X, 0) \mapsto X_0, (X, 1) \mapsto X_1, (X, 2) \mapsto X_2, (X, 3) \mapsto X_3$

Then, stepping through symbolic execution of some representative instructions:

(1) `mov` r_3, $[r_0]$ (load first element of array into r_3): adds a register mapping $r_3 \mapsto X_0$
(2) `add` $r_3, 7$ (adding seven to that value): overrides the mapping to $r_3 \mapsto X_0 + 7$
(3) `add` $[r_0 + 8], r_3$ (adding that value into the 3rd array cell): overrides the memory mapping $(X, 2) \mapsto X_2 + (X_0 + 7)$

It is important that expressions in the symbolic state are actually stored in internalized, normalized form thanks to the E-graph, so that semantic equality is checked by simple syntactic equality. We also benefit from ensuring that all symbolic memory regions being tracked do not overlap each other, allowing for simple lookup and overwriting of words in memory.

4.4 The E-graph

Following SMT-solver conventions [Detlefs et al. 2005], our E-graphs include nodes for equivalence classes of symbolic expressions, in addition to the edges representing subterm relationships. Each node is configured to present the most compact representation of its equivalence class. For instance, whenever a node becomes provably equal to a constant, it is labeled with that constant, without

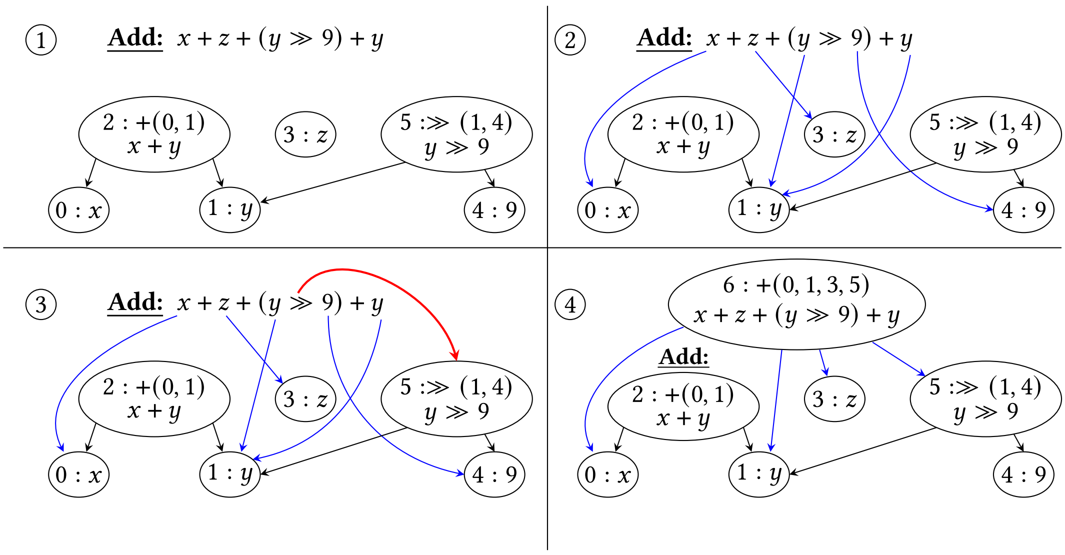

1
Add: $x + z + (y \gg 9) + y$
$2 : +(0, 1)$ $x + y$
$3 : z$
$5 : \gg (1, 4)$ $y \gg 9$
$0 : x$
$1 : y$
$4 : 9$
2
Add: $x + z + (y \gg 9) + y$
3
Add: $x + z + (y \gg 9) + y$
4
$6 : +(0, 1, 3, 5)$ $x + z + (y \gg 9) + y$
Add:

Fig. 7. Example of operations on an E-graph-style structure

outgoing edges. When a node is most succinctly expressed as a sum, it is labeled with a “+” operator and has edges to the other nodes being added. In general, a node has not a set of edges but rather an ordered list of edges.

Figure 7 animates a simple example. It steps through stages of adding a new node to the graph, representing a particular symbolic expression ($x + z + (y \gg 9) + y$). Step 1 shows the initial state. Nodes 2 and 5 are labeled with operators and IDs of operands. To process the expression we are evaluating, we first look up existing graph nodes for all of its leaf expressions, as Step 2 shows. Then we proceed bottom-up in the expression tree, finding an existing node or building a new one for each subexpression. In this case, we next need to find a node for $y \gg 9$, in Step 3. As we resolved y to node 1 and 9 to node 4, we are able to search the DAG for a node already labeled with operator $\gg$ and argument nodes 1 and 4, finding node 5. In the final step, Step 4, we need to find a node for $x + z + (y \gg 9) + y$. We have node IDs for all four operands of addition, and we *sort* them by ID, taking advantage of associativity and commutativity of addition, to find a canonical description of this node. No existing node has that description, so we add a new one. The main complication absent from this example is normalization with rewrite rules going beyond associativity and commutativity; the approach is parameterized on a set of such rules (which must have proofs in Coq), and they are applied as each node of the input AST is processed.

Next, we sketch how the `internalize` procedure (Algorithm 2) works more generally. Without loss of generality, we assume it is called on an expression that is an operator applied to a list of variables, which are used as E-graph node names. To internalize expressions with constants and/or deeper nesting of operands, we can simply traverse their trees bottom-up, internalizing each node to obtain a variable (i.e. E-graph node) to replace it with.

This code follows the maxim that provably equivalent expressions should be internalized into the same node, which requires that certain simplifications happen during internalization. The first two conditionals canonicalize uses of operators with identities and associativity, and the fourth handles commutativity. The third conditional is the one example we give of an operator-specific algebraic rewrite rule, which together form the heart of the approach. The example rule notices

Algorithm 2: Internalize expression into the E-graph

```
input  : Op an operator, Args its list of argument variables
output : n, a variable / graph node for the expression's equivalence class

function internalize(Op, Args)
begin
    if Op is associative then
        for argument a in Args do
            if a is also labeled with Op then
                Expand a in the list into its own E-graph neighbors
            end
        end
    end
    if Op has identity element e then
        Remove from Args any variable whose node is labeled with constant e.
    end
    if Op is LowByte and len(Args) = 1 and Args[0] is labeled with a constant below $2^8$ then
        return Args[0]
    end
    /* Many other algebraic rewrite rules                                */
    if Op is commutative then
        Sort Args by textual variable name.
    end
    for node n in the E-graph do
        if n is labeled with Op and has Args as its edge list then
            return n
        end
    end
    n ← new E-graph node;
    n.label ← Op;
    n.edges ← Args;
    return n
end
```

that an operation to extract the low byte of a constant is extraneous, when the constant is low enough. A comment stands in for the remaining twenty-some rules, elided for space reasons. After the main conditionals, the algorithm scans the E-graph for any existing node matching the (perhaps simplified) operator and arguments. Empirically, our programs are short enough that this linear scan finishes in reasonable time. If no compatible node is found, we create and return a new one.

The actual implementation includes a general range analysis to establish upper bounds on variable nodes based on bounds on their inputs. This feature is used to elide truncations whenever appropriate and to gate other rewrite rules: for example, the carry bit resulting from adding small numbers is always 0, and a sum of a couple of carry bits fits in a byte. In the case of code using arithmetic tricks to load and store individual flags, symbolic execution also relies on a number of other rewrite rules about flag computation in case one operand is a constant: adding a constant with all bits set to a Boolean variable produces a carry iff that variable is 1, adding the largest signed integer analogously loads the overflow flag, etc. Accesses to low or high parts of an x86 register are also modeled using bitwise operations and reconciled with the corresponding Fiat Cryptography code using rewrite rules.

4.5 Proof

We proved the Coq implementation of equivalence checking correct, in the sense that symbolically executing an assembly (or IR) program produces an expression (E-graph) that would evaluate to the same output as the original program from all starting states. A fairly direct corollary is our main theorem: if two programs are symbolically executed with the same (symbolic) inputs and produce outputs that are represented by the same E-graph nodes, then the programs are equivalent. We combine the Fiat IR symbolic evaluator with the larger Fiat Cryptography pipeline, and we verify it against the existing semantics of the IR, before extracting the pipeline to a command-line program. That program's functionality takes as input a choice of cryptographic algorithm, its numeric parameters, and an assembly file, then check that the assembly file matches the behavior of the algorithm, by comparing it with the Fiat IR code (itself generated by a fairly complex but verified compiler from prior work).

The proof of assembly-language symbolic execution is structured around a step-by-step simulation argument. Symbolic execution of one instruction is verified by tracing every E-graph variable it loads from the symbolic state through the `internalize` calls operating on it to the eventual update of the symbolic state. As not every x86-64 assembly instruction maps directly to an E-graph operation, sometimes reasoning similar to that used to justify rewrite rules is required to reestablish the simulation relation: for example, `shrd` is expanded to two shifts and an or, which are proven to compute the same value by showing that bits at all indices are the same on both sides. Some relatively tame use of ideas from separation logic [Reynolds 2002] helps us keep track of the relationship between byte-level and word-level accesses to memory.

It is important that the top-level theorem of the equivalence checker avoids mentioning any specifics of symbolic execution or E-graphs, as those are relatively complex techniques. Instead, that theorem refers only to formal semantics of Fiat IR and x86-64 assembly. The former came from Erbsen et al. [2019], and we formalized a new semantics of assembly (as a simple single-instruction interpreter, a terminating logical function) and proved that if the symbolic equivalence checker succeeds, then the semantics of the assembly program matches the semantics of a Fiat IR program it was compared against. Slightly more precisely, we depend on the following preconditions, often stated using separation logic.

- Calling-convention-designated registers hold input values, input-array base pointers, and output-array base pointers.
- Input-array base pointers point to arrays in memory holding input values.
- Output-array base pointers are valid.
- `rsp` points to the end of the stack, which must be valid.

Then the main theorem concludes the following postconditions.

- Input-array base pointers are still valid.
- Output-array base pointers point to arrays in memory holding the output values.
- The stack base-pointer address is still a valid pointer to an array of the right size.
- Callee-save registers have the same values as before the code execution.
- All other memory is untouched.

A few other concerns must be stated in the full theorem, including that all source-program initial variable values fit in machine words, arrays are layed out contiguously (the symbolic-execution engine allows more flexible specification of memory contexts), and every array has a start address stored directly in a register.

