Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, Tiago Oliveira

Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification

José Bacelar Almeida[1], Manuel Barbosa[2], Gilles Barthe[3], Vincent Laporte[4], and Tiago Oliveira[2]

[1] INESC TEC and Universidade do Minho, Portugal
[2] INESC TEC and FCUP, Universidade do Porto, Portugal
[3] Max Planck Institute for Security and Privacy; IMDEA Software Institute, Spain
[4] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract. We present a new tool for the generation and verification of high-assurance high-speed machine-level cryptography implementations: a certified C compiler supporting instruction extensions to the x86.
We demonstrate the practical applicability of our tool by incorporating it into SUPERCOP: a toolkit for measuring the performance of cryptographic software, which includes over 2000 different implementations. We show i. that the coverage of x86 implementations in SUPERCOP increases significantly due to the added support of instruction extensions via intrinsics and ii. that the obtained verifiably correct implementations are much closer in performance to unverified ones.
We extend our compiler with a specialized type system that acts at pre-assembly level; this is the first constant-time verifier that can deal with extended instruction sets. We confirm that, by using instruction extensions, the performance penalty for verifiably constant-time code can be greatly reduced.

Keywords: certified compiler, SIMD, SUPERCOP, constant-time.

1 Introduction

A key challenge in building secure software systems is to develop good implementations of cryptographic primitives like encryption schemes, hash functions, and signatures. Because these primitives are so-highly relied on, it is important that their implementations achieve maximal efficiency, functional correctness, and protection against side-channel attacks. Unfortunately, it is difficult to achieve these goals: severe security vulnerabilities are exposed in many implementations despite heavy scrutiny [2,1,40]. Computer-aided cryptography is an area of research that aims to address the challenges in creating high-assurance cryptography designs and implementations, by creating domain-specific formal verification techniques and tools [10]. This work extends the range of tools that can be used for computer-aided cryptography.

There are many paths to achieve high-assurance cryptography for assembly-level implementations. The most direct path is naturally to prove functional

correctness and side-channel protection directly at assembly level. This is the approach taken in [21] for proving functional correctness of Curve25519 and more recently in in [18,26] for proving functional correctness of several implementations. However, verifying assembly programs is less intuitive, for instance due to the unstructured control flow, and there choice of verification tools is more limited. An alternative, and popular, approach is to verify source-level programs. This approach is taken for instance in [41], where Zinzindohoue et al develop functionally verified implementations of popular cryptographic algorithms. The benefits are two-fold: there is a broader choice of verification tools. Second, the verification can benefit from the high-level abstractions enforced at source level. However, this approach introduces the question of *trust* in the compiler.

Indeed, formal guarantees can be undermined by incorrect compilers. For instance, bugs in optimization passes can generate functionally-incorrect assembly code from functionally-correct source code [36,29,39]. Worse yet, since compiler optimizations do not generally even purport to preserve security [23], compilers can generate assembly code vulnerable to side-channel attacks from carefully scrutinized source code. The prevailing approach to eliminate trust in the compiler is to use certified compilation, see e.g. [30]. Informally, a certified compiler is a compiler in which each compiler pass is augmented with a formal proof asserting that it preserves the behavior of programs. This entails that functionally-correct source code is compiled into functionally-correct assembly.

While certified compilation is a promising approach to guarantee functional correctness, it entails key limitations described next. These limitations are particularly-relevant for cryptographic code, where sub-optimal execution time can be a deal-breaker for code deployment. First, the optimizations supported by currently-available certified compilers do not produce code competitive with that produced by the most-aggressively-optimizing (non-certifying) compilers. Indeed, the certification requirement limits the set of optimizations that can currently be performed. For example, global value numbering is an optimization that is missing from most compilers; see however [13]. The preeminent certified compiler, CompCert [30], generates code whose performance typically lies between GCC's -O1 and -O2 optimization levels. While this is an impressive achievement, and arguably a small price to pay for provably-correct optimizations, the efficiency requirements on implementations of cryptographic primitives may necessitate more aggressive optimizations. Second, the current certified compilers do not apply to the most efficient implementations. This is primarily due to two factors: the presence of special compiler intrinsics and inline assembly code, and the incompleteness of automatic assembly-level verification. For example, many implementations invoke compiler intrinsics corresponding to SSE bit-wise and word-shuffling instructions. Therefore, a last line of work is to develop certified compilers for high speed cryptography.

Contributions. In this paper we further develop the route of obtaining competitive high-assurance cryptography implementations via general-purpose certified compilers. We present a new version of CompCert that significantly narrows the measured performance gap *and* implements a static analysis capable of verifying

constant-time for CompCert-generated pre-assembly code. This new certified compiler comes with the ability to handle the special compiler intrinsics used in implementations of cryptographic implementations.

To generate performance measurements, we consider the SUPERCOP benchmarking framework, which includes an extensive repository of implementations of cryptographic primitives. We pose that the performance penalty incurred by formal verification should be measured with respect to the fastest known implementations, even if these are hand-crafted directly at the assembly level and validated using heuristic means. Thus for each cryptographic primitive P we consider the *performance penalty* of a given compiler C to be the ratio of the performance of the best-performing implementation of P which is actually able to be compiled with C against by the best-performing implementation of P compiled with *any* compiler. In particular these ratios can grow either because C produces less-optimal code than some other compiler, or because C cannot process a given implementation, e.g., due to special compiler intrinsics or inlined assembly. Our findings are that the average performance penalties due to certified compilation lie between factors of 16 and 21, depending on the version of CompCert. For several primitives, we observe penalties in the range between two and three *orders of magnitude*, although the majority of implementations compensates for these degenerate cases.

