CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives

JOEL KUEPPER, University of Adelaide, Australia
ANDRES ERBSEN, Massachusetts Institute of Technology, USA
JASON GROSS, Massachusetts Institute of Technology, USA
OWEN CONOLY, Massachusetts Institute of Technology, USA
CHUYUE SUN, Stanford University, USA
SAMUEL TIAN, Massachusetts Institute of Technology, USA
DAVID WU, University of Adelaide, Australia
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
CHITCHANOK CHUENGSATIANSUP, The University of Melbourne, Australia
DANIEL GENKIN, Georgia Institute of Technology, USA
MARKUS WAGNER, Monash University, Australia
YUVAL YAROM, Ruhr University Bochum, Germany

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. The overall prototype is quite practical, e.g. producing new fastest-known implementations of finite-field arithmetic for both Curve25519 (part of the TLS standard) and the Bitcoin elliptic curve secp256k1 for the Intel 12^{th} and 13^{th} generations.

CCS Concepts: • **Software and its engineering** → **Source code generation**; • **Security and privacy** → *Public key (asymmetric) techniques*; **Logic and verification**; • **General and reference** → **Performance**; *Measurement*; • **Theory of computation** → *Cryptographic primitives.*

Additional Key Words and Phrases: elliptic-curve cryptography, assembly, search-based software engineering

Authors' addresses: Joel Kuepper, University of Adelaide, Australia; Andres Erbsen, Massachusetts Institute of Technology, USA; Jason Gross, Massachusetts Institute of Technology, USA; Owen Conoly, Massachusetts Institute of Technology, USA; Chuyue Sun, Stanford University, USA; Samuel Tian, Massachusetts Institute of Technology, USA; David Wu, University of Adelaide, Australia; Adam Chlipala, Massachusetts Institute of Technology, USA; Chitchanok Chuengsatiansup, The University of Melbourne, Australia; Daniel Genkin, Georgia Institute of Technology, USA; Markus Wagner, Monash University, Australia; Yuval Yarom, Ruhr University Bochum, Germany.

2475-1421/2023/6-ART158
https://doi.org/10.1145/3591272

ACM Reference Format:
Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. *Proc. ACM Program. Lang.* 7, PLDI, Article 158 (June 2023), 25 pages. https://doi.org/10.1145/3591272

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 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, §V-B], 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 control flow, as it offers higher 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, our work starts from the question:

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

1.1 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 randomized 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 randomized-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 (as shown in Figure 1), we get end-to-end functional-correctness proofs for fast assembly code – faster than any code 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 code generation 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 randomized-search domain to seek 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 is not worse than the original, the mutated implementation is kept; otherwise it is discarded.

Instead of trying to predict code performance like in work by Joshi et al. [2002]; Schkufza et al. [2013], CryptOpt measures actual execution time. This approach is important because it avoids inaccuracies inherent in hardware models and allows CryptOpt to tailor produced code to the target processor while treating the processor itself as an opaque unit. A particular advantage of the approach is that once manufacturers release new hardware, which changes e.g. pipelining effects or caching behavior, CryptOpt adapts to it automatically without requiring manual adaptation of hardware models. This *find-and-optimize* approach is implemented by the *Optimizer* component in Figure 1.

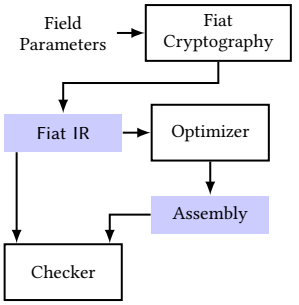

Fig. 1. Integration design. Boxes represent individual components, arrows represent data flow, and filled boxes represent files. *Optimizer* and *Checker* are results of this work. Fiat Cryptography and *Checker* are formally verified.

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 IR programs and generates x86-64 assembly code. To avoid needing to model the randomized-search process, we instead build and prove an equivalence checker, which can compare programs across Fiat 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 [Detlefs et al. 2005] to canonicalize logical expressions via rewrite rules. 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. This *checking* is done by the *Checker* component in Figure 1.

We evaluate CryptOpt using finite fields with nine different prime moduli. We use Fiat Cryptography to generate the Fiat IR for the multiply and square operations for these fields. We then use the CryptOpt optimizer to generate x86-64 assembly code for these operations and the equivalence checker to verify that the code matches the Fiat IR. The produced x86-64 assembly code achieves a mean speedup of 1.74 compared to GCC 12.1.0 (speedup 1.40 against Clang 15.0.6), across ten different x86-64 platforms (four AMD, six Intel).

We further evaluate the CryptOpt optimizer as a stand-alone code generator. For that, we create an input function from the C code of `libsecp256k1` [Bitcoin Core 2021], feed it into the CryptOpt optimizer, and obtain an average speedup of 1.04 against the hand-optimized assembly code in `libsecp256k1`. As we do not have Fiat IR code matching `libsecp256k1`, we cannot use the equivalence checker to verify the code we produce.

1.2 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 5). This broad approach is shown to apply to larger routines than past work has tackled successfully, while also being integrated with formal verification for the first time.
- We demonstrate that a relatively modest Fiat Cryptography extension (Section 6) suffices to enable integration with a wide range of 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.
- We generate formally verified high-performance cryptographic code optimized for ten CPU architectures, obtaining considerable speedups over GCC and Clang (Section 7).

The source code for CryptOpt is available at https://github.com/0xADE1A1DE/CryptOpt.

2 BACKGROUND

In this section we present basic background required for the rest of the paper.

2.1 Random Local Search

A combinatorial optimization problem aims to find an optimal solution (e.g. one that minimizes a given *objective function*) within a discrete set of candidate solutions. Random Local Search (RLS) [Auger and Doerr 2011; Doerr and Neumann 2019] is a simple optimization strategy that is often efficient and effective in finding local optima. A run of RLS starts from a random candidate solution. It then applies a random *mutation* to the current candidate solution, generating another solution within the possible solutions set. If the mutation improves (or does not deteriorate) the solution, the mutated solution is kept, replacing the current candidate solution. Otherwise, the mutation is discarded, and the current candidate solution remains unchanged. This mutation and evaluation repeats until some predefined termination condition is satisfied.

RLS is often highly sensitive to the initial conditions, i.e. the candidate solution it starts from. 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.

2.2 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).