Table 1. Overview of Target Machines Used in the Experiments.

Name	CPU	Microarchitecture	Cores (T)	Motherboard
1900X	AMD Ryzen Theadripper 1900X	Zen 1	8 (8)	ASUS ROG STRIX X399-E Gaming
5800X	AMD Ryzen 7 5800X	Zen 3	8 (16)	Gigabyte B550 AORUS ELITE V2
5950X	AMD Ryzen 9 5950X	Zen 3	16 (32)	Gigabyte X570 GAMING X
i7 6G	Intel Core i7-6770HQ	Skylake-H	4 (4)	Intel NUC6i7KYB
i7 10G	Intel Core i7-10710U	Comet Lake-U	6 (6)	Intel NUC10i7FNB
i9 10G	Intel Core i9-10900K	Comet Lake-S	10 (20)	Gigabyte H470 HD3
i7 11G	Intel Core i7-11700KF	Rocket Lake-S	8 (8)	ASRock Z590 Pro4
i9 12G	Intel Core i9-12900KF	Alder Lake-S	16 (24)	Micro-Star PRO Z690-A Wifi (MS-7D25)

5 PERFORMANCE EVALUATION

In this section, we evaluate the performance of code produced by CryptOpt. Specifically, we answer four main questions:

(1) How does optimization progress over time (Section 5.2)?
(2) How does CryptOpt compare with traditional compilation (Section 5.3)?
(3) Is CryptOpt optimization platform-specific (Section 5.4)?
(4) How does CryptOpt-optimized code performs as part of a cryptographic implementation (Section 5.5)?

We first describe the setup and the procedures we use for the evaluation. We then describe the experiments we carry out and their results.

5.1 Experimental Setup

In this subsection, we describe the hardware platforms and discuss the *generation* of results through optimization, plus the *evaluation* of which one is the best of all the results.

Hardware Platforms. To compare across multiple processor architectures, we evaluate CryptOpt on multiple hardware platforms, summarized in Table 1. We did not observe differences in optimization behavior on machines with SMT enabled or disabled. We use Ubuntu 21.04 for all machines accept for i9 12G. i9 12G runs Ubuntu 22.04 LTS, because the kernel in Ubuntu 21.04 did not support the 12th-generation Intel processors. Moreover, because performance counters are not available on the efficiency cores of the i9 12G, we only use the performance cores on the machine.

Generation. All platforms have at least four physical cores. For a fair comparison, we run the same number of optimizations processes in parallel on all platforms. We pin the optimization processes to cores to reduce noise due to context switching. We chose to run three optimizations in parallel, keeping one core free for general OS activity. This results in three assembly files for each primitive per platform. We report the performance of best performing out of the three.

Bet-and-Run. Each optimization process has a budget of 100 000 mutations. In the bet stage, we explore 20 initial candidate solutions, optimizing each for 500 mutations. Hence, overall, we use 10 000 mutations, which are 10% of the total budget for the bet part. The remaining 90 000 mutations are used for the run stage of the Bet-and-Run strategy. With those parameters, the optimization takes between 42 wall-clock hours on i9 12G and 84 hours on the less powerful NUC's i7 6G and i7 10G.

Code Verification. When combined with Fiat Cryptography, the optimization process regularly verifies that the code it produces is equivalent to the Fiat IR code. By default, we perform verification 20 times during the optimization, at intervals of 5% of the mutations budget. Overall, verification takes 10% of the total optimization time on average (median 3%, maximum 75%). In practice, verifying during the optimization may be excessive, as verification is only necessary for the final output. This would reduce the overhead to the order of 1%.

Performance Metric. To compare the performance of different implementations we need a stable metric that measures the performance. To reduce system noise we fix the CPU frequency, disable boosting, and set the governor to `performance`. We note that we only apply these settings when evaluating the performance but not during optimization. We observe that the performance loss due to controlling CPU speed outweighs the benefits of the stability it provides.

Even with a stable CPU speed, performance measurements may vary. To ensure a stable metric, we follow a careful procedure that consists of measuring multiple repetitions of the implementation and using statistical tools to ensure stable results. Specifically, to remove variations due to microarchitectural state, we measure the execution time of 400 repetitions of the code-under-test. To eliminate noise from interrupt handling and other OS functionality, we take 31 such measurements, and use the median as a representative execution time for the code. Appendix A presents the procedure in more details. The parameters here correspond to the use of $nob = 31$ and $bs = 400$ in the procedure there.

The result that we get from the measurement procedure outlined above may still depend on the memory location in which the code is instantiated [Mytkowicz et al. 2009]. To reduce this dependency, we repeat the whole procedure 20 times, obtaining a population of 20 different medians. We randomly split this population into two sets of 10 medians and use a Student t-test to compare the distributions. If the t-test indicates that the distributions sampled by the two sets are indistinguishable ($p < 0.05$), we use the average of the population as the performance metric for the code. In the rare case that the t-test fails, we restart the evaluation.

5.2 Optimization Progress

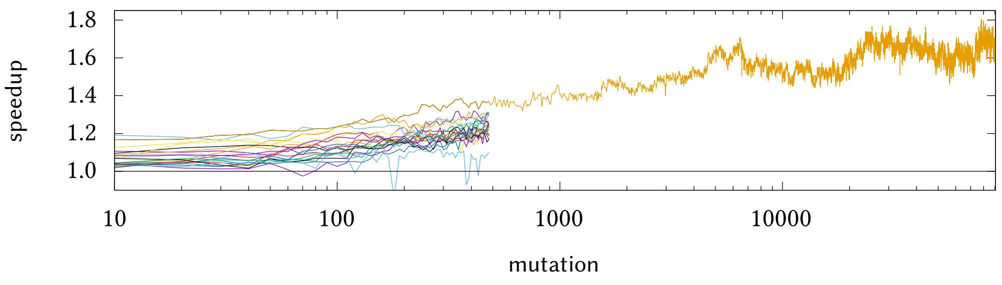
The first question we answer is how the optimization progresses over time. Figure 8 shows examples of optimization runs on 5950X and i9 10G, targeting the field multiplication operation corresponding to the prime field used for the NIST P-256 elliptic curve. The figures show performance over time, where performance is measured as gain over the baseline Clang-compiled code, while optimization time is measured in mutations (shown in log scale).

The figure shows the Bet-and-Run strategy in action. The optimization starts with 20 runs, each with a budget of 500 mutations. CryptOpt then selects the best-performing and continues optimizing it for a further 90 000 mutations. As the figure shows, in all runs, optimization progress is roughly logarithmic in the number of mutations.

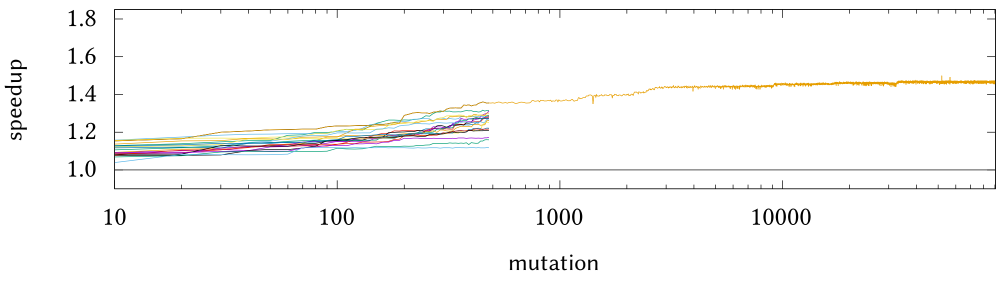
A notable difference between the two optimizations is the level of noise evident during optimization. Time measurements on the i9 10G platform for NIST P-256-multiplication are mostly stable over time, showing mostly monotonic improvement. Measurements on the 5950X for NIST P-256-multiplication are much less stable, showing variations of up to 15% between consecutive measurements. Finally, we note that despite these variations, performance tends to improve over time, achieving an overall performance gain of 47–69%.

5.3 CryptOpt vs. Off-the-Shelf Compilers

To compare CryptOpt with traditional compilers, we use the implementations of finite-field arithmetic as produced by Fiat Cryptography. Specifically, we use Fiat Cryptography to produce implementations of the multiply and the square functions in nine fields. We consider prime fields of the standardized NIST P curves: Curve P-224, Curve P-256, Curve P-384 and Curve P-521 [NIST



(a) NIST P-256-multiplication on 5950X



(b) NIST P-256-multiplication on i9 10G

Fig. 8. CryptOpt optimization progress optimizing NIST P-256-multiplication on two platforms. Each line traces an optimization run over time (X-axis, log-scale), showing the relative performance gain over Clang. Bet-and-Run used for 20 runs with a budget of 500 mutations each. The best-performing candidate continues for another 90 000 mutations. Progress over time is roughly logarithmic in the number of mutations, achieving a performance gain of 47–69%. We generate one data point every ten mutations.

2000]. Moreover, we consider the field of the popular 'Bitcoin' curve secp256k1 [Certicom Research 2000], the high-speed de-facto standard Curve25519 [Bernstein 2006], and a high-security Curve448 [Hamburg 2015]. In addition to elliptic-curve cryptography, we apply our method to the underlying fields of the post-quantum scheme SIKEp434 [Azarderakhsh et al. 2019] and of the Poly1305 message-authentication scheme [Bernstein 2005]. It is worth noting that Erbsen et al. [2019] reported that their generated C code was roughly the best-performing available for all elliptic curves, up to the usual vagaries of C-compiler optimizers fluctuating in behavior across versions, so it makes sense to use that C code as our performance baseline.

We run CryptOpt on each of the eight platforms summarized in Table 1, and select the best result as described in Section 5.1. Additionally, we compile the equivalent C code, as produced by Fiat Cryptography, with GCC 11 [GCC 2022] and Clang 13 [Clang 2022]. We use the highest optimization level the compilers support and enable native support using the compilation switches `-march=native -mtune=native -O3`.

Table 2 shows a summary of the results. For each function the table presents the geometric mean performance gain of CryptOpt over GCC and Clang. The mean is calculated over the different platforms; see Table 4 in Appendix B for full details. The table shows that CryptOpt achieves significant performance gains in the majority of functions. The performance gains are somewhat more modest when the produced code does not require any memory spills, as is the case for operations in the fields of Curve25519 and Poly1305. For large fields, as in P-521 and Curve448,

Table 2. Geometric means of CryptOpt vs. off-the-shelf compilers.