Next, we turn to side-channel protection. The gold standard for cryptographic implementations is the *constant-time* property, whereby programs' memory access behaviors are independent from secret data—for both code and data memory. In other words, for fixed public data such as input length, the memory locations accessed by loads and stores, including instruction loads, is fixed. The constant-time property is a countermeasure against timing-attacks, whereby attackers learn secrets by measuring execution time variations, e.g. due to cache behavior. While this verification traditionally amounted to manual scrutiny of the generated assembly code by cryptographic engineers, recent work has shown progress in automatic verification to alleviate this burden [38,7,3,6,20,22].

The prevailing trend in this line of work is to carry verification of side-channel protection and this is typically performed towards the end of the compilation chain or directly on the generated assembly code. This is because even certified compilers may not preserve countermeasures against side-channel attacks — a notable exception being the CompCertCT compiler that has been formally proved to preserve these countermeasures [12]. Following this trend, our version of CompCert includes an intrinsics-aware constant-time verifier, following the type-checking approach at Mach level of [11]. The reason for focusing on the Mach intermediate language is that, although it is very close to assembly, is more suitable for analysis. The type system is described as a data-flow analysis, which keeps track of secret-dependent data and rejects programs that potentially use this data to violate the constant-time property. Because our type system is able to check programs that rely on instruction extensions, we are able to compile C code into functionally correct and verifiably constant-time implementations offering unprecedented efficiency. As an example, we can verify an implementation of AEZ

relying on AES-NI that executes 100 times faster than the fastest implementation that could be compiled with the original version of CompCert.

2 Related Work

Our work follows prior work in verified compilers and computer-aided cryptography. We refer the interested reader to [10] for an extensive recent review of the state of the art in computer-aided cryptography, namely the different techniques and tools for functional correctness and constant-time verification. We focus here on closely related works on (secure) verified compilation to cryptography.

The earlier applications of verified compilers to cryptographic implementations were inspired by CompCert [30], a moderately optimizing verified compiler for a significant fragment of C. Almeida et al [4] leverage the correctness theorem of CompCert to derive both functional correctness *and* side-channel resistance (in the program counter model [32]) for an implementation of RSA-OAEP encryption scheme as standardized in PKCS#1 v2.1. In a subsequent, related work, Barthe et al [11] build a verified static analysis at pre-assembly level for cryptographic constant-time, ensuring that programs which pass the analysis do not branch on secrets and do not perform secret-dependent memory accesses, guaranteeing that such implementations are protected (to some extent) against cache-based side-channel attacks; moreover, they use their static analysis to validate several implementations from the NaCl library. Our work builds on this development.

More recently, Almeida et al [5] propose a general methodology for carrying provable security of algorithmic descriptions of cryptographic constructions and functional correctness of source-level implementations to assembly-level implementations, and for proving that assembly-level implementations are constant-time; moreover, they apply their methodology to an implementation of MEE-CBC. In parallel, Appel et al [8,9] have developed general-purpose program logics to reason about functional properties of source-level programs, and applied these program logics to prove functional correctness of a realistic SHA-256 implementation; in a follow-up work, Beringer et al [15] combine the Foundational Cryptography Framework of [33] to build a machine-checked proof of the (elementary) reductionist argument for HMAC.

Fiat-Crypto [24] is a recently proposed framework for the development of high-speed functionally correct implementations of arithmetic libraries for cryptography. Certified compilation is used to convert a high-level algebraic specification of the library functionality into C code that can subsequently be compiled into executable code. Our approach is complementary to Fiat-Crypto, in that our verified compiler can be used to compile the generated C code, carrying the functional correctness and constant-time guarantees to low level code, including code that relies on intrinsics.

Hacl* [41] is a library of formally verified, high-speed implementations. Hacl* is included in recent versions of Mozilla Firefox's NSS security engine. It has recently been extended to vectorized implementations [34]. F* programs from Hacl* library can be compiled into C code using KreMLin [35] and then compiled into assembly,

for instance, using CompCert. Kremlin is a high-assurance compiler/extractor tool that generates provably correct C code from F* specifications. Additionally KreMLin is the tool used to generate the TLS1.3 code verified in Project Everest.[5] Again, our verified compiler can be used to compile C code relying on intrinsics generated by KreMLin with the guarantee that functional correctness is preserved.

Finally, we mention work on Jasmin [3,6], a new pre-assembly language for the implementation of high-assurance crypto code. Jasmin comes with a verified compiler that is guaranteed to preserve, not only functional correctness but also source-level constant-time properties [14,12] The Jasmin language also supports intrinsics and has been shown to give rise to competitive assembly implementations of cryptographic libraries. Our work provides an alternative to Jasmin, and indeed direct assembly verification using e.g., Vale [18], when functionally correct implementations are generated at C level. Very recently, Fromherz et al. [25] demonstrated the feasibility of formally verifying C code with inlined assembly. These lines of work are complementary to our own.

3 Background on x86 Instruction Extensions

x86 is a family of instruction sets that dates back to the Intel 8086/8088 processors launched of the late 70s. Throughout the years, successor Intel processors (and compatible models from other manufacturers, namely AMD) evolved from 16-bit to 32-bit architectures. The x86 designation is typically used to refer to instruction sets that are backward compatible with this family of processors. In this paper we will use x86 to (loosely) refer to the set of instructions that is supported transparently by most compilers that claim to support x86-compatible 32-bit architectures. We will use amd64 to refer to x86 extensions for 64-bit architectures, which we do not address specifically in this paper (although we present some data that permits evaluating what is lost by imposing this restriction).

In addition to the core x86 instructions, some architectures support additional domain-specific instruction sets, so-called instruction extensions. We will describe the instruction extensions introduced by Intel in the x86 architecture, since these are the ones that we target in this work. Intel introduced MMX in 1997, which included eight 64-bit registers (called MMX0-MMX7) in 32-bit machines and allowed for single instruction, multiple data (SIMD) operations over these registers for integer values of sizes 8 to 64. A SIMD instruction permits computing the same operation simultaneously over several values, e.g., by seeing a 64-bit register as two 32-bit values. In MMX the new 64-bit registers were overlapped with floating-point registers, which was a limitation.