Because the FFA is a performance-critical component in the implementations, many tend to implement it by hand. For instance, 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. In particular, a field element is typically represented by multiple limbs using several CPU registers, and thus every field operation requires multiple CPU cycles. However, these implementations tend to be straight-line code, heavy on arithmetic rather than control flow, leading to ineffective optimization by standard compilers. Human experts instead manually apply 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.

2.3 Fiat Cryptography

Erbsen et al. [2019] present the Fiat Cryptography framework, which translates descriptions of field arithmetic into code with Coq proof of functional correctness. The starting point is a library of functional programs that are used as templates for generating code for performing operations in finite fields. These functional programs, which have been proven correct, can be specialized with a specific prime order for generating an intermediate representation (Fiat IR) of the code that performs field arithmetic operations for the required field.

In execution, Fiat Cryptography selects the functional program to specialize for the required field size and produces provably correct Fiat IR. It then uses one of the available backends to process the Fiat IR code and produce an implementation in one of the supported languages, including C, Java, and Zig.

2.4 Equivalence Checking

Formally proved compilers are the gold standard to address concerns of optimization soundness. 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 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 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.

2.5 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 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.

Figure 2 animates a simple example. It steps through stages of adding a new node to the graph, representing the new 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.

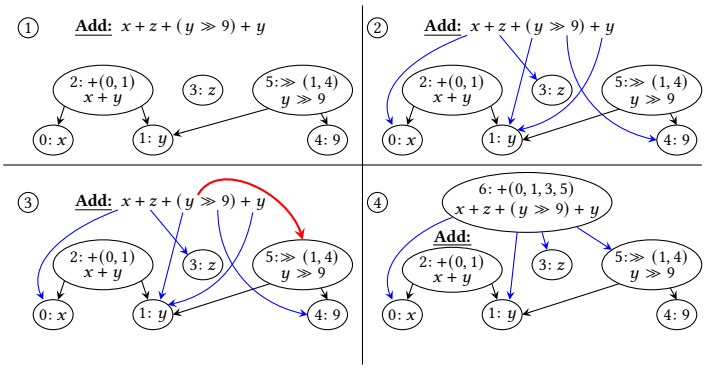

Fig. 2. Example of operations on an E-graph-style structure

3 RELATED WORK

With CryptOpt we combine several known techniques to generate fast and formally verified code. We now sketch related work in the areas of genetic improvement (GI), optimal optimization pass finding for off-the-shelf compilers, superoptimization, peephole optimization, translation validation and computer-aided cryptography.

3.1 Genetic Improvement

CryptOpt applies GI, an area within search-based software engineering [Harman and Jones 2001] that automatically searches out improved software versions. 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.

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 straight-line 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). Their approach can improve the performance of existing code and verify the correctness of the produced code. Their starting point is handwritten assembly code within a relatively shallow metaprogramming framework. In contrast, we show that randomized 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. Generating the code allows CryptOpt additional flexibility, with support for optimizing register allocation (in particular spills to memory) and instruction selection. For example, using their reordering transformation, they cannot substitute an add-using-overflow `adox` for an add-using-carry `adcx`, let alone have 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). CryptOpt also benefits from compatibility with Fiat Cryptography, which makes code generation for finite fields for new primes easy.

3.2 Optimization-Pass Finding

Stephenson et al. [2003] and Peeler et al. [2022] both use GI to select and order existing optimization passes of off-the-shelf compilers 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 apply optimizations selectively at certain locations and avoid using them at others.

3.3 Superoptimization

Massalin [1987] coined the term "superoptimizer" to describe a tool for exhaustive enumeration of all possible programs to implement a given function. The key idea making this feasible is the use of a probabilistic test set, which rejects the majority of incorrect candidates. At the time of writing, it is able to generate programs of 12 instructions after several hours of running (on a 16MHz 68020 computer). Sasnauskas et al. [2017] present Souper, a tool to synthesize new optimizations on the LLVM IR. Working on the IR, by design, they are unable to generate optimizations to exploit target-specific code sequences.

Joshi et al. [2002] present Denali, a superoptimizer for very short programs. They model the architecture of their processor (Alpha EV6) and use solvers to reject conjectures of the form "No program can compute P in at most 8 cycles." Combining this framing with a binary-search algorithm, they end up with the most efficient program. Schkufza et al. [2013] present STOKE, a superoptimizer which is able to synthesize and optimize programs. It combines correctness indicator and performance into a cost function and then randomly (1) changes opcodes, (2) changes arguments, (3) deletes instructions, and (4) inserts `nops`. By starting from scratch, they can find algorithmically different solutions, which cannot be found by other superoptimizers. The resulting programs range up to tens of instructions long. Subsequent work extends STOKE to optimize floating-point kernels [Schkufza et al. 2014], optimize loops [Sharma et al. 2013], and more aggressively optimize kernels based on verified runtime preconditions with cSTOKE [Sharma et al. 2015].

While CryptOpt shares the idea of a randomized search with superoptimization approaches, there are a few important differences between the two. First, superoptimization approaches, and in particular STOKE, apply random changes to existing code, whereas CryptOpt aims to generate the code from a high-level specification. Moreover, unlike superoptimization, CryptOpt only uses semantics-preserving approaches. This significantly reduces the search space, allowing CryptOpt to handle functions with hundreds and even thousands of instructions. As shown in Section 7.4, superoptimizers tend to fail on the inputs that CryptOpt processes. Finally, superoptimizers tend to use a model of the hardware for which they optimize the code. CryptOpt, in contrast, optimizes to the actual hardware, allowing it to adapt to new hardware without the need to model the microarchitecture of the new hardware.

3.4 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 usually done based on a predefined rule set (applying only to short instruction sequences), which itself is based on heuristics for estimating which set of instructions is shorter or more performant. Yet another approach is to find and learn good peephole optimizations automatically: 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 one 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.

3.5 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.

CryptoLine [Chen et al. 2014; Fu et al. 2019; Polyakov et al. 2018; Tsai et al. 2017] is an automatic verification engine utilizing SMT solvers (BOOLECTOR) and computer-algebra systems (Singular), applying to their own IR. The approach to validating assembly files is similar to ours. CryptoLine has only worked via unverified translators from assembly languages to their IR, and the translator must be trusted, unlike ours, though it likely accepts some correct programs that ours rejects.

Last, Sewell et al. [2013] go further and parse the binary code of the seL4-Linux microkernel and transform it until they could prove equivalence to the already-verified C code.