	Multiply		Square	
Curve	GCC	Clang	GCC	Clang
Curve25519	1.16	1.25	1.20	1.16
P-224	2.55	1.51	2.47	1.34
P-256	2.65	1.63	2.53	1.52
P-384	2.07	1.21	2.15	1.17
SIKEp434	2.10	1.46	2.13	1.44
Curve448	0.82	0.96	0.89	0.88
P-521	1.00	1.24	1.11	1.42
Poly1305	1.17	1.11	1.18	1.03
secp256k1	2.30	1.61	2.24	1.51

CryptOpt is less successful, achieving modest gains (or on par) for the former and underperforming for the latter. We note that the code produced for these curves is quite large. As an example, the resulting x86-64 assembly files for Curve448-mul are in the range of 525–700 instructions and for P-521-mul 463–630 instructions depending on the platform. For comparison, Curve25519-mul is in the order of 160–180 instructions, and in the range of 70–80 for Poly1305-mul. We suspect that CryptOpt's simple mutations have less impact on the execution time for those big functions than what would be needed to be measurable and direct the optimizer towards the optimal instruction sequences. We leave more sophisticated genetic-improvement strategies for future work.

5.4 Platform-Specific Optimization

CryptOpt optimizes the execution time of the function on the platform it executes on. Because different platforms have different hardware components, the fastest code on one platform is not necessarily the fastest on another. Appendix B goes into detail on our analysis of when best results come from running the optimization search on the target platform or on a different platform. For some curves, like Curve25519, optimizing and running on the same architecture tends to outperform cross-architecture optimizations, which aligns with common intuitions. However, there are counterexamples. In particular, it appears that optimization of P-384 functions on AMD is not ideal, and the results underperform code optimized on Intel processors. We note that measurement on AMD machines is less stable than on Intel machines (see 5950X and i9 10G in Figure 8). We therefore hypothesize that although optimization is possible on the AMD machines, in some cases the local search may drift in the configuration space, missing optimization opportunities and producing less performant code (still, for the majority faster than compiling with off-the-shelf compilers, see Appendix B).

5.5 Scalar Multiplication

So far we have focused on the optimized functions in isolation. In this section we investigate the use of the field operations within the context of elliptic curves cryptography. Specifically, we investigate implementations of three popular elliptic curves, Curve25519, NIST P-256, and secp256k1. We compare the performance of 19 implementations of these curves, five of which use CryptOpt code for field operations. In Table 3 we summarize the implementations we investigate.

State of the art. For Curve25519, the SUPERCOP benchmark framework [Bernstein and Lange 2022] provides us with many implementations: sandy2x [Chou 2015]), amd64-51 and amd64-64 [Chen et al. 2014], as well as donna and donna-c64 Langley [2022]). OpenSSL [OpenSSL 2022]

Table 3. Performance of Scalar Multiplication (Geometric Mean).

Curve25519		
Implementation	Lang	Cycles
sandy2x [Chou 2015]	asm-v	496k
amd64-64 [Chen et al. 2014]	asm	564k
amd64-51 [Chen et al. 2014]	asm	564k
donna [Langley 2022]	asm-v	985k
donna-c64 [Langley 2022]	C	723k
O’SSL [OpenSSL 2022]	C	545k
O’SSL fe-51 [OpenSSL 2022]	asm	554k
O’SSL fe-64 [OpenSSL 2022]	asm	465k
O’SSL fe-51+**CryptOpt**	asm	542k
Boring [BoringSSL 2022]	C	682k
Boring+**CryptOpt**	asm	633k

NIST P-256		
Implementation	Lang	Cycles
z-256 [Gueron and Krasnov 2013]	asm-v	608k
Boring C-Fallback	C	1231k
Boring+**CryptOpt**	asm	1053k

secp256k1		
Implementation	Lang	Cycles
O’SSL	C	5242k
O’SSL+**CryptOpt**	asm	2258k
libsecp256k1 [Bitcoin Core 2022]	asm	595k
libsecp256k1 [Bitcoin Core 2022]	C	566k
libsecp256k1+**CryptOpt**	asm	580k

provides many implementations (two assembly-based and one portable C version) and chooses at run time which one to use, opting by default for OSSL fe-64, a variant of amd64-64. BoringSSL [BoringSSL 2022] only has a C version which uses Fiat Cryptography’s C code.

For NIST P-256, we use z256 [Gueron and Krasnov 2013], which is the default implementation in both BoringSSL and OpenSSL. The implementation relies on highly-optimized vectorized assembly code for the entire point operations. Additionally, we use the C fallback code of BoringSSL, which relies on the Fiat Cryptography C code for field operations.

For secp256k1 we use the generic implementation from OpenSSL and two implementations from the libsecp256k1 library [Bitcoin Core 2022].

CryptOpt-based Implementations. We replace the field operations in five of the implementations with CryptOpt-optimized assembly code. In implementations, which provide an API similar to Fiat Cryptography, we can just replace the field operations. This is the case for OpenSSL’s and BoringSSL’s implementations of Curve25519 and the C-based fallback implementation of NIST P-256 from BoringSSL. For secp256k1, we specialize the generic implementation of OpenSSL with Fiat+CryptOpt code. Additionally, in Case Study 2, we generate a `libsecp256k1`-compatible implementation and can replace the existing ones.

Evaluation. We use the SUPERCOP benchmark framework [Bernstein and Lange 2022] to measure the performance of the evaluated implementations. For each implementation, SUPERCOP tries multiple combinations of compilers and compiler options and reports the execution time of the fastest implementation. We evaluate each implementation on the 8 machines (c.f. Table 1) and report the geometric mean (rounded to the nearest 1000 cycles) in Table 3. (See Table 5 in Appendix C for the full details.)

Results. In both BoringSSL implementations, we see that CryptOpt outperforms the Fiat Cryptography C code. Consistent with the results we report in Section 5.3. Because the implementations share significant parts of the code, the improvement with CryptOpt is lower than when only comparing field operations. CryptOpt outperforms generic C code, e.g. as used by OpenSSL for secp256k1. This is expected and consistent with the findings of Erbsen et al. [2019], because the code that CryptOpt optimizes is specialized to the specific field.

Comparing CryptOpt with the similarly structured hand-optimized implementations of OpenSSL fe-51 and libsecp256k1, we find that the performance with and without CryptOpt is similar. Manual optimization of code requires significant expertise and a large time investment, that needs to be repeated for each finite field. In contrast, using CryptOpt is fairly straightforward and it only requires moderate computing resources to achieve similar results.

CryptOpt underperforms highly-optimized implementations that use different data representations (OSSL fe-64) or assembly features (z256) we do not support. We leave the task of supporting these features to future work.

CryptOpt on Intel i9 12G. When looking at the results on specific machines, we see that CryptOpt excels on the i9 12G platform. On this platform, CryptOpt-based implementations of Curve25519 and secp256k1 are the fastest, outperforming hand-optimized implementations, including those that use advanced processor features, such as vector instructions. The 12th generation of Intel processors is a major update of the microarchitecture. We believe that CryptOpt's automated search allows it to exploit the benefits of the new design automatically. In contrast, prior implementations and mainstream compilers need to change to adapt to these new features. We anticipate that in due course, implementations will be adapted to the new design and hand-tuned implementations will outperform CryptOpt. However, CryptOpt does not require manual effort to adapt to new designs.

6 RELATED WORK

CryptOpt applies Genetic improvement (GI), an area within Search-Based Software Engineering [Harman and Jones 2001] that uses automated search to find improved versions of existing software. To carry out this (typically combinatorial) search, GI applies program synthesis and transformation operators with the aim of improving the functional or non-functional properties of a target program. The key promise of GI is to automate the production of routine improvements to programs and free software developers to tackle more challenging tasks. Genetic improvement is a relatively young research field; its first survey appeared in 2018 [Petke et al. 2018]. Despite its youth, GI has already had real-world impact: maintainers have accepted GI patches into both open-source [Langdon et al. 2015] and commercial [Haraldsson et al. 2017] projects.

An effective GI practice is to use edit operators that use raw material from the program [Petke et al. 2019]. These include: copy, delete, swap, and replacement. The search process then aims to evolve patches consisting of sequences of one or more edits. Other practices in GI involve, among others, the selection of alternative implementations [Burles et al. 2015], the tuning of deeply embedded constants [Wu et al. 2015], and the transplantation of functionality [Barr et al. 2015].

CryptOpt utilizes GI in the code-generation phase to generate fastest code per-microarchitecture. To the best of our knowledge, CryptOpt is the first automatic compiler to offer both this level of microarchitecture-tuned performance (albeit for the limited domain of straightline crypto code) and the highest level of formal assurance (Coq verification of all compiler phases that matter for soundness). However, related work has tackled some of the constituent challenges.

Bosamiya et al. [2020] is most similar to CryptOpt as they also use GI to find fast code per architecture while being provably correct. The primary objective is to parse optimized x86-64 assembly and then use verified transformations to transform it back into a clean form, which is then easier to reason about. From those transformations, they selected the "prefetch insertion" and "instruction-reordering" transformations and conducted a case study on using GI to find fast implementations based on OpenSSL's AES-GCM (which uses AES-NI instructions). They showed that they can generate verified correct per-architecture optimized code. In contrast to CryptOpt, they rely on an initial, cleverly optimized assembly implementation, which they then use as a base to alter the order of instructions, whereas CryptOpt generates the code itself and can thus take advantage of microarchitectural resources, with optimization of register allocation (in particular spills to memory) and instruction selection. For example, using their reordering transformation, they cannot use an add-using-overflow `adox` over an add-with-carry `adcx`, let alone having those calculate two independent additions in parallel, because they only consider reads and writes to the flag register in general, rather than the granularity of individual flags (i.e. read CF, write OF are independent). The benefit of CryptOpt is also the compatibility with Fiat Cryptography, which makes code generation for finite fields for new primes easy.