The Streaming SIMD Extensions (SSE) introduced in 1999 removed this limitation by introducing eight 128-bit registers (XMM0-XMM7) that could be used as four single-precision floating point registers in the SIMD style. The SSE2 improvement introduced in 2001 provided a better alternative to the original MMX extension[6] by allowing the XMM registers to be also used to process integer data

[5] https://project-everest.github.io

[6] In some cases relying on MMX in parallel to SSE can give a performance advantages.

of various sizes. For this reason we do not consider the original MMX extensions in our tools. Subsequent SSE3 and SSE4 extensions increased the number of operations that can be performed over the XMM registers.

More recently, Intel launched the Advanced Vector Extensions (AVX and AVX2) that introduced 256-bit registers and instructions to allow SIMD computations over these registers. Our tools do not yet provide support for the AVX extensions. Another important class of instruction extensions are those associated with cryptographic computations. Intel added support for hardware-assisted AES computations in 2011 (AES-NI), and announced extensions for the SHA hash function family in 2013.[7]

INTRINSICS. Instruction extensions are usually domain-specific, and they provide relatively complex operations that should be directly available to the programmer, even if the programmer is using a high-level language such as C. For this reason, compilers typically provide a special mechanism to allow a programmer to specifically request the usage of an extended instruction; this mechanism is typically called an *intrinsic*. Intrinsics in C compilers such as GCC and CLANG are simply special function names and data types that are handled by the compiler in a different way to normal function declarations/definitions; for the most part, usage of these special functions is passed transparently through various compiler passes, and eventually translated into a single assembly instruction.

4 Adding x86 Instruction Extensions to CompCert

Our extension to CompCert was adapted from the 2.2 distribution of CompCert,[8] and it focuses only on the part of the distribution that targets the ia32 architecture. There is no particular reason for our choice of CompCert version, except that this was the most recent release when our project started. Equivalent enhancements can be made to more recent versions of CompCert with some additional development effort.

RELEVANT COMPCERT FEATURES. The architecture of CompCert is depicted in Figure 1. We follow [30] in this description. The source language of CompCert is called CompCertC, which covers most of the ISO C 99 standard and some features of ISO C 2011 such as the _Alignof and _Alignas attributes. Some features of C not directly supported in CompCertC v2.2, such as struct-returning functions, are supported via rewriting from the C source during parsing. The semantics of CompCertC is formalized in Coq and it makes precise many behaviors unspecified or undefined in the C standard, whilst assigning other undefined behaviours as "bad". Memory is modeled as a collection of byte-addressable disjoint blocks, which permits formalizing in detail pointer arithmetic and pointer casts.

CompCert gradually converts the CompCertC input down to assembly going through several intermediate languages. Parts of CompCert are not implemented directly in Coq. These include the non-certified translator from C to CompCertC

[7] We did not have access to a machine running SHA instruction extensions.
[8] http://compcert.inria.fr/

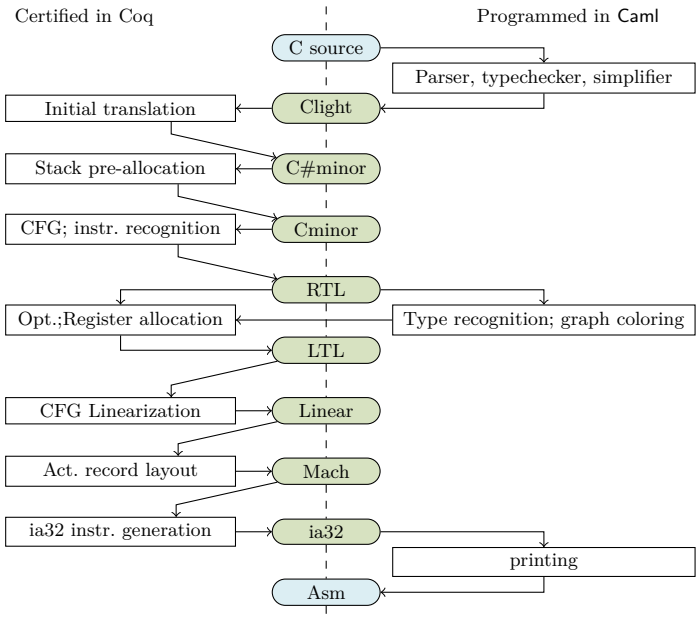

Fig. 1. CompCert architecture.

and the pretty-printer of the assembly output file. Additionally, some internal transformations of the compiler, notably register allocation, are implemented outside of Coq, but then subject to a *translation validation* step that guarantees that the transformation preserves the program semantics.

The front-end of the compiler comprises the translations down to Cminor: this is a typeless version of C, where side-effects have been removed from expression-evaluation; local variables are independent from the memory model; memory loads/stores and address computations are made explicit; and a simplified control structure (e.g. a single infinite loop construct with explicit block exit statements).

The backend starts by converting the Cminor program into one that uses processor specific instructions, when these are recognized as beneficial, and then converted into a standard Register Transfer Language (RTL) format, where the control-flow is represented using a Control Flow Graph (CFG): each node contains a machine-level instruction operating over pseudo-registers. Optimizations including constant propagation and common sub-expression elimination are then carried out in the RTL format, before the register allocation phase that produces what is called a LTL program: here pseudo-registers are replaced with hardware registers or abstract stack locations. The transformation to Linear format linearizes the CFG, introducing labels and explicit branching. The remaining transformation steps comprise the Mach format that deals with the layout of stack frames in accordance to the function calling conventions, and the final Asm language modeling the target assembly language.