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.

3.6 Real-World Applications of Computer-Aided Cryptography

Provably correct generated code is already deployed in important projects: all major web browsers use finite-field code generated by Fiat Cryptography [Erbsen et al. 2019] (via Google's BoringSSL and

other libraries), and Firefox includes routines arising from the more comprehensive efforts of Project Everest [Bhargavan et al. 2017b], including compilation of nonstraightline code to C [Protzenko et al. 2017], verified metaprogramming of assembly [Fromherz et al. 2019], tying it together in the EverCrypt library [Protzenko et al. 2020], and even adding protocol verification [Bhargavan et al. 2017a]. The Everest stack 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 hand-optimized implementations depending on platform and hardware. We already mentioned the work of Bosamiya et al. [2020] on automatic program search, the only approach to *automatic* assembly generation that we have seen in the Everest ecosystem, and it does not seem to have been applied yet to elliptic curves. CryptOpt also has the usual advantage of proof-assistant work, that, despite our usage of a stack of domain-specific tools, none of them need be trusted.

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.

4 CRYPTOPT OVERVIEW

Our aim is to strengthen Fiat Cryptography to both increase the performance of the produced code and to decrease the size of the trusted code base. To that end, we implement two novel components and integrate them with Fiat Cryptography as sketched in Figure 1.

The first component, the CryptOpt optimizer, is a new backend for Fiat Cryptography, which ingests Fiat IR and produces x86-64 assembly code. A unique and novel property of the optimizer is that instead of relying on classic compiler-optimization techniques, it draws on techniques from the domain of evolutionary computation. Specifically, as illustrated in Figure 3, the CryptOpt optimizer first randomly generates x86-64 assembly code that implements the input function. It then repeatedly mutates the code and measures the execution time of both the original and the mutated code, discarding the slower one. The process continues until a predefined computational budget is used up.

The second component is a new program-equivalence checker, which, given an original Fiat IR program and its optimized assembly, is able to verify behavior preservation with no further hints (c.f. Figure 1). It is codesigned with the CryptOpt optimizer to support the same set of transformations, significantly reducing the complexity compared to a generic equivalence checker. The checker itself is implemented and verified in Coq. Thus, only Fiat Cryptography and the checker are verified, whereas the optimizer itself need not be trusted. While not trusted, the optimizer is designed to only use semantics-preserving transforms. Thus, during normal operation, we expect verification always to succeed.

The combination of these two components allows us to achieve our aims. As we demonstrate, random search can generate code that performs as well as hand-optimized code, significantly surpassing the performance of compiler-produced code. At the same time, the formally verified checker strengthens the unified theorems of compilation to cover the whole span from functional programs to x86-64 assembly instead of Fiat IR, removing the compiler from the trusted code base.

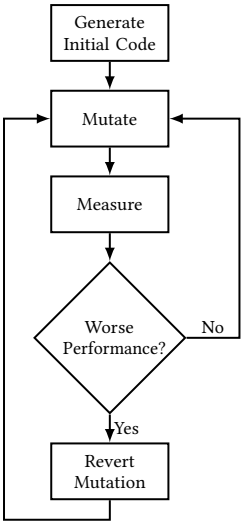

Fig. 3. Optimizer system architecture

While designed as a backend for Fiat Cryptography, CryptOpt can operate as a stand-alone optimizer. To demonstrate this use

case, we transform a C implementation of field arithmetic to LLVM IR using Clang. (Specifically, we use the C implementation of the Bitcoin `libsecp256k1` library.) We then use a simple script to translate the LLVM IR to the input format of CryptOpt and optimize it. Because the C code we use is not formally verified, we skip the formal-verification aspect in this use case, focusing just on further evaluation of the randomized optimizer.

Finally, we note that the generated code from Fiat Cryptography utilizes neither secret-dependent memory accesses nor secret-dependent branching. Consequently, the code follows the constant-time programming paradigm [Almeida et al. 2016], providing protection against microarchitectural side-channel attacks [Ge et al. 2018].

Now we are ready to fill in the details of CryptOpt (randomized search in Section 5 and equivalence checking in Section 6) and how we evaluated it (Section 7).

5 RANDOMIZED SEARCH FOR ASSEMBLY PROGRAMS

The first half of our approach is the CryptOpt optimizer, which generates highly performant x86-64 assembly code that implements an input Fiat IR function. 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, we observe that searching the set of assembly code sequences that implement a given input function is discrete. Hence, the problem of searching this set for the assembly code sequence that minimizes execution time is a combinatorial optimization problem.

For optimization, we employ the RLS strategy with the Bet-and-Run heuristics. Recall that for RLS, we need to first choose a random solution and then repeatedly mutate the solution. While RLS is a relatively simple approach for combinatorial optimization, we find that it is effective and, as we show below, achieves good results. We leave the task of experimenting with more complex optimization strategies to future work. In the rest of this section we present our approach for generating and mutating solutions. We start with a description of the input format and then explain how we generate and mutate code.

5.1 Input Format

Recall (Section 4) that CryptOpt takes Fiat IR as an input. This intermediate language is truly a minimal one (see Figure 4), with the only noteworthy syntactic twist being that integer constants are presented as binary numbers (clearly indicating bitwidth), though we will often abbreviate them in decimal form, when bitwidth is clear from context. Operators may in general take not just multiple operands but also generate multiple results, associated with less common operators like addition with carry or multiplication producing double-wide results via two output words.

Variable	x		
Binary integer	b		
Operand	e	::=	$x \mid b$
Operator	o	::=	! | & | $*$ | + | − | << | = | >> | ~ | or | addcarryx | cmovznz | mulx | static_cast | subborrowx
Expression	E	::=	return $e \mid x, \ldots, x \leftarrow o(e, \ldots, e); E$

Fig. 4. Fiat IR syntax

Input programs contain no branches or 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). The assembly we generate is timing-secure for the same reasons that Fiat IR is timing-secure; we only use primitive program mutations that either preserve relevant behavior (the trace of program memory accesses and control-flow decisions) or add new behaviors in ways independent of secrets (all memory

accesses are to constant offsets from either the stack pointer or arrays that are function parameters), so by induction the initially secure programs are still secure after optimization.