Optimization pass finding. Typical compilers have a myriad of different optimization passes [Aho et al. 1986]. They range from high-level (algorithmic) optimizations—such as loop-invariant code motion, constant folding, dead-code elimination, common-subexpression elimination and function inlining—to more low-level (architecture-dependent) optimizations—such as loop unrolling (for example based on cache sizes), code alignment, vectorization, etc. The former are typically guaranteed to improve code performance [Seo et al. 2021] because they remove big chunks of redundant and superfluous instructions. To ease use for the developer, they employ different optimization levels (like "-O3") in which a selection of those passes are applied. It is well known that those are targeting the average program and are not necessarily optimal for specific use cases [Mytkowicz et al. 2009; Naono et al. 2010][Supalov et al. 2014, p.218]. Thus, there is work on tuning compiler passes in selecting different passes and ordering them. Stephenson et al. [2003] and Peeler et al. [2022] both use GI to select and order existing optimization passes for optimal running time, where the former uses a simulated running time (Trimaran [Chakrapani et al. 2005]) as the objective function and the latter uses the actual running time. CryptOpt, however, is not bound by either applying or not applying those fixed optimization passes from off-the-shelf compilers. Rather, it explores many different variations for any particular code section and is also able to selectively apply optimizations at certain locations and avoid using them at others. Batina et al. [2014] use genetic algorithm to find the optimal positioning of flip-flops in for use in hardware implementations of AES.

Peephole optimization. Peephole optimizers use a sliding window on instructions (the peephole) and replace sets of instructions with more performant instructions [Aho et al. 1986; Bergmann 2003; Cooper and Torczon 2012]. The replacement is done based on a predefined rule set, which itself is based on heuristics for estimating which set of instructions is more performant. Hundt et al. [2011] present MAO, an assembly-to-assembly optimizer, which optimizes based on similar predefined rules. Finding and learning good peephole optimizations automatically is a different approach. Bansal and Aiken [2006] use machine-learning techniques to characterize small sections of code. Then, based on those characteristics, they replace a code sequence with a semantically equivalent ones assumed to be more performant. While they only focus on small sections (on the order of tens of instructions), Pekhimenko and Brown [2010] use machine-learning techniques to characterize entire methods and then apply certain optimization transformations to them. Similarly, CryptOpt considers the entire function as a whole, but rather than characterizing, learning and applying that knowledge to new functions, CryptOpt considers each architecture and function as a black box and eventually finds a fast implementation.

Verified compilers. Certified compilers, such as CompCert [Leroy et al. 2016] and derivatives [Almeida et al. 2020; Barthe et al. 2020], are proven to preserve the semantics of the code. CompCert consists of multiple optimization passes, which seem generic instead of specializing to microarchitectural features. Moreover, CompCert focuses on control flow rather than aggressively optimizing straight-line code. Similarly, Necula [2000] validates correctness of some GCC optimization passes of GCC, ensuring that they preserve program semantics.

Verified transformations. Instead of proving the correctness of the compiler, translation validation [Pnueli et al. 1998] does not trust the compiler, but verifies that the compiled code preserves the semantics of the source. Bosamiya et al. [2020], as already mentioned, used their tool to transform optimized (manually written) assembly code to easily verifiable code. Similarly, TInA [Recoules et al. 2019] lifts inline assembly to semantically equivalent C code amenable to verification with known tools. Only targeting the code for Curve25519, Schoolderman et al. [2021] used the Why3 proving platform [Filliâtre and Paskevich 2013] to verify an 8-bit AVR implementation, finding a fixing a bug in the process. Chen et al. [2014] formally verified the field arithmetic used in the ladder step in both radix-2^{51} and radix-2^{64} using the BOOLECTOR SMT Solver. Last, Sewell et al. [2013] go one step further and parse the binary (as opposed to assembly) code of the seL4-Linux microkernel and transform it until they could prove equivalence to the already-verified C code.

Similar to those, with CryptOpt, we extend Fiat Cryptography with x86 semantics and then use verified transformations to prove equivalence of the emitted assembly to the Fiat IR.

We would like to emphasize that those works aim to verify *existing* implementations, whereas CryptOpt *generates* them. Targeting a wide range of microarchitectures for performance optimizations manually would quickly become practically infeasible.

Real-world applications of computer-aided cryptography. We see provably correct generated code being used in the real world: *Project Everest* aims to combine many tools and languages to generate an entire verified HTTPS software stack. They primarily use F* [Swamy et al. 2016], a dependently typed language in the ML family, allowing correctness to be encoded in types. F* can then be compiled to F# or OCaml. Low* [Protzenko et al. 2017], a subset of F*, can be compiled to C in a provably correct way. Vale [Bond et al. 2017; Fromherz et al. 2019] is a collection of verified high-performance assembly code for several primitives, also using F*/Low* to ensure correctness.

Project Everest includes HACL* [Zinzindohoué et al. 2017], which produces verified high-performance C code using F*, Low* and assembly using Vale.

Finally, *EverCrypt* [Protzenko et al. 2020] combines HACL* and Vale to generate a carefully designed API, which supports many different algorithms (for the same functionality, e.g. AES+GCM or ChaCha20+Poly1305 for Authenticated Encryption) and for each of those many different implementations optimized depending on platform and hardware. EverCrypt thus aims more at narrowing the gap from those research projects to industry usage.

On the usage of verification of existing code, we see that protocols in TLS 1.3 have been verified using ProVerif and CryptoVerif in Bhargavan et al. [2017]. Mozilla uses *HACL** and *HACLxN* [Polubelova et al. 2020; Zinzindohoué et al. 2017], and Google uses code generated by Fiat Cryptography [Erbsen et al. 2019] in their cryptographic library BoringSSL. A main limitation of adapting provable secure code is that the implementation must be at least as fast as the current implementation, and the API must be compatible. When it comes to ECC primitives, Belyavsky et al. [2020] published work on generating prime-agnostic point-arithmetic in C using verified field arithmetic from Fiat Cryptography and claim timing side-channel-resistant code; however, there is no formal verification of correctness or constant time.

7 CONCLUSION

In this work, we present CryptOpt, a code-generation framework that aims at generating high-performance cryptographic code. Unlike mainstream compilers, CryptOpt uses combinatorial optimization for producing code. That is, CryptOpt represents code generation as a search problem in a solution space, where the aim is to find the best solution.

CryptOpt operates on straight-line code that does not contain any control-flow instructions. This simplifies code generation because the constraints on instruction ordering are fully captured in the acyclic data-flow graph. At the same time, significant sections of typical cryptographic code are straight-line. Hence, CryptOpt is suitable for the purpose it is designed for.

The code produced by CryptOpt is highly efficient, achieving a performance improvement by a factor of 1.68–2.54 over the generic OpenSSL field operations. Moreover, the code shows similar performance to non-vectorized hand-tuned assembly implementations.

A further advantage of CryptOpt is that it is designed to work as a back-end for Fiat Cryptography. When used with Fiat Cryptography, CryptOpt extends the correctness proof to the assembly level, using a new formally verified program-equivalence checker.

ACKNOWLEDGMENTS

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the ARC Discovery Early Career Researcher Award DE200101577; the ARC Discovery Projects DP200102364 and DP210102670; the Blavatnik ICRC at Tel-Aviv University; the National Science Foundation under grants CNS-1954712 and CNS-2130671; the National Science Foundation Expedition on the Science of Deep Specification (award CCF-1521584); the Phoenix HPC service at the University of Adelaide; and gifts from Amazon Web Services, AMD, Facebook, Google, Intel and the Tezos Foundation.

REFERENCES

0xAde1a1de. 2022. AssemblyLine. https://github.com/0xAde1a1de/AssemblyLine 31

Andreas Abel and Jan Reineke. 2021. Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. arXiv 2107.14210. 12

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. 2, 24

Alaa R Alameldeen and David A Wood. 2003. Variability in architectural simulations of multi-threaded workloads. In *HPCA*. 7–18. 30

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. 2020. Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification. In *INDOCRYPT*. 107–127. 24

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *PACT*. 303–316. 12

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*. 135–150. 12

Anne Auger and Benjamin Doerr (Eds.). 2011. *Theory of Randomized Search Heuristics: Foundations and Recent Developments.* Series on Theoretical Computer Science, Vol. 1. World Scientific. 7

Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. 2019. Supersingular Isogeny Key Encapsulation – Submission to the NIST Post-Quantum Standardization Project, round 2. https://sike.org 4, 20

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *ASPLOS*. 394–403. 24

Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *ISSTA*. 257–269. 23

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. In *POPL*. 7:1–7:30. 24

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *CSF*. 328–343. 2

Lejla Batina, Domagoj Jakobovic, Nele Mentens, Stjepan Picek, Antonio de la Piedra, and Dominik Sisejkovic. 2014. S-box Pipelining Using Genetic Algorithms for High-Throughput AES Implementations: How Fast Can We Go?. In *INDOCRYPT*. 322–337. 24

Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. 2020. Set It and Forget It! Turnkey ECC for Instant Integration. In *ACSAC*. 760–771. 25

Seth D. Bergmann. 2003. Compilers. In *Encyclopedia of Information Systems*. 141–170. 24

Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *FSE*. 32–49. 4, 20

Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC*. 207–228. 20

Daniel J. Bernstein, Tung Chou, and Peter Schwabe. 2013. McBits: Fast Constant-Time Code-Based Cryptography. In *CHES*, Vol. 8086. 250–272. 2

Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. 2014a. Curve41417: Karatsuba Revisited. In *CHES*. 316–334. 2

Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. 2014b. Kummer Strikes Back: New DH Speed Records. In *ASIACRYPT*. 317–337. 2

Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. 2017. NTRU Prime: Reducing Attack Surface at Low Cost. In *SAC*. 235–260. 2

Daniel J. Bernstein and Tanja Lange. 2022. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to/supercop/supercop-20220213.tar.xz 21, 22

Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *IEEE SP*. 483–502. 25

Bitcoin Core. 2022. libsecp256k1 - Optimized C Library for ECDSA Signatures and Secret/Public Key Operations on Curve secp256k1. https://github.com/bitcoin-core/secp256k1 3, 22, 35

Mahmoud A. Bokhari, Brad Alexander, and Markus Wagner. 2020. Towards rigorous validation of energy optimisation experiments. In *GECCO*. 1232–1240. 30

Mahmoud A Bokhari, Lujun Weng, Markus Wagner, and Bradley Alexander. 2019. Mind the gap - a distributed framework for enabling energy optimisation on modern smart-phones in the presence of noise, drift, and statistical insignificance. In *IEEE CEC*. 1330–1337. 30

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium*. 917–934. 25

BoringSSL. 2022. BoringSSL. https://boringssl.googlesource.com/boringssl 22, 35

Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. 2020. Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language. In *VSTTE*. 106–123. 3, 23, 24

Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In *SSBSE*. 255–261. 23

Christopher Celio, Palmer Dabbelt, David A. Patterson, and Krste Asanovic. 2016. The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V. arXiv 1607.02318. 4

Certicom Research. 2000. SEC 2: Recommended elliptic curve domain parameters, version 1.0. http://www.secg.org/SEC2-Ver-1.0.pdf. 20

Lakshmi N. Chakrapani, John Gyllenhaal, Wen-mei W. Hwu, Scott A. Mahlke, Krishna V. Palem, and Rodric M. Rabbah. 2005. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. In *Languages and Compilers for High Performance Computing*. 32–41. 24

Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *CCS*. 299–309. 21, 22, 25, 35

Tung Chou. 2015. Sandy2x: New Curve25519 Speed Records. In *SAC*. 145–160. 2, 21, 22, 35

Tung Chou. 2016. QcBits: Constant-Time Small-Key Code-Based Cryptography. In *CHES*, Vol. 9813. 280–300. 2

Chitchanok Chuengsatiansup, Michael Naehrig, Pance Ribarski, and Peter Schwabe. 2013. PandA: Pairings and Arithmetic. In *Pairing*, Vol. 8365. 229–250. 2

Chitchanok Chuengsatiansup and Damien Stehlé. 2019. Towards Practical GGM-Based PRF from (Module-) Learning-with-Rounding. In *SAC*. 693–713. 2

Clang. 2022. Clang: a C language family frontend for LLVM. https://clang.llvm.org 20

Keith D. Cooper and Linda Torczon. 2012. Chapter 11 - Instruction Selection. In *Engineering a Compiler (Second Edition)*. 597–638. 24

Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: statistically sound performance evaluation. In *ASPLOS*. 219–228. 30

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340. 12

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (2005), 365–473. 13, 14

Benjamin Doerr and Frank Neumann. 2019. *Theory of evolutionary computation: Recent developments in discrete optimization.* Springer. 7

Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *IEEE SP Workshops*. 73–87. 2

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE SP*. 1202–1219. 2, 3, 5, 17, 20, 22, 25

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP*. 125–128. 25

Matteo Fischetti and Michele Monaci. 2014. Exploiting Erraticism in Search. *Operations Research* 62, 1 (2014), 114–122. 8

Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. In *POPL*. 63:1–63:30. 25

GCC. 2022. GCC, the GNU Compiler Collection. https://gcc.gnu.org 20

Shay Gueron and Vlad Krasnov. 2013. Fast Prime Field Elliptic Curve Cryptography with 256 Bit Primes. IACR ePrint 2013/816. 22, 35

Mike Hamburg. 2015. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive* 2015 (2015), 625. 20

Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *GECCO (Companion)*. 1513–1520. 23

Mark Harman and Bryan F. Jones. 2001. Software engineering using metaheuristic innovative algorithms: workshop report. *Inf. Softw. Technol.* 43, 14 (2001), 905–907. 23

Rodney E. Hooker and Collin Eddy. 2013. Store-to-load forwarding based on load/store address computation source information comparisons. US Patent 8533438. 4

Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. 2011. MAO - An extensible micro-architectural optimizer. In *CGO*. 1–10. 24

Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005a. Automated detection of performance regressions: The Mono experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 183–190. 30

Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005b. Benchmark precision and random initial state. In *2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS)*. 853–862. 30

Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. *ACM SIGPLAN Notices* 48, 11 (2013), 63–74. 30

Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. 2019. Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates. In *ACNS*. 281–301. 2

Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS*. 573–582. 2

William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO*. 1063–1070. 23

Adam Langley. 2022. Curve25519-donna. https://github.com/agl/curve25519-donna 21, 22, 35

Kevin M. Lepak and Mikko H. Lipasti. 2000. On the value locality of store instructions. In *ISCA*. 182–191. 4

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. 12

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert — A Formally Verified Optimizing Compiler. In *ERTS*. 12, 24

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. 65–79. 12

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 265–276. 19, 24, 30, 33

Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda (Eds.). 2010. *Software Automatic Tuning, From Concepts to State-of-the-Art Results*. Springer. 24

George C. Necula. 2000. Translation validation for an optimizing compiler. In *PLDI '00*. 24

NIST. 2000. FIPS PUB 186-2: Digital signature standard. 19

OpenSSL. 2022. OpenSSL. https://www.openssl.org/ 4, 21, 22, 35

Hannah Peeler, Shuyue Stella Li, Andrew N. Sloss, Kenneth N. Reid, Yuan Yuan, and Wolfgang Banzhaf. 2022. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework. arXiv 2201.13305. 24

Gennady Pekhimenko and Angela Demke Brown. 2010. Efficient Program Compilation Through Machine Learning Techniques. In *Software Automatic Tuning, From Concepts to State-of-the-Art Results*. Springer, 335–351. 24

Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E. I. Brownlee, Markus Wagner, and David Robert White. 2019. A survey of genetic improvement search spaces. In *GECCO (Companion)*. 1715–1721. 23

Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 415–432. 23

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS*. 151–166. 24

Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella Béguelin. 2020. HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms). In *CCS*. 899–918. 25

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *IEEE SP*. 983–1002. 25

Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages* 1 (2017), 1 – 29. 25

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. 2012. Thread tranquilizer: Dynamically reducing performance variation. *ACM TACO* 8, 4 (2012), 46. 30

Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get Rid of Inline Assembly through Verification-Oriented Lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 577–589. https://doi.org/10.1109/ASE.2019.00060 24

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 17

Ronny Ronen, Alexander Peleg, and Nathaniel Hoffman. 2004. System and method for fusing instructions. US Patent 6675376B2. 4

Marc Schoolderman, Jonathan Moerman, Sjaak Smetsers, and Marko C. J. D. van Eekelen. 2021. Efficient Verification of Optimized Code - Correct High-Speed X25519. In *NFM*. 304–321. 25

Hwajeong Seo, Pakize Sanal, Wai-Kong Lee, and Reza Azarderakhsh. 2021. No Silver Bullet: Optimized Montgomery Multiplication on Various 64-bit ARM Platforms. *IACR Cryptology ePrint Archive* 2021 (2021), 185. 24

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*. 471–482. 25

Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman P. Amarasinghe. 2003. Genetic Programming Applied to Compiler Heuristic Optimization. In *EuroGP*. 238–253. 24

Samantika Subramaniam and Gabriel H. Loh. 2006. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *MICRO*. 273–284. 4

Alexander Supalov, Andrey Semin, Michael Klemm, and Christopher Dahnken. 2014. *Addressing Application Bottlenecks: Microarchitecture*. 201–246. 24

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. 256–270. 25

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*. 17–27. 12

Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: learning replacement policies from hardware caches. In *PLDI*. 519–532. 4

Thomas Weise, Zijun Wu, and Markus Wagner. 2019. An Improved Generic Bet-and-Run Strategy with Performance Prediction for Stochastic Local Search. In *AAAI*. 2395–2402. 8

Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation. In *GECCO*. 1375–1382. 23

Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS*. 1789–1806. 25

A ON RELIABLE PERFORMANCE MEASUREMENT

A.1 Background

The execution time of a program can be affected by a myriad of factors, such as, the state of cache memory [Alameldeen and Wood 2003]; context switches [Pusukuri et al. 2012]; memory layout and OS environment variables [Curtsinger and Berger 2013; Mytkowicz et al. 2009]. Even when a platform is rebooted between runs there can be significant variation in benchmark run times [Kalibera et al. 2005a]. Furthermore, measurements of the benchmark operation can be subject to inherent and external random changes to the benchmark operation, as well as the system state prior to execution [Kalibera et al. 2005b; Kalibera and Jones 2013]. In addition, sensor readings themselves can drift over time [Bokhari et al. 2019]. Consequently, all these factors together result in measurement noise, which can affect both the sensitivity and specificity of performance measurements. One recent work, which surveyed and compared validation approaches in the context of energy consumption optimization [Bokhari et al. 2020], found that a number of measurement approaches failed to mitigate the effects of measurement noise. Only the proposed approach R3-validation was not misled: it exercised the tests in a rotated-round-robin fashion, while accommodating the necessary regular restarting and recharging of the test platform.

A.2 Cost-Function Evaluation

Recall that CryptOpt generates solutions, mutates them, and measures their respective performance. Measuring this performance on physical machines is inherently noisy. To lower the effect of noise and enable more stable and fair comparisons, we base our measurement on R3-validation, which exercises tests in a rotated-round-robin fashion. This approach was used in previous work [Bokhari et al. 2020] in the context of energy consumption in order to mitigate the effects of measurement noise.

In our work, we adapt R3-validation in two ways. First, we forgo the restart of the computer as we do not observe any measurement drift over time. Second, we perform a random scheduling of program variants for performance measurement instead of strictly following a given order of measurements.

Measuring Performance. Algorithm 3 presents our performance-measurement routine. We now explain the rationale behind this from the inside out (inside the loop of the algorithm to the outside of the measure function).

Cycle counters are integers with a granularity of at best 1. Recall that CryptOpt can optimize functions of arbitrary size, ranging in our experiments from fewer than 100 to around 2000 instructions. That means that the accuracy of the fixed-granularity counter relative to the length-varying function changes. In other words, the same "stopwatch" is not appropriate to time an Olympic sprinter and a marathon runner.

The solution to that is to measure multiple executions of the same function instead of just one. We call multiple executions of the same function a *batch*. The batch size *bs* specifies how many executions of the same function are being measured with one measurement. It follows logically that small functions should have a bigger *bs* than long functions in order to get similar cycle accuracy.

Hence, we came up with the idea of the *cyclegoal*. That is, we want the measured cycles count for one batch across all functions to be in the order of $cyclegoal = 10\,000$ measured cycles. The reason for this particular number is that we then have four to five digits of accuracy. CryptOpt then scales the value for *bs* up or down by comparing the measured cycles against the *cyclegoal*.