The generation of the executable file is not included in the certified portion of CompCert – instead, the Asm abstract syntax is pretty-printed and the resulting programs is assembled/linked using standard system tools.

SEMANTIC PRESERVATION.CompCert is proven to ensure the classical notion of correctness for compilers known as *semantic preservation*. Intuitively, this property guarantees that, for any given source program S, the compiler will produce a compiled program T that operates *consistently* with the semantics of S. Consistency is defined based on a notion of *observable behaviour* of a program, which captures the interaction between the program's execution and the environment. Let us denote the evaluation of a program P over inputs $\vec{p}$, resulting in outputs $\vec{o}$ and observable behaviour B as $P(\vec{p}) \Downarrow (\vec{o}, B)$. Then, semantic preservation can thus be written as

$$\forall B, \vec{p}, \vec{o},\ T(\vec{p}) \Downarrow (\vec{o}, B) \quad \Longrightarrow \quad S(\vec{p}) \Downarrow (\vec{o}, B)$$

meaning that any observable behaviour of the target program is an admissible observable behaviour of the source program. Observable behaviours in CompCert are possibly infinite sequence of events that model interactions of the program with the outside world, such as accesses to volatile variables, calls to system libraries, or user defined events (so called annotations).

HIGH-LEVEL VIEW OF OUR COMPCERT EXTENSION. Our extension to CompCert is consistent with the typical treatment of instruction extensions in widely used compilers such as GCC: instruction extensions appear as *intrinsics*, i.e., specially named functions at source level. Calls to intrinsics are preserved during the first stages of the compilation, and eventually they are mapped into (typically) one assembly instruction at the end of the compilation. Intrinsic-specific knowledge is added to the compiler infrastructure only when this is strictly necessary, e.g., to deal with special register allocation restrictions; so transformations and optimizations treat intrinsic calls as black-box operations.

We have extended CompCert with generic intrinsics configuration files. Our current configuration was automatically generated from the GCC documentation[9] and the machine-readable x86 assembly documentation from x86asm.net.[10] This configuration file allows the CompCert parser to recognize GCC-like intrinsics as a new class of built-in functions that were added to the CompCert semantics. For this, we needed to extend the core libraries of CompCert with a new integer type corresponding to 128-bit integers; in turn this implies introducing matching changes to the various intermediate languages and compiler passes to deal with 128-bit registers and memory operations (e.g., a new set of alignment constraints; calling conventions; etc.). The new built-ins associated with intrinsics are similar to other CompCert builtins, apart from the fact that they will be recognized by their name, and they may carry immediate arguments (i.e., constant arguments that must be known at compile-time, and are mapped directly to the generated assembly code). These extended built-in operations are propagated down to assembly level, and are replaced with the corresponding assembly instructions at the pretty-printing pass. All changes were made so as to be, as much as possible, intrinsics-agnostic, which means that new instruction extensions can

[9] http://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html
[10] http://ref.x86asm.net

be added simply by modifying the configuration file. Overall, the development modified/added approx. 6.3k lines of Coq and ML, spread among 87 files from the CompCert distribution. We now present our modifications to CompCert in more detail.

Modifications to the CompCert front-end. Modifications at the compiler front-end are generally dedicated to making sure that the use of intrinsics in the source file are recognized and adequately mapped into the CompCertC abstract syntax tree, and that they are subsequently propagated down to the Cminor level. This includes modifications and extension to the C parser to recognize the gcc-style syntax extensions for simd vector types (e.g., the vector_size attribute), as well as adapted versions of intrinsics header files giving a reasonable support for source-level compatibility between both compilers. These header files trigger the generation of the added builtins, whose specification is included on the configuration file. For each new builtin, the following data is specified:

- the function identifier that is used to recognized the intrinsic by name;
- the signature of the intrinsic, i.e., the types of the input parameters and return type;
- an instruction class identifier that is used to group different intrinsics into different sets that can be activated/deactivated for recognition in different platforms (this is linked to a set of command-line option switches);
- the assembly instruction(s) that should be used when pretty-printing an assembly file in which that particular built-in operation appears;
- a Boolean value indicating whether the associated assembly instruction is two-address, which is relevant for register allocation later on.

Translation into CompCertC maps all vector types/values into a new 128-bit scalar type. Subsequent transformations were extended to support this data type.

Modifications to the CompCert backend. The most intrusive modifications to CompCert were done at the back-end level, most prominently in the register allocation stage. CompCert uses a non-verified graph-coloring algorithm to compute a candidate register allocation, whose output is then checked within Coq for correctness. We added the eight 128-bit xmm register-bank to the machine description, taking into account that floating point operations in CompCert were already using 64-bit half of these registers. This implied extending the notion of interference used during register allocation and adapting the corresponding proof of correctness. During the constructions of the stack-frame layout, the calling convention for vector parameters/return-values was implemented supporting up to 4 parameters and the return-value passed on registers. The final component of our extensions was the addition to the assembly pretty-printer, supporting a flexible specification of the code to be produced by each built-in.

Consequences for semantics preservation. Our new version of CompCert comes with an extended semantics preservation theorem that has essentially the same statement as the original one. The difference resides in the fact that the machine model now explicily allows built-in functions to manipulate 128-bit values. Note that, although we did *not* add a detailed formalization of the

semantics of all instruction extensions, this is not a limitation when it comes to the correctness of the compiler itself: indeed, our theorem says that, whatever semantics are associated by a machine architecture to a particular extended instruction, these will have precisely the same meaning at source level. This is a powerful statement, since it allows us to deal with arbitrary instruction extensions in a uniform way. Such detailed semantics would be important if one wished to reason about the meaning of a program at source level, e.g., to prove that it computes a particular function. In these cases a formal semantics can be given just for the relevant instructions.

5 Experimental Results

5.1 Coverage

SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) is a toolkit for measuring the performance of cryptographic software. In this work we looked at version 20170105 of the toolkit. SUPERCOP evaluates cryptographic primitives according to several criteria, the most prominent of which for the purpose of this work is execution time.