Algorithm 1 shows a Fiat IR program, which we will use throughout this paper as a running example. The program takes three inputs (X, Y, Z) and outputs $Z^2+(Y+Z)\cdot X+Z$ 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 arguments are in the range $0 \le X,\ Y,\ Z < 2^{63}$, allowing us to ignore some carries known to be 0. We mark these carries with $c_\varnothing$.

Algorithm 1: An Example Function

input : X, Y, Z such that $0 \le X, Y, Z < 2^{63}$
output: $O = 2^{64}O_1 + O_0 = Z^2 + (Y+Z)\cdot X + Z$

function *example*$(X,\ Y,\ Z)$
begin
  $t_2,\ t_1 \leftarrow \text{Mul}_1(Z,\ Z)$
  $c_\varnothing,\ t_0 \leftarrow \text{Add}_1(Y,\ Z,\ 0)$
  $t_4,\ t_3 \leftarrow \text{Mul}_2(t_0,\ X)$
  $c_0,\ t_5 \leftarrow \text{Add}_2(t_3,\ t_1,\ 0)$
  $c_1,\ O_0 \leftarrow \text{Add}_3(t_5,\ Z,\ 0)$
  $c_\varnothing,\ t_6 \leftarrow \text{Add}_4(t_4,\ t_2,\ c_0)$
  $c_\varnothing,\ O_1 \leftarrow \text{Add}_5(t_6,\ 0,\ c_1)$
  return $O_1,\ O_0$
end

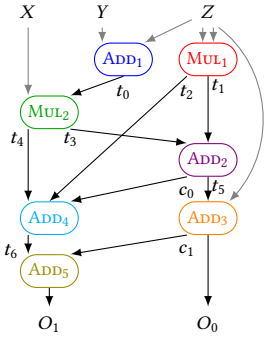

Fig. 5. Data flow of the running example in Algorithm 1

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 5 shows the data-flow graph of the code in Algorithm 1.

5.2 Code Generation

The process of generating those assembly candidates 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 changed by a later random mutation. We summarize the phases here before returning to details of generation and mutation.

First, for instruction scheduling (see Section 5.3), we use data-flow analysis to determine the data-flow dependencies between operations and choose a random topological order of the dependency graph as the initial order of operations in the code. Then, for instruction selection (see Section 5.4), each operation gets assigned a compatible x86-64 assembly instruction template. Finally, in our setting, register allocation (see Section 5.5) arises mostly in ensuring compatibility with operand restrictions of the instruction templates 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 6 shows how operations are

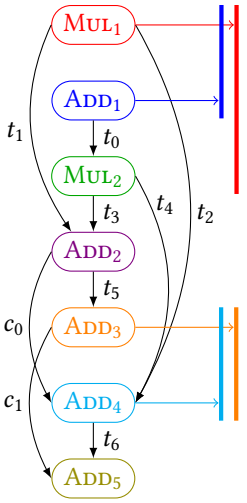

Fig. 6. One ordering. Round rectangles show the operations, which are evaluated top-down. Arrows indicate creation and consumption of intermediate values. Vertical bars show the intervals in which an operation can be scheduled.

initially ordered and potential scheduling intervals; the mutations are shown in Figure 7; until finally we see the emitted code with and without the effects of the mutations in Figure 8. We will reference those illustrations later as needed.

5.3 Instruction Scheduling

Thanks to the simplicity of Fiat IR, every variable is assigned exactly once in the straight-line programs that the CryptOpt optimizer 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.

Initial Ordering. To create an initial ordering, the CryptOpt optimizer simply computes a topological order of the data-flow graph. Figure 6 shows an example of an initial ordering 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.

Mutation Step. An instruction-scheduling mutation step randomly selects one operation in the current ordering. This operation is moved to a randomly chosen location within 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. We begin with the ordering shown in Figure 6. The vertical colored bars indicate the intervals where the respective operations can be scheduled. Step α of Figure 7 shows the effect of moving $\textsc{Add}_4$ up one position.

Selecting the position to move an operation to is biased towards larger moves, i.e. further away from the initial position, in an effort to minimize spills by minimizing distance between computing a value and using it.

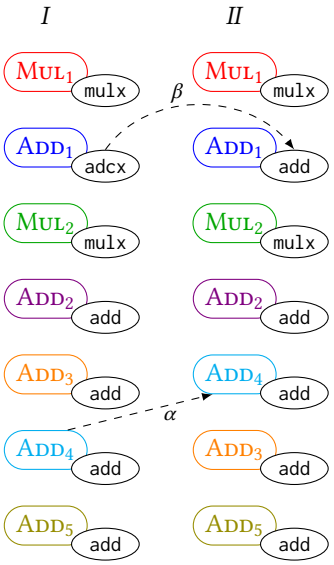

Fig. 7. *I*: Initial ordering of operations (colored rounded rectangles) with attached templates (black ellipses); *II*: After two mutations α and β: mutation α in topological ordering, mutation β in instruction-template selection. Data-flow arrows omitted; dashed arrows indicate mutations.

5.4 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 slightly different semantics and impact on the processor pipeline. To match machine instructions to operations, the CryptOpt optimizer uses templates describing the possible implementations for each operation.

Consider, for example, the Add operation, which can be implemented using multiple instructions, such as `add`, `adcx`, or `adc`. Though semantically equivalent, these choices influence program state differently. For instance, in the case of no input carry ($\textsc{Add}_{1-3}$), 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, unless CryptOpt determines that the carry is already clear. Lines 6 and 7 of Figure 8 show this choice for $\textsc{Add}_1$.

Initial Template Mapping. When creating the initial code, CryptOpt selects a random template for each operation as an initial mapping. Figure 7 (I) shows an example of a possible mapping of templates to the operations of our example function. The figure indicates the operation $\textsc{Mul}_1$ is implemented using the `mulx`, $\textsc{Add}_1$ 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 alternative. Mutation β in Figure 7 shows an example of replacing the template for $\textsc{Add}_1$ to use the `add` instruction instead of the original `adcx`.

Flag Spills. The series of additions $\textsc{Add}_2, \ldots, \textsc{Add}_5$ show not only the influence of different orderings on the resulting code but also how a template is used to handle flag spills. Consider the code in Figure 8a where $\textsc{Add}_2$ results in `CF`=c_0. Later on, $\textsc{Add}_3$ 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 17). Similarly, $\textsc{Add}_4$ then needs to spill c_1 (line 22) to avoid overwriting it with its $c_\emptyset$.