This technique also mitigates another inherent problem that arises when measuring performance in cycles on hardware: different hardware platforms interpret "cycles" differently, and additional challenges arise due to different types of boosting. However, because CryptOpt adjusts *bs* in each

Algorithm 3: Execution Time Measurement

```
input  : A pointer to code A,
         B pointer to code B,
         bs size of a measurement batch,
         nob number of batches to measure
output : PA performance of code A in cycles,
         PB performance of code B in cycles

function measure(A, B, bs, nob)
begin
    /* initialize cycle lists                          */
    $cycles_A \leftarrow$ empty list
    $cycles_B \leftarrow$ empty list
    repeat
        /* f points to either A or B                   */
        $f \leftarrow \mathrm{randomSelect}(A,\ B)$

        /* run bs times and get elapsed cycles         */
        $cycles \leftarrow \mathrm{countCyclesForNRuns}(bs, f)$

        /* append elapsed cycles to list A/B resp.     */
        $\mathrm{append}(cycles_f, cycles)$
    until both have been measured nob times;

    $PA \leftarrow \mathrm{median}(cycles_A)$
    $PB \leftarrow \mathrm{median}(cycles_B)$

    return PA, PB
end
```

evaluation of the mutation (i.e. before each call to "measure") and it does so on each platform independently, we get comparable accuracy for all functions across platforms.

Inherent with measuring time on hardware is measurement noise, which we as developers can hardly control. This noise can be due to the OS's interrupt event handlers or process scheduling. It can also be due to temperature rises the hardware limits itself. The technique above mitigates the problems having a bad "stopwatch" but at the same time increases the captured noise. Similar to the R3-validation, we mitigate this problem by simply measuring each batch multiple times, called *numbers of batches* or *nob* for short. That means that we have *nob* measurements for the same function. We take the median as the performance measurement for a function to drop outlier measurements. (Using the minimum rather than median did not have statistically significant effects – neither positive nor negative.) Empirically, setting $nob = 31$ works well on our experimental platforms.

Complex processors nowadays can cause biases to one or another function by their speculative execution, prediction behavior and caching, just to name a few reasons. The R3-validation also mitigates this issue by using a random sequence of the functions to measure. We do this by randomly selecting either function to measure a batch of.

A.3 AssemblyLine

CryptOpt uses AssemblyLine [0xAde1a1de 2022] to assemble the generated initial and mutated code into memory positions *A* and *B*. AssemblyLine is a lightweight in-memory assembler for

Table 4. Optimization results. We show the relative improvements in % for the multiplication (top) and squaring (bottom) operations; time savings are marked in blue. First, to observe hardware-specific optimization, the 8-by-8 matrix shows the performance the optimized operation that have been optimized on one machine and then run on another. The subsequent two rows (Clang/GCC) then show the time savings of our optimized operations over off-the-shelf-compilers. Lastly, "Final" shows the time savings of our best-performing implementation over the best-performing compiler-generated version.

Curve25519

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.14	1.08	1.22	1.20	1.20	1.24	1.11	1.15
5800X	1.05		0.96	1.10	1.12	1.15	1.21	1.08	1.08
5950X	1.05	1.05		1.11	1.21	1.14	1.25	1.07	1.11
i7 6G	1.07	1.18	1.13		1.02	1.08	1.11	1.16	1.09
i7 10G	1.01	1.11	1.06	0.98		1.00	1.07	1.10	1.04
i9 10G	1.06	1.15	1.09	0.98	1.00		1.10	1.24	1.07
i7 11G	1.01	1.09	1.04	1.06	1.08	1.09		1.08	1.05
i9 12G	1.11	1.17	1.11	1.15	1.19	1.17	1.21		1.14
Clang	1.14	1.25	1.19	1.30	1.31	1.34	1.33	1.14	1.25
GCC	0.98	1.13	1.07	1.20	1.21	1.22	1.29	1.18	1.16
Final	0.98	1.13	1.11	1.23	1.21	1.22	1.29	1.14	1.16
1900X		1.15	1.12	1.12	1.11	1.08	1.17	1.15	1.11
5800X	0.98		0.98	1.11	1.09	1.10	1.14	1.16	1.07
5950X	1.01	1.00		1.10	1.09	1.10	1.13	1.16	1.07
i7 6G	1.13	1.20	1.18		0.99	1.02	1.12	1.26	1.11
i7 10G	0.95	1.08	1.06	0.94		0.91	1.05	1.34	1.03
i9 10G	1.05	1.17	1.16	1.04	1.03		1.17	1.36	1.12
i7 11G	1.06	1.14	1.12	1.06	1.05	1.06		1.29	1.09
i9 12G	1.02	1.14	1.12	1.08	1.09	1.04	1.11		1.07
Clang	1.04	1.15	1.12	1.20	1.20	1.15	1.17	1.22	1.16
GCC	1.01	1.17	1.19	1.27	1.25	1.22	1.33	1.16	1.20
Final	1.07	1.15	1.14	1.28	1.21	1.27	1.17	1.16	1.18

P-224

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.14	1.11	1.19	1.18	1.22	1.27	1.03	1.14
5800X	1.03		0.99	1.13	1.13	1.17	1.14	0.96	1.07
5950X	1.03	1.01		1.14	1.15	1.19	1.19	1.01	1.09
i7 6G	1.00	1.04	1.01		1.00	1.04	1.06	0.87	1.00
i7 10G	0.98	1.03	1.01	1.00		1.07	1.09	0.86	1.00
i9 10G	0.99	1.00	0.98	0.96	0.97		1.05	0.86	0.98
i7 11G	0.96	1.02	1.00	1.05	1.07	1.09		0.85	1.00
i9 12G	1.12	1.21	1.19	1.23	1.24	1.28	1.24		1.19
Clang	1.42	1.48	1.46	1.55	1.58	1.62	1.62	1.35	1.51
GCC	2.21	2.11	2.07	2.90	2.90	3.00	2.82	2.63	2.55
Final	1.49	1.48	1.48	1.61	1.63	1.62	1.62	1.58	1.56
1900X		1.14	1.14	1.21	1.20	1.11	1.26	1.11	1.14
5800X	1.05		0.99	1.19	1.17	1.09	1.21	1.06	1.09
5950X	1.03	0.98		1.13	1.11	1.03	1.11	1.02	1.05
i7 6G	1.02	1.02	1.02		0.99	0.95	1.07	0.91	1.00
i7 10G	1.01	1.01	1.02	1.06		0.93	1.06	0.91	1.00
i9 10G	1.08	1.09	1.09	1.09	1.08		1.13	0.95	1.06
i7 11G	1.01	1.00	1.00	1.07	1.05	0.99		0.88	1.00
i9 12G	1.11	1.16	1.15	1.21	1.20	1.10	1.21		1.14
Clang	1.26	1.28	1.28	1.45	1.44	1.33	1.42	1.26	1.34
GCC	2.24	2.01	2.01	2.88	2.83	2.62	2.72	2.60	2.47
Final	1.26	1.30	1.29	1.45	1.46	1.44	1.42	1.43	1.38

P-256

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.00	1.02	1.05	1.08	1.05	1.11	0.98	1.04
5800X	1.24		1.03	1.11	1.15	1.11	1.12	1.01	1.09
5950X	1.12	0.98		1.08	1.12	1.09	1.12	1.03	1.07
i7 6G	1.08	0.99	1.01		1.00	1.03	1.07	0.95	1.02
i7 10G	1.05	0.98	1.00	0.97		0.97	1.07	0.94	1.00
i9 10G	1.05	1.00	1.02	0.96	0.99		1.07	0.92	1.00
i7 11G	1.07	0.99	0.99	1.01	1.04	1.01		0.93	1.00
i9 12G	1.12	1.07	1.10	1.10	1.17	1.11	1.14		1.10
Clang	1.59	1.59	1.63	1.60	1.66	1.61	1.74	1.61	1.63
GCC	2.38	2.16	2.20	2.89	2.97	2.89	2.93	2.97	2.65
Final	1.59	1.63	1.64	1.67	1.67	1.66	1.74	1.76	1.67
1900X		1.08	1.08	1.15	1.11	1.09	1.14	1.01	1.08
5800X	1.05		1.00	1.13	1.09	1.08	1.12	1.04	1.06
5950X	1.07	1.01		1.16	1.12	1.10	1.16	1.07	1.08
i7 6G	1.05	1.04	1.05		0.96	0.95	1.08	0.95	1.01
i7 10G	1.05	1.06	1.07	0.99		0.95	1.07	0.96	1.02
i9 10G	1.08	1.10	1.10	1.05	1.05		1.12	0.98	1.06
i7 11G	1.05	1.02	1.02	1.08	1.05	1.03		0.94	1.02
i9 12G	1.12	1.16	1.17	1.19	1.14	1.13	1.15		1.13
Clang	1.49	1.50	1.50	1.57	1.54	1.50	1.59	1.48	1.52
GCC	2.33	2.14	2.15	2.93	2.79	2.77	2.76	2.51	2.53
Final	1.49	1.50	1.50	1.58	1.60	1.58	1.59	1.58	1.55

P-384

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		0.97	0.98	1.67	1.45	1.68	1.40	1.01	1.24
5800X	1.08		1.02	1.80	1.57	1.79	1.59	1.10	1.33
5950X	1.07	0.98		1.67	1.46	1.69	1.61	1.10	1.29
i7 6G	0.78	0.74	0.75		0.87	1.03	1.05	0.68	0.85
i7 10G	0.83	0.78	0.80	1.16		1.14	1.13	0.76	0.94
i9 10G	0.76	0.72	0.74	1.00	0.85		1.04	0.68	0.84
i7 11G	0.75	0.71	0.72	1.01	0.87	1.01		0.64	0.83
i9 12G	1.07	1.03	1.05	1.66	1.43	1.66	1.53		1.27
Clang	0.99	1.03	1.05	1.45	1.27	1.45	1.53	1.05	1.21
GCC	1.63	1.34	1.37	2.85	2.49	2.87	3.00	1.86	2.07
Final	1.32	1.46	1.45	1.45	1.49	1.45	1.53	1.65	1.47
1900X		0.94	0.96	1.65	1.60	1.68	1.52	1.02	1.26
5800X	1.17		1.02	1.78	1.73	1.81	1.80	1.07	1.38
5950X	1.09	0.99		1.80	1.75	1.83	1.56	1.10	1.34
i7 6G	0.78	0.72	0.73		0.98	1.02	1.06	0.68	0.86
i7 10G	0.79	0.73	0.75	1.01		1.02	1.05	0.67	0.87
i9 10G	0.79	0.72	0.73	0.98	0.96		1.05	0.64	0.85
i7 11G	0.79	0.71	0.72	1.03	0.99	1.03		0.68	0.86
i9 12G	1.06	0.99	1.01	1.68	1.63	1.71	1.58		1.29
Clang	0.98	0.94	0.95	1.41	1.37	1.44	1.48	0.98	1.17
GCC	1.76	1.32	1.34	2.98	2.90	3.03	3.26	1.70	2.15
Final	1.25	1.32	1.31	1.44	1.44	1.44	1.48	1.53	1.40