A SUPERCOP release contains all the machinery required to carry out performance evaluation in a specific computer; any computer can be used as a representative for a particular architecture and collected data can be submitted to a central repository of collected benchmarks. The implementations contained in a SUPERCOP release are organized in three hierarchical levels. At the top level reside so-called *operations* these correspond to the algorithms that constitute an abstract interface to a cryptographic component; for example, for digital signatures the operations comprise key generation, signature generation and signature verification. For each cryptographic component there can be many instantiations. These are called *primitives* in SUPERCOP. Different primitives will provide different tradeoffs; for examples some primitives may rely on standardised components such as AES, which may be a requirement for some applications, whereas other primitives may offer better performance by relying on custom-designed stream ciphers. Finally, for each primitive there may be many implementations, e.g., targetting different architectures, with different side-channel countermeasures, or simply adopting different implementation strategies.

In total, the SUPERCOP release we considered contained 2153 such implementations for 593 primitives. In Table 1 we present a more detailed summary of these counts, focusing on some interesting categories for this work. In particular, we detail the following successive refinements of the original implementation set: i. the number of implementations that target the x86-32 architecture (x86), which we identified by excluding all implementations that explicitly indicate a different target architecture; ii. how many of the above implementations remain (x86-C) if we exclude those that are given (even partially) in languages other than C, such as assembly; and iii. how many of these use instruction extensions (x86-ext) such as those described in Section 3. Additionally, we give an implementation count that extends the x86 one by including also implementations that explicitly

Table 1. SUPERCOP implementation histogram.

operations	x86	x86-C	x86-ext	amd64
aead	644	548	118	660
auth	19	19	6	19
box	2	2	0	2
core	25	25	0	29
dh	123	17	11	185
encrypt	13	11	4	13
hash	550	464	144	664
hashblocks	16	15	5	21
onetimeauth	9	2	0	11
scalarmult	7	5	0	14
secretbox	2	2	0	2
sign	62	47	11	65
stream	168	95	31	228
verify	3	3	0	3
total count	1643	1255	330	1916

target 64-bit architectures (amd64); this gives an idea of how much coverage is lost by restricting attention to 32-bit architectures.

One can see that 330 implementations resorting to x86 instruction extensions can be found in SUPERCOP, corresponding to 168 primitives—out of a total of 576 primitives that come equipped with a x86 implementation. This set of primitives represents the universe over which the new formal verification tools that we put forth in this work will provide benefits over pre-existing tools. Before moving to this detailed analysis, we conclude this sub-section with a high-level view of the data we collected in SUPERCOP that permits comparing certified compilers to general-purpose compilers. This statistic is a byproduct of our work and we believe it may be of independent interest, as it gives us an indication of what the state-of-the-art in certified compilation implies for cryptography.

Table 2 gives coverage statistics, i.e., how many implementations each compiler was able to successfully convert into executable code accepted by SUPERCOP in the machine we used for benchmarking. This machine has the following characteristics: Intel Core i7-4600U processor, clocked at 2.1 GHz, with 8 Gb of RAM, running Ubuntu version 16.04. We note that SUPERCOP exhaustively tries many possible compilation strategies for each compiler in a given machine. The baseline here corresponds to implementations in the set tagged as x86-C in Table 1 that were successfully compiled with GCC version 5.4.0 or CLANG version 3.8.0. The apparent discrepancy (1020 versus 1643) to the number of possible x86 implementations indicated in Table 1 is justified by the fact that some implementations omit the target architecture and incompatibility with x86 is detected only at compile-time.[11] In the table, ccomp refers to CompCert and ccomp-ext refers to the CompCert extension we presented in Section 4.

[11] The degenerate red value in the table is caused by implementations that use macros to detect intrinsic support; ccomp-ext activates these macros, but then launches an error in a GCC-specific cast.

Table 2. SUPERCOP coverage statistics for various compilers.

architecture	x86-32				amd64	
operations	baseline	ccomp-2.2	ccomp-ext	ccomp-3.0	baseline	ccomp-3.0
aead	343	258	290	178	506	269
auth	16	10	10	8	19	10
box	2	2	2	2	2	2
core	21	25	25	25	29	25
dh	2	2	2	2	7	3
encrypt	4	5	**1**	5	5	6
hash	471	323	356	239	562	380
hashblocks	12	8	8	8	16	11
onetimeauth	5	5	5	6	7	7
scalarmult	6	6	6	6	13	9
secretbox	2	2	2	2	2	2
sign	12	0	0	2	18	3
stream	121	91	114	91	152	19
verify	3	3	3	3	3	3
total count	1020	740	**824**	577	1341	749

One important conclusion we draw from this table is that, at the moment, the version of CompCert we present in this paper has the highest coverage out of all certified compiler versions, due to its support for intrinsics. Nevertheless, we still do not have full coverage of all intrinsics, which justifies the coverage gap to the baseline. In particular, we do not support the `__m64` MMX type nor AVX operations, as mentioned in Section 3. Furthermore, we do not use any form of syntactic sugar to hide the use of intrinsics, e.g., allowing XOR operations (^) over 128-bit values, which is assumed by some implementations fine-tuned for specific compilers.

5.2 Methodology for performance evaluation

We will be measuring and comparing performance penalties incurred by using a particular compiler. These penalties originate in two types of limitations: i. the compiler does not cover the most efficient implementations, i.e., it simply does not compile them; or ii. intrinsic limitations in the optimization capabilities of the compiler. In particular, we will evaluate the trade-off between assurance and performance when compiling cryptographic code written in C for different versions of CompCert. Our metric will be based on average timing ratios with respect to a baseline measurement. In all cases, the timing ratio is always reported to the fastest implementation overall, often given in assembly, as compiled by a non-verified optimizing compiler in the best possible configuration selected by SUPERCOP. We now detail how we compute our metrics.