At this point (line 23), 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 $\textsc{Add}$ operation is associative and commutative, any evaluation order maintains correctness. The equivalence checker accounts for this change in evaluation order (see Section 6.3).

Strength Reduction. Some of the templates we use support limited forms of strength reduction. For example, we have templates to implement multiplication by a constant, including using a left-shift operation (i.e. $x \times 8 \implies x \ll 3$), a series of multiplications and additions (i.e. $x \times 5 \implies x \times 2 + x$), or a combination thereof. The final template selection is left to the optimizer.

```
1  | ; t2, t1 ← MUL1(Z, Z)
2  | mov  rdx,  [Z]
3  | mulx r8,   r9,   [Z]
4  |
5  | ; c∅, t0 ← ADD1(Y, Z, 0)
6  | clc
7  | adcx rdx,  [Y]
8  |
9  | ; t4, t3 ← MUL2(t0, X)
10 | mov  [rsp], r8 ; spill
11 | mulx r8,   rdx, [X]
12 |
13 | ; c0, t5 ← ADD2(t3, t1, 0)
14 | add  r9,   rdx
15 |
16 | ; c1, O0 ← ADD3(t5, Z, 0)
17 | setc dl          ; dl ← c0
18 | add  r9,   [Z]
19 | mov  [O0], r9
20 |
21 | ; c∅, t6 ← ADD4(t4, t2, c0)
22 | setc r9b         ; r9b ← c1
23 | movzx rdx, dl
24 | add  rdx,  [rsp]
25 | add  r8,   rdx
26 |
27 | ; c∅, O1 ← ADD5(t6, 0, c1)
28 | movzx r9,  r9b
29 | add  r8,   r9
30 | mov  [O1], r8
```

(a) Code from I (initial code)

```
1  | ; t2, t1 ← MUL1(Z, Z)
2  | mov  rdx,  [Z]
3  | mulx r8,   r9,   [Z]
4  |
5  | ; c∅, t0 ← ADD1(Y, Z, 0)
6  |
7  | add  rdx,  [Y]
8  |
9  | ; t4, t3 ← MUL2(t0, X)
10 | mov  [rsp], r8 ; spill
11 | mulx r8,   rdx, [X]
12 |
13 | ; c0, t5 ← ADD2(t3, t1, 0)
14 | add  rdx,  r9
15 |
16 | ; c∅, t6 ← ADD4(t4, t2, c0)
17 |
18 | adc  r8,   [rsp]
19 |
20 |
21 |
22 | ; c1, O0 ← ADD3(t5, Z, 0)
23 |
24 | add  rdx,  [Z]
25 | mov  [O0], rdx
26 |
27 | ; c∅, O1 ← ADD5(t6, 0, c1)
28 |
29 | adc  r8,   0
30 | mov  [O1], r8
```

(b) Code from II (code after two mutations)

Fig. 8. Emitted assembly code. Highlighted lines show effects from mutations.

5.5 Register Allocation

The register-allocation step of CryptOpt achieves two aims. It must both decide which values are assigned to registers and which registers to spill to memory when running out of registers. CryptOpt uses randomized search for the former and a deterministic strategy for the latter.

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 registers and spills the register whose next use is furthest based on the current operation order.

For example, lines 10–11 in Figure 8 implement the $\textsc{Mul}_2$ operation. (For this example, we assume that the architecture only has three general-purpose registers: `r8`, `r9`, and `rdx`.) At this

```
Inductive REG := rax | rcx | (* ...65 omitted... *) | r15b.
Inductive AccessSize := byte | word | dword | qword.
Record MEM := { mem_bits_access_size : option AccessSize; mem_base_reg : option REG;
        mem_scale_reg : option (Z * REG); mem_base_label : option string; mem_offset : option Z }.
Inductive FLAG := CF | PF | AF | ZF | SF | OF.
Inductive OpPrefix := rep | repz | repnz.
Inductive OpCode := adc | adcx | add | adox | and | bzhi | call | clc | cmovb | cmovc | cmovnz | cmp
  | db | dd | dec | dq | dw | imul | inc | je | jmp | lea | mov | movzx | mul | mulx | pop | push
  | rcr | ret | sar | sbb | setc | seto | shl | shlx | shr | shrx | shrd | sub | test | xchg | xor.
Record JUMP_LABEL := { jump_near : bool; label_name : string }.
Inductive ARG := reg (r : REG) | mem (m : MEM) | const (c : Z) | label (l : JUMP_LABEL).
Record NormalInstruction := { prefix : option OpPrefix; op : OpCode; args : list ARG }.
```

Fig. 9. Syntax of Coq embedding of x86-64 assembly

point, registers `r8`, `r9`, `rdx` have been used for t_2, t_1, and t_0, respectively. Hence, the need to spill a register. Observing that t_2 is not required until ADD_4, whereas t_0 and t_1 are used earlier, CryptOpt spills `r8` (line 10).

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, though mutations may later alter the choice.

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 technique commonly used by off-the-shelf compilers.

5.6 Objective-Function Evaluation

We compare different random program variants by running them on the actual processors of interest. Controlling noise in running-time measurement is of utmost importance because with too much noise, randomized search could be driven into unproductive oscillation. We found the details of such measurement surprisingly difficult to get right. The full version of this paper 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.

6 CHECKING PROGRAM EQUIVALENCE

Our goal with CryptOpt was to preserve or even strengthen the formal guarantees of Fiat Cryptography. One way to achieve that goal would have been to verify the whole randomized-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.

Figure 9 shows a nearly complete description of the x86-64 assembly syntax accepted by our checker. For better error messages, some control-flow opcodes like `jmp` are included, though they will always be rejected by the checker. This type of syntax trees is given a very standard semantics, in the form of an interpreter as a Coq function. The simplicity of syntax and semantics is important, since both are referenced by the final theorem for any specific compilation, while the syntax and semantics of Fiat IR drop out of the picture as untrusted.

6.1 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.

The public API connecting the checker's two main components 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.

6.2 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.

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.

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.

6.3 Equivalence

Algorithm 2 presents the `internalize` algorithm 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. Elided from the figure are additional rewrite rules to complement the one we include, which notices that an operation to extract the low byte of a constant is extraneous, when the constant is low enough. The algorithm ends in a linear scan of the E-graph for existing nodes matching the normalized input, which perhaps surprisingly turned out to be more than fast enough for our examples.