SIKEp434

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		0.93	0.96	1.65	1.69	1.64	1.54	0.97	1.25
5800X	1.16		0.99	1.78	1.87	1.78	1.80	1.04	1.38
5950X	1.13	0.98		1.80	1.85	1.80	1.78	1.03	1.37
i7 6G	0.59	0.63	0.65		1.02	1.01	1.00	0.61	0.79
i7 10G	0.62	0.63	0.65	0.97		0.97	1.01	0.59	0.78
i9 10G	0.61	0.66	0.68	1.01	1.03		1.07	0.65	0.82
i7 11G	0.60	0.64	0.66	1.02	1.05	1.01		0.63	0.80
i9 12G	1.13	0.99	1.01	1.80	1.84	1.80	1.72		1.36
Clang	1.01	1.19	1.22	1.73	1.78	1.74	2.10	1.26	1.46
GCC	1.43	1.32	1.35	2.91	2.98	2.91	3.54	1.68	2.10
Final	1.73	1.89	1.87	1.79	1.78	1.79	2.10	2.15	1.88
1900X		0.95	0.96	1.59	1.84	1.46	1.66	0.98	1.26
5800X	1.15		1.01	1.72	2.01	1.55	1.76	1.07	1.36
5950X	1.12	0.99		1.73	2.01	1.57	1.79	1.05	1.35
i7 6G	0.62	0.67	0.69		1.17	0.91	1.05	0.65	0.82
i7 10G	0.56	0.63	0.64	0.86		0.78	0.94	0.57	0.73
i9 10G	0.72	0.74	0.75	1.10	1.28		1.17	0.72	0.91
i7 11G	0.57	0.64	0.65	0.87	1.01	0.78		0.61	0.75
i9 12G	1.12	1.01	1.02	1.69	1.97	1.53	1.70		1.33
Clang	1.07	1.30	1.31	1.64	1.90	1.48	1.89	1.20	1.44
GCC	1.51	1.38	1.40	2.80	3.26	2.53	3.75	1.65	2.13
Final	1.90	2.05	2.05	1.90	1.90	1.90	2.02	2.11	1.98

Curve448

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.08	1.06	1.32	1.35	1.24	1.31	0.92	1.15
5800X	0.94		0.98	1.23	1.26	1.13	1.17	0.91	1.07
5950X	0.94	1.02		1.31	1.32	1.16	1.25	0.92	1.10
i7 6G	0.87	0.95	0.93		1.02	0.92	0.99	0.75	0.92
i7 10G	0.85	0.95	0.92	0.98		0.90	0.98	0.75	0.91
i9 10G	0.89	0.98	0.97	1.12	1.11		1.13	0.84	1.00
i7 11G	0.90	1.01	0.98	1.13	1.16	1.06		0.78	0.99
i9 12G	1.15	1.24	1.23	1.57	1.62	1.45	1.51		1.33
Clang	0.79	0.78	0.76	1.23	1.25	1.14	1.11	0.83	0.96
GCC	0.69	0.73	0.71	1.00	1.03	0.93	0.89	0.65	0.82
Final	0.81	0.77	0.77	1.02	1.03	1.02	0.91	0.88	0.89
1900X		1.06	1.06	1.23	1.17	1.21	1.14	0.94	1.10
5800X	1.06		1.00	1.32	1.28	1.30	1.24	1.06	1.15
5950X	1.06	1.01		1.27	1.23	1.25	1.27	1.00	1.13
i7 6G	0.99	1.05	1.06		1.02	0.99	1.06	0.94	1.01
i7 10G	0.99	1.03	1.04	1.06		1.04	1.14	0.99	1.03
i9 10G	0.97	1.06	1.06	0.98	0.94		1.04	0.93	1.00
i7 11G	0.99	1.03	1.03	1.13	1.08	1.15		0.97	1.05
i9 12G	1.10	1.13	1.12	1.37	1.31	1.34	1.34		1.21
Clang	0.78	0.82	0.82	1.02	0.97	1.00	0.98	0.73	0.88
GCC	0.79	0.84	0.86	1.01	0.96	1.00	0.95	0.75	0.89
Final	0.80	0.82	0.82	1.03	1.02	1.01	0.95	0.79	0.90

P-521

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.00	0.99	1.38	1.23	1.15	1.17	0.96	1.10
5800X	0.97		1.00	1.42	1.27	1.18	1.19	0.93	1.11
5950X	0.99	0.99		1.36	1.23	1.13	1.16	0.99	1.10
i7 6G	0.88	0.98	0.97		0.89	0.84	0.91	0.85	0.91
i7 10G	0.90	0.96	0.96	1.05		0.88	0.93	0.95	0.95
i9 10G	0.95	1.05	1.05	1.24	1.06		1.07	0.99	1.05
i7 11G	0.92	0.97	0.97	1.24	1.10	1.03		0.92	1.02
i9 12G	1.09	1.16	1.17	1.65	1.46	1.37	1.36		1.27
Clang	1.05	1.08	1.07	1.57	1.39	1.32	1.26	1.27	1.24
GCC	0.92	1.02	1.01	1.18	1.05	0.99	0.96	0.90	1.00
Final	1.05	1.05	1.05	1.18	1.18	1.17	1.06	1.06	1.10
1900X		1.03	1.00	1.14	1.14	1.14	1.04	1.04	1.06
5800X	1.08		1.00	1.19	1.19	1.19	1.06	1.12	1.10
5950X	1.11	1.02		1.27	1.22	1.22	1.09	1.03	1.12
i7 6G	1.09	1.11	1.09		1.00	1.04	1.04	1.08	1.06
i7 10G	1.06	1.07	1.05	1.00		1.07	1.03	1.06	1.04
i9 10G	1.09	1.12	1.10	0.99	1.00		1.05	1.07	1.05
i7 11G	1.08	1.05	1.03	1.12	1.12	1.12		1.07	1.07
i9 12G	1.17	1.13	1.10	1.32	1.31	1.32	1.26		1.19
Clang	1.28	1.22	1.19	1.65	1.63	1.63	1.72	1.19	1.42
GCC	1.12	1.09	1.07	1.16	1.15	1.17	1.04	1.09	1.11
Final	1.12	1.09	1.07	1.16	1.16	1.17	1.04	1.09	1.11

Poly1305

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.12	1.09	1.18	1.19	1.16	1.11	1.14	1.12
5800X	1.19		1.01	1.14	1.16	1.12	1.10	1.09	1.10
5950X	1.16	0.99		1.14	1.13	1.10	1.14	1.16	1.10
i7 6G	1.20	1.12	1.11		1.02	0.99	1.04	1.29	1.09
i7 10G	1.18	1.12	1.12	0.99		0.98	1.08	1.32	1.09
i9 10G	1.21	1.14	1.11	1.02	1.03		1.06	1.35	1.11
i7 11G	1.22	1.10	1.06	1.09	1.10	1.07		1.29	1.11
i9 12G	1.16	1.08	1.06	1.09	1.11	1.07	1.09		1.08
Clang	1.09	1.08	1.04	1.16	1.14	1.11	1.08	1.15	1.11
GCC	1.10	1.13	1.09	1.24	1.25	1.21	1.15	1.17	1.17
Final	1.09	1.10	1.04	1.18	1.14	1.13	1.08	1.15	1.11
1900X		1.18	1.15	1.09	1.05	1.16	1.15	1.24	1.13
5800X	1.09		0.98	1.02	0.99	1.09	1.06	1.16	1.05
5950X	1.13	1.00		1.02	1.06	1.09	1.12	1.11	1.07
i7 6G	1.14	1.12	1.07		0.91	1.00	1.03	1.32	1.07
i7 10G	1.20	1.16	1.12	0.94		1.00	1.03	1.36	1.10
i9 10G	1.15	1.16	1.12	0.94	0.91		1.07	1.38	1.08
i7 11G	1.09	1.12	1.10	0.99	0.96	1.06		1.32	1.08
i9 12G	1.11	1.08	1.05	1.00	0.99	1.06	1.06		1.04
Clang	0.98	1.09	1.03	0.98	0.95	1.06	1.03	1.16	1.03
GCC	1.05	1.23	1.18	1.16	1.13	1.23	1.21	1.22	1.18
Final	0.98	1.09	1.05	1.05	1.04	1.06	1.03	1.16	1.06

secp256k1

run on	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
opt on									
1900X		1.13	1.14	1.23	1.28	1.19	1.22	1.09	1.16
5800X	0.97		1.00	1.13	1.16	1.10	1.07	0.93	1.04
5950X	0.99	0.99		1.18	1.25	1.11	1.17	1.02	1.09
i7 6G	0.93	1.03	1.04		1.03	0.99	1.02	0.88	0.99
i7 10G	0.93	1.01	1.02	0.97		0.95	0.99	0.86	0.96
i9 10G	0.95	1.06	1.08	1.03	1.06		1.03	0.90	1.01
i7 11G	0.93	1.06	1.07	1.07	1.09	1.06		0.87	1.02
i9 12G	1.06	1.18	1.20	1.26	1.29	1.22	1.17		1.17
Clang	1.39	1.63	1.63	1.70	1.75	1.67	1.68	1.43	1.61
GCC	2.12	1.85	1.87	2.70	2.77	2.62	2.40	2.31	2.30
Final	1.50	1.65	1.63	1.75	1.75	1.76	1.69	1.67	1.67
1900X		1.20	1.17	1.25	1.27	1.24	1.27	1.24	1.20
5800X	0.94		0.98	1.11	1.12	1.10	1.14	1.02	1.05
5950X	0.98	1.03		1.13	1.14	1.12	1.12	1.02	1.06
i7 6G	0.91	1.01	0.98		0.99	0.97	1.00	0.89	0.97
i7 10G	0.91	1.01	0.99	0.99		0.98	1.06	0.90	0.98
i9 10G	0.94	1.06	1.05	1.01	1.02		1.05	0.94	1.01
i7 11G	0.96	1.06	1.05	1.08	1.09	1.07		0.93	1.03
i9 12G	1.04	1.19	1.16	1.21	1.22	1.19	1.17		1.14
Clang	1.25	1.55	1.52	1.62	1.63	1.61	1.56	1.37	1.51
GCC	1.81	1.92	1.87	2.63	2.65	2.59	2.39	2.23	2.24
Final	1.38	1.55	1.56	1.64	1.65	1.66	1.56	1.53	1.56

x86-64 assembly instructions. It takes an input string of instructions, assembles them into machine code and returns a pointer to the executable code. We use AssemblyLine to eliminate the overhead of writing to the file system and invoking a typical compiler tool chain for every evaluation. Additionally, CryptOpt uses AssemblyLine to assemble the code to the start of a memory page. This reduces noise in measurements because it always assembles aligned code for all functions, which, in turn, reduces memory-biased performance impacts [Mytkowicz et al. 2009]. Once assembled, CryptOpt calls the measure function with pointers to that code and values for batch size *bs* and number of batches *nob*.