PERFORMANCE METRICS. We consider each SUPERCOP operation separately, so let us fix an arbitrary one called $o \in O$, where O is the set of all operations in SUPERCOP. Let C be the set of compilation tools activated in SUPERCOP and $P(o)$ a set of primitives that realize o. Denote $I(p)$ as the set of all implementations provided for primitive $p \in P(o)$. Let also t_C^p denote the fastest timing value

reported by SUPERCOP over all implementations $i \in I(p)$, for primitive $p \in P(o)$, when compiled with all of the compilers in C. Note that, if such a value t_C^p has been reported by SUPERCOP, then this means that at least one implementation $i \in I(p)$ was correctly compiled by at least one of the configured compilers in C. Furthermore, t_C^p corresponds to the target code that runs faster over all the implementations given for p, and over all compilation options that were exhaustively attempted over C.

To establish a baseline, we have configured SUPERCOP with GCC version 5.4.0 and CLANG version 3.8.0 and collected measurements for all primitives. Let us denote this set of compilers by C^*. We then independently configured and executed SUPERCOP with different singleton sets of compilers corresponding to different versions of CompCert. Let us designate these by $C_{2.2}$, $C_{3.0}$ and $C_{2.2\text{-ext}}$, where the last one corresponds to our extension to CompCert described in Section 4. Again we collected information for all primitives in SUPERCOP.

For a given operation $o \in O$ we assess a compiler configuration C by computing average ratio:

$$R_C^o = \frac{1}{|P|} \cdot \sum_{p \in P} \frac{t_C^p}{t_{C^*}^p} ,$$

where we impose that t_C^p and $t_{C^*}^p$ have both been reported by SUPERCOP, i.e., that at least one implementation in $I(p)$ was successfully compiled via C and one (possibly different) implementation in $I(p)$ was successfully compiled by C^*.

When we compare two compiler configurations C_1 and C_2 we simply compute independently $R_{C_1}^o$ and $R_{C_2}^o$. However, in this case we first filter out any primitives for which either C_1 or C_2 did not successfully compile any implementations. The same principle is applied when more than two compiler configurations are compared; hence, as we include more compiler versions, the number of primitives considered in the rations tends to decrease. In all tables we report the number of primitives $|P|$ considered in the reported ratios.

Finally, since we are evaluating the penalty for using certified compilers, we introduced an extra restriction on the set of selected primitives: we want to consider only the performance of implementations covered by the correctness theorems. Our approach was heuristic here: if SUPERCOP reports that the most efficient implementation compiled by a CompCert version (including our new one) includes assembly snippets, we treat this primitive as if no implementation was successfully compiled.

THE COST OF CERTIFIED COMPILATION. If one looks at the performance penalty per operation for CompCert version 2.2 and version 3.0, as detailed above, and take the average over all operations, the we obtain a factor of 3.34 and 2.58, respectively. Note that, in primitives such as AES-GCM, the timing ratio is huge and can reach 700-fold because baseline implementations use AES-NI, while CompCert is generating code for AES. Nevertheless, these findings are consistent with what is usually reported for other application domains, and it does show that CompCert version 3.0 has significantly reduced the performance penalty when compared to previous versions. Note, however, that this is at the cost of a

Table 3. Performance ratios aggregated by instantiated operation.

operation	$\|P\|$	ccomp-2.2	ccomp-3.0	ccomp-ext
aead.decrypt	120	24.78	18.75	5.23
aead.encrypt	120	28.04	20.85	5.32
auth	5	3.50	1.76	3.58
box.afternm	2	1.90	1.52	1.83
box.open	2	1.80	1.50	1.84
dh	2	5.59	4.67	5.81
dh.keypair	2	4.65	3.93	4.65
encrypt	6	3.09	2.68	3.23
encrypt.open	6	5.04	4.08	5.09
hash	25	7.55	6.27	3.51
scalarmult.base	2	5.29	4.29	5.16
scalarmult	2	5.72	4.66	5.79
secretbox	2	2.24	1.74	2.09
secretbox.open	2	2.09	1.64	2.03
sign	25	4.87	3.64	4.55
sign.open	25	3.73	2.87	3.55
stream	10	1.91	1.42	1.93
stream.xor	10	1.62	1.35	1.66
global	494	21.00	15.83	4.79

operation	$\|P\|$	ccomp-2.2	ccomp-ext
aead.decrypt	166	22.03	5.39
aead.encrypt	166	24.70	5.49
hash	32	8.13	5.01
global	639	20.00	5.08

reduction in coverage (cf. Table 2). More recent versions of CompCert that we have benchmarked using a different set-up confirm a gradual improvement in the optimization capabilities of the compiler.

5.3 Performance boost from certified instrinsics-aware compilation

In this section we measure the performance improvements achieved by our new version of CompCert supporting instruction extensions. Table 3 shows two views of the collected results: the top table compares three versions of CompCert, whereas the bottom table compares only the vanilla version of CompCert 2.2 with our extended version of it. In the bottom table we list only the lines where the set of considered primitives differs from the top table. The results speak for themselves: for operations where a significant number of primitives come equipped with an intrinsics-relying implementation, the performance penalty falls by a factor of 5 when comparing to CompCert 2.2, and a factor above 3 when comparing to CompCert 3.0.