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
```

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.

Here it is interesting to note which standard features of SMT-based verification tools did *not* need to be implemented, saving us from needing to prove their soundness. We incorporated no SAT-style management of case splits, nor did we need to include specialized cooperating decision procedures for domains like arithmetic on mathematical integers or bitvectors. It sufficed to stick to congruence-closure-style reasoning in the theory of equality with uninterpreted functions, augmented with a modest pool of rewrite rules, as SMT solvers are also often effective at applying. Note also that we omitted a central complexity of E-graph implementations: merging nodes (usually through union-find algorithms)

as it is discovered that they are equal. Our domain is simple enough that relevant equalities are always discovered at node-creation time. As for symbolic execution, we avoided the characteristic complexities of control flow (e.g. merging logical states) and memory access (e.g. flexible-enough addressing that pointer aliasing is nontrivial to check).

6.4 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 takes as input a choice of cryptographic algorithm, its numeric parameters, and an assembly file, and it checks that the assembly file matches the behavior of the algorithm, by comparing it with the Fiat IR code (itself generated by a verified compiler).

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. Slightly more precisely, we depend on the following preconditions, several of which we formalize using notations borrowed from separation logic [Reynolds 2002].

- 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 laid 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.

7 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 7.2)?
(2) How does CryptOpt compare with traditional compilation (Section 7.3)?
(3) Is CryptOpt optimization platform-specific (Section 7.5)?
(4) How does CryptOpt-optimized code perform as part of a cryptographic implementation (Section 7.6)?

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

7.1 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 Server 22.04.1 LTS for all machines, with all packages up-to-date. Moreover, because performance counters are not available on the efficiency cores of the i9 12G and i9 13G, we only use the performance cores on those machines.

Table 1. Overview of target machines used in the experiments

Name	CPU	μ-architecture
1900X	AMD Ryzen Theadr. 1900X	Zen 1
5800X	AMD Ryzen 7 5800X	Zen 3
5950X	AMD Ryzen 9 5950X	Zen 3
7950X	AMD Ryzen 9 7950X	Zen 4
i7 6G	Intel Core i7-6770HQ	Skylake-H
i7 10G	Intel Core i7-10710U	Comet Lake-U
i9 10G	Intel Core i9-10900K	Comet Lake-S
i7 11G	Intel Core i7-11700KF	Rocket Lake-S
i9 12G	Intel Core i9-12900KF	Alder Lake-S
i9 13G	Intel Core i9-13900KF	Raptor Lake-S

Generation. Every platform has at least four physical cores. For a fair comparison, we run the same number of optimization processes in parallel on all platforms. We pin the optimization processes to cores to reduce noise due to context switching. We choose 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, of which we report only the best-performing. The number of assembly instructions varies from 51 to 1281, depending on the field, the operation, and how well the optimization went. See Table 2 for details.

Bet-and-Run. Each optimization process has a budget of 200 000 mutations. In the bet stage, we explore 20 initial candidate solutions, optimizing each for 1 000 mutations. Hence, overall, we use 20 000 mutations, which are 10% of the total budget, for the bet part. The remaining 180 000 mutations are used for the run stage of the Bet-and-Run strategy. With those parameters, the generation and verification of all of the 18 Fiat IR primitives takes between 20 and 40 wall-clock hours, depending on the machine.

Table 2. Instruction count (average over all eight platforms)

Primitive	Multiply	Square
Curve25519	173.300	123.333
NIST P-224	226.967	222.233
NIST P-256	206.133	198.600
NIST P-384	580.400	570.300
SIKEp434	968.333	927.833
Curve448	550.133	359.467
NIST P-521	575.233	359.233
Poly1305	73.333	55.633
secp256k1	228.333	223.500

Code Verification. When combined with Fiat Cryptography, the optimization process verifies that the code it produces is equivalent to the Fiat IR code. We perform verification only on the final output of the optimization. Table 3 shows the time it takes to verify one assembly implementation as an average over the ten platforms.

Table 3. Verification times (average over all ten platforms)

Primitive	Multiply	Square
Curve25519	0.29 *s*	0.18 *s*
NIST P-224	2.16 *s*	1.94 *s*
NIST P-256	1.07 *s*	0.95 *s*
NIST P-384	29.09 *s*	25.81 *s*
SIKEp434	315.23 *s*	284.32 *s*
Curve448	3.02 *s*	1.44 *s*
NIST P-521	3.14 *s*	1.46 *s*
Poly1305	0.07 *s*	0.06 *s*
secp256k1	1.78 *s*	1.65 *s*

Performance Metric. To compare the performance of different implementations, we need a stable metric. 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. For more details on this aspect of our experimental setup, see the full version of this paper.

7.2 Optimization Progress

The first evaluation question we answer is how the optimization progresses over time. We include details in the full version of this paper, but a summary is that optimization progress is roughly logarithmic in the number of mutations, with some interesting differences in measurement stability across platforms.

7.3 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 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 ten platforms summarized in Table 1 and select the best result as described in Section 7.1. Additionally, we compile the equivalent C code, as produced by Fiat Cryptography, with GCC 12.1.0 [GCC 2022] and Clang 15.0.6 [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 4. Geometric means of CryptOpt vs. off-the-shelf compilers.

	Multiply		Square	
Curve	Clang	GCC	Clang	GCC
Curve25519	1.25	1.16	1.18	1.17
P-224	1.54	2.52	1.40	2.56
P-256	1.70	2.61	1.63	2.59
P-384	1.45	2.49	1.37	2.51
SIKEp434	1.70	2.43	1.73	2.39
Curve448	1.19	0.98	1.07	1.05
P-521	1.30	0.97	1.35	1.03
Poly1305	1.12	1.22	1.11	1.26
secp256k1	1.80	2.62	1.71	2.54

Table 4 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 the full version of this paper for the 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 the large fields, as in P-521 and Curve448, CryptOpt is less successful, achieving modest gains for the square function compared to GCC and slightly underperforming for multiply compared to GCC. 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 511–602 instructions and for P-521-mul 509–648 instructions depending on the platform. For comparison, Curve25519-mul is in the order of 160–195 instructions and in the range of 66–83 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.

We further evaluated the impact of profile-guided optimization (PGO) on the performance of code produced by the mainstream compilers. Specifically, we compiled the methods adding the profiling option (`-fprofile-generate` in GCC and `-fprofile-instr-generate` in Clang). We then ran each function in a tight loop for 10 000 iterations with random input, generating profile traces. Lastly, to use the profile traces, we added the `-fprofile-use` switch (including one call to `llvm-profdata merge` to create the correct format and the switch `-fprofile-instr-use` when

using Clang) to our compilation options. We then measured the performance of the code generated from this last compilation.