B PLATFORM-SPECIFIC OPTIMIZATION

CryptOpt optimizes the execution time of the function on the platform it executes on. Because different platforms have different hardware components, the fastest code on one platform is not necessarily the fastest on another. Thus, in this section we ask whether CryptOpt overfits to the platform it executes on, as opposed to universally optimizing for all platforms. That is, we test how *native optimization*, where the code executes on the same platform it was optimized on, compares with *cross optimization*, where code is optimized on one platform and executes on another.

To test this, we first optimize each of the functions on each of the platforms. We then measure the performance of the produced code from all platforms on all platforms (as per evaluation described in Appendix A). Table 4 shows the results for the operations multiply and square on all nine fields we tested. Rows correspond to the platform the code was optimized on and columns to the platform the code is executed on. The value in a cell shows the ratio between the execution time of the cross-optimized code and that of the native code. An example value of 1.10 indicates that the cross-platform code (from the "row" platform) needs 1.10 times as many cycles as the native code (from the "column" platform). In other words, a blue cell means that code from the column architecture outperforms code from the row architecture, and the number indicates by how much.

The additional column G.M. shows the geometric mean of the numbers in the row. This gives a measure of "how platform-specific" one optimization is: A value larger than one (blue) in column G.M. in the row for platform X indicates that on average, code from other platforms outperform the code from X, if ran on platforms other than X. In turn, a value smaller than one means that the code from platform X tends to outperform platform-specific code on other platforms. Please note that a "large number" can arise for two reasons: (a) The platform-specific code is very bad (thus easy to outperform), or (b) the generated code is very generic and can thus compete with the platform-specific ones. To see which is applicable per platform, one needs to see how well the code compares to off-the-shelf compilers. Note that this is also not a universal measure, as off-the-shelf compilers generate different code per platform. Alternatively, running the same code on all platforms would also not give a fair baseline, as compilers would not be enabled to generate platform-specific code, whereas CryptOpt would be.

As can be seen in Table 4 for Curve25519, optimizing and running on the same architecture tends to outperform cross-architecture optimizations (which would mean the 8-by-8 is mostly blue, and the G.M. column is fully blue, too). However, there are cases where cross-architectural optimizations are better. In particular it appears that optimization of P-384 functions on AMD is not ideal, and the results underperform code optimized on Intel processors. We note that measurement on AMD machines is less stable than on Intel machines (see 5950X and i9 10G in Figure 8). We therefore hypothesize that although optimization is possible on the AMD machines, in some cases the local search may drift in the configuration space, missing optimization opportunities and producing less performant code (still, for the majority faster than compiling with off-the-shelf compilers).

Another effect that the table highlights is that optimization overfits for processor families, for some families more than others. In particular, Intel 6th and 10th generations show little differences in the respective inter-platform performance. For example, the related 3x3 sub-matrices for Curve25519 in Table 4 have values in the small interval [0.98, 1.08] (mul), [0.91, 1.04] (square); as do AMD Zen 3 processors in the 2x2 sub-matrices have values in the small interval [0.96, 1.05] (mul), [0.98, 1.00] (square) .

This specialization to platforms is a double-edged sword. On the one hand, it allows CryptOpt to produce high-performing code, as previously shown. On the other hand, it also means that the code is potentially less generic, and it might be less performant if executed on platforms other than originally intended, as observed in our a-posteriori analysis.

C DETAILED PERFORMANCE INFORMATION

Table 5 shows the detailed performance information of the scalar multiplication experiments. The table is divided in three sections, each for a different curve: Curve25519, NIST P-256 and `libsecp256k1`. In each of those sections we compare different implementations of the same curve against each other. The language, in which the core operations are implemented, is written in the column *Lang*. The following columns show the elapsed cycles for the scalar multiplication. The fastest implementation per section and per platform is highlighted in bold text. All other implementations are then compared against this fastest in the form of the ratio, which is written in parenthesis. E.g. 1.05x means, that it takes 1.05 times as many cycles than the fastest. The last column, G.M., is the geometric mean of the cycles across all platforms, the ratio is then recalculated on that G.M. as well.

Table 5. Cost of scalar multiplication (in cycles) of different implementations benchmarking on different machines.

	Implementation	Lang.	1900X	5800X	5950X	i7 6G	i7 10G	i9 10G	i7 11G	i9 12G	G.M.
Curve25519	sandy2x [Chou 2015]	asm-v	604k (1.05x)	**426k (1.00x)**	**426k (1.00x)**	521k (1.12x)	521k (1.12x)	521k (1.12x)	492k (1.06x)	480k (1.16x)	496k (1.07x)
	amd64-64 [Chen et al. 2014]	asm	622k (1.08x)	576k (1.35x)	574k (1.35x)	570k (1.23x)	571k (1.23x)	572k (1.23x)	536k (1.16x)	455k (1.10x)	558k (1.20x)
	amd64-51 [Chen et al. 2014]	asm	772k (1.34x)	584k (1.37x)	585k (1.37x)	567k (1.22x)	566k (1.22x)	567k (1.22x)	501k (1.08x)	423k (1.02x)	564k (1.21x)
	donna [Langley 2022]	asm-v	1081k (1.88x)	959k (2.25x)	969k (2.27x)	1013k (2.18x)	1015k (2.18x)	1016k (2.18x)	955k (2.06x)	886k (2.15x)	985k (2.12x)
	donna-c64 [Langley 2022]	C	865k (1.51x)	689k (1.62x)	689k (1.62x)	763k (1.64x)	764k (1.64x)	764k (1.64x)	715k (1.54x)	573k (1.39x)	723k (1.56x)
	O'SSL [OpenSSL 2022]	C	694k (1.21x)	538k (1.26x)	538k (1.26x)	564k (1.21x)	564k (1.21x)	565k (1.21x)	499k (1.08x)	428k (1.04x)	545k (1.17x)
	O'SSL fe-51 [OpenSSL 2022]	asm	707k (1.23x)	519k (1.22x)	520k (1.22x)	585k (1.26x)	585k (1.26x)	586k (1.26x)	522k (1.13x)	443k (1.07x)	554k (1.19x)
	O'SSL fe-64 [OpenSSL 2022]	asm	**574k (1.00x)**	442k (1.04x)	442k (1.04x)	**465k (1.00x)**	**465k (1.00x)**	**466k (1.00x)**	**463k (1.00x)**	416k (1.01x)	**465k (1.00x)**
	O'SSL 51+**CryptOpt**	asm	736k (1.28x)	523k (1.23x)	532k (1.25x)	559k (1.20x)	542k (1.17x)	583k (1.25x)	498k (1.08x)	**413k (1.00x)**	542k (1.17x)
	Boring [BoringSSL 2022]	C	900k (1.57x)	647k (1.52x)	639k (1.50x)	728k (1.57x)	735k (1.58x)	728k (1.56x)	616k (1.33x)	522k (1.26x)	682k (1.47x)
	Boring+**CryptOpt**	asm	827k (1.44x)	629k (1.48x)	633k (1.49x)	659k (1.42x)	640k (1.38x)	662k (1.42x)	576k (1.24x)	489k (1.18x)	633k (1.36x)
P-256	z-256 [Gueron and Krasnov 2013]	asm-v	**736k (1.00x)**	**566k (1.00x)**	**564k (1.00x)**	**614k (1.00x)**	**622k (1.00x)**	**622k (1.00x)**	**588k (1.00x)**	**572k (1.00x)**	**608k (1.00x)**
	Boring C-Fallback	C	1418k (1.93x)	1249k (2.21x)	1249k (2.21x)	1253k (2.04x)	1253k (2.02x)	1261k (2.03x)	1164k (1.98x)	1034k (1.81x)	1231k (2.02x)
	Boring+**CryptOpt**	asm	1219k (1.66x)	1081k (1.91x)	1056k (1.87x)	1077k (1.75x)	1072k (1.72x)	1077k (1.73x)	985k (1.68x)	891k (1.56x)	1053k (1.73x)
secp256k1	O'SSL	C	6245k (8.92x)	4868k (8.65x)	4829k (8.55x)	5900k (10.16x)	5952k (10.35x)	5906k (10.27x)	4988k (9.28x)	3751k (8.28x)	5242k (9.26x)
	O'SSL+**CryptOpt**	asm	2862k (4.09x)	2308k (4.10x)	2332k (4.13x)	2239k (3.85x)	2231k (3.88x)	2264k (3.94x)	2120k (3.94x)	1828k (4.04x)	2258k (3.99x)
	libsecp256k1 [Bitcoin Core 2022]	asm	740k (1.06x)	588k (1.05x)	591k (1.05x)	607k (1.05x)	606k (1.05x)	606k (1.05x)	584k (1.09x)	471k (1.04x)	595k (1.05x)
	libsecp256k1 [Bitcoin Core 2022]	C	**700k (1.00x)**	**563k (1.00x)**	**565k (1.00x)**	**581k (1.00x)**	**575k (1.00x)**	**575k (1.00x)**	**538k (1.00x)**	458k (1.01x)	**566k (1.00x)**
	libsecp256k1+**CryptOpt**	asm	750k (1.07x)	567k (1.01x)	588k (1.04x)	600k (1.03x)	585k (1.02x)	593k (1.03x)	545k (1.01x)	**453k (1.00x)**	580k (1.03x)

Note: G.M. stands for geometric mean; asm means assembly; -v indicates the use of vector instructions.