In Table 3 we are including primitives for which no implementation relying on instruction extensions is given. In that case our new version of CompCert does not give an advantage, and so the performance gain is diluted. To give a better idea of the impact for primitives where instruction extensions are considered, we

Table 4. Performance ratios aggregated by instantiated operation, restricted to primitives including at least one implementation relying on instruction extensions.

operation	$\|P\|$	ccomp-2.2	ccomp-3.0	ccomp-ext
aead.decrypt	50	56.60	42.84	7.89
aead.encrypt	50	64.29	47.77	7.94
auth	2	4.23	3.88	4.06
hash	43	6.63	5.42	4.21
stream	3	1.43	1.05	1.34
stream.xor	3	1.31	1.19	1.25
global	201	48.08	36.25	6.92

operation	$\|P\|$	ccomp-2.2	ccomp-ext
aead.decrypt	60	55.10	7.51
aead.encrypt	60	62.18	7.56
hash	46	6.40	4.14
global	234	48.63	6.70

present in Table 4 the average ratios that result from restricting the analysis to primitives where instruction extensions are used. These results show that, as would be expected, intrinsics-based implementations allow a huge speed-up when compared to implementations in plain C. The most significant improvements are visible in the AEAD operations, where one important contributing factor is the enormous speed boost that comes with relying on an AES hardware implementation, rather than a software one.

6 An intrinsics-aware constant-time checker

We now address two limitations of existing approaches to verifying constant-time implementations. The first limitation is the lack of support for instruction extensions. The second limitation is that, if one is looking to use a certified compiler that is not guaranteed to preserve the constant-time property, then using a constant-time verifier at source level does not guarantee constant-time at the target level. We integrated a new constant-time verification tool into the extended version of CompCert that we introduced in Section 4 and it follows the type-checking approach at Mach level of [11]. The reason for focusing on the Mach intermediate language is that, although it is very close to assembly, is more suitable for analysis.

The checker operates in three steps. First a value analysis computes an over-approximation of the values of the pointers: this is key for the precision of the checker when the program to verify stores sensitive data into memory. Then, a type system infers what are the run-time values that may depend on sensitive data. Finally, the policy checker validates that neither the control-flow nor the memory accesses depend on sensitive data.

TYPE SYSTEM OVERVIEW. The type system assigns a *security level* at each program point and in each calling context to each register and memory location—collectively called *locations*. Here, a calling context is a stack of call sites. Security

levels are taken in the usual security lattice with two points High and Low. Locations are labeled High at a particular program point and calling context if they may hold a value that depends on secret data whenever execution reaches that point in that context. The ones that are labeled Low are guaranteed to always hold values derived from public data only.

The type system is described as a data-flow analysis. Typing rules describe how the type information evolves when executing a single instruction. For instance, for an arithmetic operation like x = y + z; the corresponding rule mandates that the security level of x, after the execution of this instruction, should be, at least, the least upper bound of the security levels of y and z. Finally, rules for instructions that manipulate the memory rely on an external points-to information to resolve the targets of pointers. As an example, the rule for instruction x = *p; states that the security level of x after the instruction is above the security levels of all memory cells that may be targeted by pointer p. Note that the type-system applies to whole programs, rather than to individual functions; therefore a typing derivation actually unfolds the call-graph and it cannot be used in the presence of recursion. The implementation of this type system relies on the generic implementation of Kildall's algorithm in CompCert [30].

Once a typing derivation is found for a function, we check that the inferred type is consistent with the constant-time policy. For instance, the type information before a branch if(b) ... else ... should be such that all locations involved in condition b have security level Low. Furthermore, the security level of all pointers that are used in memory accesses are required to be Low.

Program analysis required for type-checking. Our type-system relies on a general purpose value analysis that is targeted to the inference of points-to information. It builds a flow-sensitive and context-sensitive approximation of the targets of pointers. However, in low-level languages, the boundary between pointer arithmetic and other computations is blurred. We thus need an analysis that can precisely cope with bit-vector arithmetic so as to infer precise approximations of the pointer offsets. Our implementation builds on the ideas present in the Verasco static analyzer [28]. On one hand, we reuse one of its non-relational numerical abstract domains that is suitable for the analysis of pointer offsets [16]. On the other hand, we implemented a memory abstract domain similar to Verasco's [17]. The result of the analysis is computed by iterating the abstract semantics of the program until a fixed point is found. It uses Bourdoncle algorithm [19] to build, for each function, an iteration strategy; when encountering a function call, the called function is analyzed in the current state through a recursive call of the analyzer, effectively unfolding the call-graph, for maximal precision. Between the memory abstract domain and the iterator, we squeezed in a trace partitioning domain [37,31] that is dedicated to the full unrolling of array initialisation loops. This domain is driven by annotations in the source code: the programmer must indicate that the loop should be fully unrolled in order to take advantage of the added precision of this analysis.

Support for intrinsics. Handling the intrinsic instructions in the analyses needs special care. To keep the analyses general (i.e., not tied to a specific instruction set), the type-system relies on an external oracle that classifies every built-in call in one of the following categories: pure, memory load, and memory store. This oracle is trusted and is built on the configuration files described in Section 4. A call to a built-in is pure when it has no effect beyond explicitly writing to some registers. Moreover, the security level of the result is the least upper bound of the levels of its arguments. For instance the call `y = _mm_and_si128(x, mask);` which computes the bitwise logical AND of its 128bit arguments, is pure: its effect is limited to writing to the `y` variable. Also, the content of the `y` variable will be considered public only if the two arguments are public. The built-ins that belong to the memory load category are the ones which treat one of their arguments as a pointer and read memory through it. They need to be treated as a memory load in the constant-time analysis. For instance, the call `v = _mm_loadu_si128(p);` is classified as a load through pointer `p`. Therefore, to comply with the constant-time policy, the value of this pointer must have the `Low` security level. Finally, the built-ins in the memory store category are the ones that write to the memory, and must be treated as such in both analyses. For instance, the call `_mm_storeu_si128(p, x);` is classified as a memory store of the value of `x` to the address targeted by `p`.