Using PGO with Clang only changes the position of the code for the Curve25519 and P-521 fields and only results in negligible performance changes. Using PGO with GCC improves the mean performance by ~2% over the whole set of functions. Notably, PGO improves the performance of the SIKEp434 operations by ~15%. In all tested functions, CryptOpt significantly outperforms GCC even when used with PGO.

7.4 CryptOpt vs. Superoptimization

To compare CryptOpt to superoptimizers, we use STOKE [Schkufza et al. 2013]. STOKE supports two operation modes: synthesize, where it aims to generate new code that performs the function; and optimize, where it attempts to modify a function to find a faster alternative. Synthesize mode failed to generate correct code from scratch even though we let it run for more than three days. This is expected because in synthesize mode, STOKE aims for small kernels, whereas functions for finite-field arithmetic are on the order of hundreds of instructions.

For the optimize mode, we compile our test functions with Clang 3.4 and GCC 4.9. (STOKE does not support newer versions of these compilers.) We then try to optimize the assembly code that the compilers emit.

In four of the prime fields (Curve25519, Poly1305, P-448 and P-521), the code produced by the compilers uses the `shrd` instruction, which is not supported in STOKE due to the potential for undefined behavior. For the remaining fields, in most optimization attempts, STOKE either times out or emits assembly code that contains either syntactical errors that prevent it from being assembled or logical errors that result in incorrect code.

We only managed to get results for the square function of secp256k1, when compiled with GCC 4.9. When optimizing this function, STOKE produces code that is about 6% faster than the output of compilation with GCC 11. However, the code is still 73% slower than the code that CryptOpt produces for the same function.

7.5 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. Surprisingly, sometimes it does pay off to optimize on a different platform than the target, apparently when the host supports more stable performance measurement. See the full version of this paper for details.

7.6 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-curve cryptography. Specifically, we investigate implementations of two popular elliptic curves: Curve25519 and secp256k1. We compare the performance of 15 implementations of these curves, four of which use CryptOpt code for field operations. In Table 5, 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] provides three implementations: a portable C implementation that we identify as O'SSL, and two assembly-based implementations, based on amd64-51 and amd64-64, which we identify as O'SSL fe-51 and O'SSL fe-64, respectively. At runtime, OpenSSL chooses which implementation to use, opting by default

for O'SSL fe-64. Additionally, Project Everest [HACL 2022] provides an assembly implementation in which the computations of two field operations are interleaved to achieve better utilization of the CPU pipeline.

For secp256k1, we use two implementations from the libsecp256k1 library [Bitcoin Core 2021], one hand-optimized assembly and the other portable C.

CryptOpt-Based Implementations. For the comparison, we use four implementations with CryptOpt-optimized code. Specifically, for Curve25519, we replace the field operations in O'SSL-fe51 and in O'SSL-fe64 with CryptOpt-optimized field operations. We call these implementations O'SSL fe-51+CryptOpt and O'SSL fe-64+CryptOpt, respectively.

For secp256k1 we use two implementations. The first, libsecp256k1+CryptOpt (Fiat), uses the scalar multiplication code from `libsecp256k1` with the field operations as produced by Fiat Cryptography and optimized with CryptOpt.

Additionally, to demonstrate the use of the CryptOpt optimizer as a stand-alone tool, we use Clang to compile the field operations of the portable implementation of `libsecp256k1` into LLVM IR, which we convert to the input format of the CryptOpt optimizer. We then use the latter to optimize the code, replacing the field operations with the optimized code. This implementation is called libsecp256k1+CryptOpt (Opt).

Table 5. Performance of Scalar Multiplication (Geometric Mean).

Curve25519		
Implementation	Lang	Cycles
sandy2x [Chou 2015]	asm-v	486k
amd64-64 [Chen et al. 2014]	asm	542k
amd64-51 [Chen et al. 2014]	asm	546k
donna [Langley 2022]	asm-v	972k
donna-c64 [Langley 2022]	C	577k
O'SSL [OpenSSL 2022]	C	530k
O'SSL fe-51 [OpenSSL 2022]	asm	530k
O'SSL fe-51+**CryptOpt**	asm	524k
O'SSL fe-64 [OpenSSL 2022]	asm	455k
O'SSL fe-64+**CryptOpt**	asm	461k
HACL* fe-64 [HACL 2022]	asm	452k
secp256k1		
Implementation	Lang	Cycles
libsecp256k1 [Bitcoin Core 2021]	asm	547k
libsecp256k1 [Bitcoin Core 2021]	C	530k
libsecp256k1+**CryptOpt** (Fiat)	asm	527k
libsecp256k1+**CryptOpt** (Opt)	asm	528k

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 (for two base-point multiplications and two variable-point multiplications) of the fastest compiler setting. We evaluate each implementation on the ten hardware platforms (c.f. Table 1) and report the geometric mean (rounded to the nearest 1000 cycles) in Table 5. (See the full version of this paper for the full details.)

Results. Comparing CryptOpt with the similarly structured hand-optimized implementations of OpenSSL fe-51 and fe-64, we find that the performance with and without CryptOpt is similar. On average, CryptOpt generates slightly faster implementations for fe-51 and slightly slower implementations for fe-64. Manual optimization of code requires significant expertise and a large time investment, which needs to be repeated for each finite field. In contrast, using CryptOpt is fairly straightforward and only requires moderate computing resources to achieve similar results. CryptOpt also underperforms highly optimized implementations that use a different API (HACL*), which we do not support yet.

For secp256k1, the libsecp256k1+CryptOpt (Fiat) implementation beats the performance of the hand-tuned assembly, slightly outperforms the C compiled code, and provides verified formal correctness. The libsecp256k1+CryptOpt (Opt) implementation achieves higher performance than both the state-of-the-art and our verified implementation albeit slightly slower than the version based on Fiat Cryptography.

We leave the tasks of verifying the implementation for secp256k1 and emitting field operations similar to the HACL* API to future work.

CryptOpt on New Hardware. When looking at the results on specific machines (more details in the full version of this paper), we see that CryptOpt excels on the i7 11G, i9 12G, and i9 13G platforms, providing the overall fastest implementations for fe-64-based field operations. On these platforms, 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 12^{th} 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.

7.7 Artifacts

The artifact, on which the evaluation was done, is available at: https://zenodo.org/record/7710435, with the DOI `10.5281/zenodo.7710435`. The artifact includes instructions to reproduce the claimed results in this paper. As of April 2023, the most up-to-date version of Fiat Cryptography can be found in their repository at https://github.com/mit-plv/fiat-crypto, and up-to-date versions of CryptOpt at https://github.com/0xADE1A1DE/CryptOpt.

8 CONCLUSION

We presented CryptOpt, a tool that brings a perhaps-surprising confluence of improving performance and increasing formal assurance. It tackles the distinctive simplifications and complexities of straight-line cryptographic code. We showed empirically that certain simplifications to established techniques suffice to set new performance records for important routines on some relevant platforms. In *generation* of fast code, we developed a simple set of transformation operators that make genetic search effective. In *checking* of fast code with foundational mechanized proofs, we followed SMT solvers and symbolic-execution engines, while avoiding their most complex aspects, like arithmetic decision procedures or nontrivial pointer-aliasing checks. We hope that these techniques can be generalized to other domains of compilation.

ACKNOWLEDGMENTS

Input from many anonymous reviewers has helped shaping this paper. We are grateful to all for their work and valuable comments. In particular, we thank our shepherd, Yaniv David, for the careful reading, guidance, and support.

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Australian Research Council projects DE200101577, DP200102364 and DP210102670; the Blavatnik ICRC at Tel-Aviv University; CSIRO's Data61; the Deutsche Forschungsgemeinschaft (DFG, Germany's Excellence Strategy) under Germany's Excellence Strategy EXC 2092 CASA - 390781972; 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.

Part of this work was carried out while Chitchanok Chuengsatiansup, Markus Wagner, and Yuval Yarom were affiliated with the University of Adelaide, and while Chuyue Sun was affiliated with the Massachusetts Institute of Technology.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. 2, 8

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida 10

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. 5

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. 4

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 19

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *ASPLOS*. 394–403. 8

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

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. 9

Seth D. Bergmann. 2003. Compilers. In *Encyclopedia of Information Systems*. 141–170. 8

Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *FSE*. 32–49. 19

Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC*. 207–228. 19

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-20221005.tar.xz 20, 21

Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017a. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *IEEE SP*. 483–502. 9

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017b. Everest: Towards a Verified and Drop-in Replacement of HTTPS. In *Proc. SNAPL*. https://project-everest.github.io/assets/snapl2017.pdf 9

Bitcoin Core. 2021. libsecp256k1 - Optimized C Library for ECDSA Signatures and Secret/Public Key Operations on Curve secp256k1. https://github.com/bitcoin-core/secp256k1/blob/9526874d1406a13193743c605ba64358d55a8785/src/field_5x52_int128_impl.h 3, 21

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. 6, 8, 9

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. 5

Certicom Research. 2000. SEC 2: Recommended elliptic curve domain parameters, version 1.0. http://www.secg.org/SEC2-Ver-1.0.pdf. 19

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. 7

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. 8, 20, 21

Tung Chou. 2015. Sandy2x: New Curve25519 Speed Records. In *SAC*. 145–160. 2, 20, 21

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 19

Keith D. Cooper and Linda Torczon. 2012. Chapter 11 - Instruction Selection. In *Engineering a Compiler (Second Edition)*. 597–638. 8

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340. 5

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (2005), 365–473. 3, 5

Benjamin Doerr and Frank Neumann. 2019. *Theory of evolutionary computation: Recent developments in discrete optimization*. Springer. 4

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, 8, 19

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP*. 125–128. 8

Matteo Fischetti and Michele Monaci. 2014. Exploiting Erraticism in Search. *Operations Research* 62, 1 (2014), 114–122. 4

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. 9

Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Signed Cryptographic Program Verification with Typed CryptoLine. In *CCS*. ACM, 1591–1606. 8

GCC. 2022. GCC, the GNU Compiler Collection. https://gcc.gnu.org 19

Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. 10

HACL. 2022. HACL. https://github.com/hacl-star/hacl-star 21

Mike Hamburg. 2015. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive* 2015 (2015), 625. 19

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. 6

Mark Harman and Bryan F. Jones. 2001. Software engineering using metaheuristic innovative algorithms: workshop report. *Inf. Softw. Technol.* 43, 14 (2001), 905–907. 6

Rodney E. Hooker and Collin Eddy. 2013. Store-to-load forwarding based on load/store address computation source information comparisons. US Patent 8533438. 5

Rajeev Joshi, Greg Nelson, and Keith H. Randall. 2002. Denali: A Goal-directed Superoptimizer. In *PLDI*. ACM, 304–314. 3, 7

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. 6

Adam Langley. 2022. Curve25519-donna. https://github.com/agl/curve25519-donna 20, 21

Kevin M. Lepak and Mikko H. Lipasti. 2000. On the value locality of store instructions. In *ISCA*. 182–191. 5

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. 5

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert — A Formally Verified Optimizing Compiler. In *ERTS*. 5

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. 65–79. 5, 14

Henry Massalin. 1987. Superoptimizer - A Look at the Smallest Program. In *ASPLOS*. ACM Press, 122–126. 7

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

OpenSSL. 2022. OpenSSL. https://www.openssl.org/ 4, 20, 21

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. 7

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. 8

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. 6

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS*. 151–166. 8

Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2018. Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk). In *CONCUR (LIPIcs, Vol. 118)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik,

4:1–4:16. 8

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. 9

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. 9

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 8

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. 5

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). 7

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*. ACM, 305–316. 3, 7, 20

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*. ACM, 53–64. 7

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. 8

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*. 471–482. 8

Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *OOPSLA*. ACM, 391–406. 7

Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. In *OOPSLA*. ACM, 147–162. 7

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. 7

Samantika Subramaniam and Gabriel H. Loh. 2006. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *MICRO*. 273–284. 5

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*. 17–27. 5

Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *CCS*. ACM, 1973–1987. 8

Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: learning replacement policies from hardware caches. In *PLDI*. 519–532. 5

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. 4

Received 2022-11-10; accepted 2023-03-31