Enhancements with respect to [11]. This work improves the checker for constant-time of Barthe et al. [11] in several ways. First the value analysis is much more precise than their alias type-system: not only it is inter-procedural and context-sensitive, but it also finely analyzes the pointer offsets, so that the type-system for constant-time can cope with memory blocks that hold high and low data at the same time (in disjoint regions of the block). In particular, this means that local variables that may hold sensitive data need not be promoted to global variables. Second, our checker is inter-procedural, therefore can analyze programs with functions without a complete inlining prior to the analysis. Finally, our analyses soundly and precisely handle compiler intrinsics.

An example. aez [27] is an authenticated encryption scheme that was designed with the use of hardware support for aes computations in mind. The implementations for aez included in supercop comprise both reference code written purely in C, and high-speed code relying on aes-ni extensions. Our experiments in running CompCert 2.2 and CompCert with intrinsics support over aez indicates that the ratio with respect to the non-verified baseline compilation is over 700 in the case of the former and drops to roughly 7 when intrinsics support is added.

We ran our new type-system over the aez implementation, and we found a constant-time violation, albeit a benevolent one. The code causing the violation is the following:

```
if (d && !is_zero(vandnot(loadu(pad+abytes), final0))) return -1;
```

This is part of the AEZCore implementation for the decryption operation: it checks whether the correct padding has been returned upon decryption and

immediately exits the function if the check fails. Strictly speaking this is a violation of the constant-time policy, as the inverted value depends on the secret key. However, this violation can be justified down to the fact that the result of the check will be made publicly available anyway. Rather than doing this, we modified the AEZ implementation so as to store the result of the check and return it only after all subsequent operations are carried out. The modified code was accepted by our type-checker. As a result, we obtain a verifiably correct and verifiably constant-time implementation of AEZ. The combined level of assurance and speed of this implementation is unprecedented and is only possible due to the guarantees provided by the tools presented in this paper.

7 Conclusions and upcoming developments

Our work initiates a systematic study of the coverage of formal methods tools for cryptographic implementations, and develops generalizations of the CompCert verified compiler and of a constant-time static analysis to accommodate intrinsics. The statistics are encouraging, but there is significant room for achieving further coverage.

The development is available at https://github.com/haslab/ccomp-simd. We are currently porting our work to version 3.7 of CompCert, which will allow us to benefit from numerous new features that have been added since. Most notably, support to 64 bit architectures (in particular amd64), which by itself widens the applicability of the tool, and opens the way to support intrinsics for new vector extensions such as AVX, AVX2 and AVX-512. Finally, we are also updating our benchmarking set-up to the most recent versions of SUPERCOP, GCC and CLANG. We do not expect the main conclusions to change, but the number of assessed implementations will grow significantly.

Acknowledgements. This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within the project PTDC/CCI-INF/31698/2017, and by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550 (REASSURE).

References

1. M. R. Albrecht and K. G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In M. Fischlin and J. Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643. Springer, 2016.
2. N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.

3. J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017.
4. J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS*, 2013.
5. J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 163–184. Springer, 2016.
6. J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 965–982. IEEE, 2020.
7. J. C. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, Aug. 2016. USENIX Association.
8. A. W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014.
9. A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, 2015.
10. M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy* (*SP*). IEEE, 2021. https://oaklandsok.github.io/papers/barbosa2021.pdf.
11. G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *ACM SIGSAC Conference on Computer and Communications Security, CCS'14*. ACM, 2014.
12. G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020.
13. G. Barthe, D. Demange, and D. Pichardie. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* (*TOPLAS*), 36(1):4, 2014.
14. G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 328–343. IEEE Computer Society, 2018.
15. L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of openssl HMAC. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 207–221. USENIX Association, 2015.
16. S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of the 20th Static Analysis Symposium* (*SAS*), Lecture Notes in Computer Science. Springer-Verlag, 2013.
17. S. Blazy, V. Laporte, and D. Pichardie. An abstract memory functor for verified C static analyzers. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming,* (*ICFP*), pages 325–337, 2016.

18. B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 917–934, USA, 2017. USENIX Association.
19. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.
20. S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. Fact: a DSL for timing-sensitive computation. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019.
21. Y. Chen, C. Hsu, H. Lin, P. Schwabe, M. Tsai, B. Wang, B. Yang, and S. Yang. Verifying curve25519 software. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 299–309. ACM, 2014.
22. L.-A. Daniel, S. Bardin, and T. Rezk. Binsec / rel: Efficient relational symbolic execution for constant-time at binary level. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1021–1038. IEEE, 2020.
23. V. D'Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 73–87. IEEE Computer Society, 2015.
24. A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.
25. A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy. A verified, efficient embedding of a verifiable assembly language. In *Principles of Programming Languages (POPL 2019)*. ACM, January 2019.
26. Y. Fu, J. Liu, X. Shi, M. Tsai, B. Wang, and B. Yang. Signed cryptographic program verification with typed cryptoline. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1591–1606. ACM, 2019.
27. V. T. Hoang, T. Krovetz, and P. Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2015.
28. J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Proc. of the 42th Symp. on Princ. of Prog. Languages* (*POPL*). ACM, 2015.
29. V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.
30. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

31. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming* (*ESOP'05*), volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
32. D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.
33. A. Petcher and G. Morrisett. The foundational cryptography framework. In R. Focardi and A. C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015.
34. M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Z. B. 'e guelin. Hacl texttimes n: Verified generic SIMD crypto (for all your favorite platforms). *IACR Cryptol. ePrint Arch.*, 2020:572, 2020.
35. J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in f^*. *CoRR*, abs/1703.00053, 2017.
36. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012.
37. X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), 29(5), 2007.
38. B. Rodrigues, F. Pereira, and D. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of Compiler Construction*, 2016.
39. C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in GCC and LLVM. In A. Zeller and A. Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305. ACM, 2016.
40. Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: A timing attack on openssl constant time RSA. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 346–367. Springer, 2016.
41. J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. In *SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 10 2017.