FaCT: A Flexible, Constant-Time Programming Language

Sunjay Cauligi[†] Gary Soeller[†] Fraser Brown[⋆] Brian Johannesmeyer[†] Yunlu Huang[†]
Ranjit Jhala[†] Deian Stefan[†]
[⋆]Stanford University [†]UC San Diego

***Abstract*—We argue that C is unsuitable for writing timing-channel free cryptographic code that is both fast and readable. Readable implementations of crypto routines would contain high-level constructs like `if` statements, constructs that *also* introduce timing vulnerabilities. To avoid vulnerabilities, programmers must rewrite their code to dodge intuitive yet dangerous constructs, cluttering the codebase and potentially introducing new errors. Moreover, even when programmers are diligent, compiler optimization passes may still introduce branches and other sources of timing side channels. This status quo is the worst of both worlds: tortured source code that can still yield vulnerable machine code. We propose to solve this problem with a domain-specific *language* that permits programmers to intuitively express crypto routines and reason about secret values, and a *compiler* that generates efficient, timing-channel free assembly code.**

I. Introduction

Cryptographic routines must be fast. Though these routines are computationally intensive, they must not impose significant overhead on the operations that they are intended to secure. Cryptosystem designers take pains to ensure good performance; in fact, whole venues like *Fast Software Encryption* are dedicated to high-speed cryptography. As one extreme example, the Curve25519 elliptic curve cryptosystem set new speed records by making heroic use of floating-point hardware to accelerate integer arithmetic [33]. At the very least, though, developers implement cryptographic libraries in low-level, efficient languages like C.

The C language is fast, but it does not help crypto developers safely handle sensitive data. Like most general-purpose languages, C has no notion of "sensitive data" at all. Consequently, developers are on their own to avoid—and often *fail* to avoid—leaking secrets. For example, a developer implementing RSA decryption may betray information by branching on the bits of a secret key [41]. The developer follows the textbook RSA decryption algorithm, raising the cipher to the power of the secret key. This amounts to performing a squaring operation for each bit of the key—and, for each set bit, performing an additional bignum multiplication. An attacker can recover individual bits from the secret key by measuring the time it takes to decrypt a message: because of the bignum multiplication, decryption takes far longer if a bit is set. *Timing attacks* like this one are a practical concern. For example, implementations of both RSA [35, 41] and AES [32, 49] have been susceptible to timing attacks that allowed (even remote network) attackers to recover secret keys.

To avoid introducing timing vulnerabilities, developers translate unsafe C functions into safe *constant-time* functions using a selection of standard recipes. For example, to implement a safe version of the conditional assignment from the RSA function above, the alert developer rewrites "`if (secret) x = expr`" to avoid branching on the secret; they choose their favorite "recipe" to convert the snippet into constant-time code. Conceptually, this vulnerable code is equivalent to the branchless arithmetic assignment "`x = (secret * expr) + (1 - secret) * x`." In this rewrite, `x` takes on the value `expr` if the `secret` bit is equal to one; it remains the same when `secret` is equal to `0` (since `0 + 1 * x` is `x`). This rewrite is no longer vulnerable, as its runtime does *not* depend on the value of `secret`. There are other safe rewrite alternatives for the vulnerable RSA code: the developer could choose the bitwise rewrite "`x = (expr & -secret) | (x & (secret-1))`," or "`x ^= (expr ^ x) & -secret`," or another folk remedy for curing timing-vulnerable crypto code.

But following constant-time rewrite recipes is error-prone. For example, the "Lucky 13" timing vulnerability in OpenSSL [26] was introduced in a patch intended to fix a *different* timing attack vulnerability in TLS [45]. It didn't end there. The patch addressing Lucky 13 itself introduced a new, separate vulnerability—CVE-2016-2107 [57].

Even if the developer manages to string together a correct series of constant-time translations, they face a further challenge: the compiler. Compiler optimization passes can introduce new vulnerabilities into correctly written code. For example, an optimization pass may introduce branching instructions in originally branchless C code [46]. Compilers may also generate instructions that have inherent data-dependent timing variations, like `div` and `idiv` [28]. As a result, crypto developers must manually inspect all generated assembly code to ensure that it only consists of instructions that handle secret data "safely"—but figuring out which instructions are safe can seem like divination. Cryptography Stack Exchange user `dave_thompson_085` sums up the frustration: "you aren't guaranteed that even bitwise operations are constant-time but in practice they're your best bet" [21].

We propose a domain-specific language and compiler, FaCT, for writing correct, fast, maintainable, and readable constant-time routines without resorting to rewrite recipes. Developers use the FaCT language to write idiomatic code like "if (secret) x = expr" in a *.fact file. The FaCT type system stops developers from accidentally leaking sensitive data. Since the FaCT compiler is aware of which data are sensitive, it can generate constant-time machine code and avoid optimizations that betray secrets. The FaCT compiler produces object files that developers link with their existing projects (e.g., OpenSSL); they write their general-purpose code in C and only rely on FaCT for constant time routines.

II. Constant-time C: traps and pitfalls

It is notoriously difficult to write correct constant-time code in C [2–4, 7, 20, 21, 25–27, 32, 35]. To avoid leaking sensitive data through timing side-channels, developers must avoid common C language features like if statements on secret data; moreover, they must structure their code to prevent the compiler from introducing timing variabilities during optimization passes. In this section, we recount Homerean tales of developers avoiding and succumbing to constant-time C traps. These Odysseys reveal how general-purpose languages like C render writing safe crypto code a Herculean challenge.

A. Developers must avoid unsafe language features

Since C has no notion of data sensitivity, it can't treat computations on sensitive data any differently from those on public data. Developers must manually structure their code so that it compiles to constant-time assembly *without* help from the language or compiler. To this end, developers often use a "safe" subset of C, encoding operations that rely on dangerous constructs like if statements into more attack-resilient alternatives—often straight-line bitwise operations. Below, we describe some of the unsafe constructs and the recipes for encoding the constructs in the safe C subset.

Control-flow constructs As mentioned in the introduction, developers should not use control flow constructs that branch on secret data. For example, the following code is unsafe:

```
if (secret)
  for( i = 0; i < Y->n; i++ )
    X->p[i] = Y->p[i];
```

This code assigns every element of Y->p to a corresponding location in X->p if the boolean flag secret is true. Unfortunately, this code is not constant-time—the if branch iterates Y->n times while the else branch is a no-op. An attacker who can measure the duration of the computation (e.g., over the network [35]) can infer the value of secret. Furthermore, even if the duration of both branches were the same, a local attacker can infer secret through other means like the data cache [32] or the branch predictor [24, 25, 39].

Safely expressing control flow statements (e.g., conditional, iterative, and switch statements [40]) on sensitive data amounts to transforming the statements into *branch-free*, arithmetic or bitwise expression statements [46]. Following this kind of recipe, a developer could rewrite the snippet in the previous paragraph to be constant-time:

```
/* mbedtls/library/bignum.c */
for( i = 0; i < Y->n; i++ )
  X->p[i] = X->p[i] * ( 1 - secret ) + Y->p[i] * secret;
```

In this mbedTLS version of the code snippet, the conditional is folded into the for-loop and then transformed into an arithmetic expression, so the control flow of the program is no longer affected by secret. Unfortunately, this example is deceptively simple—transforming programs with only slightly more interesting control flow (e.g., one that has nested if statements) becomes unwieldy [18].

Logical and relational expressions Conditional statements and expressions obviously compile to assembly code that contains branches. These statements, however, are not the only source of timing-vulnerable code: logical expressions—expressions involving &&, ||, and !—and relational expressions—expressions involving >, !=, <, etc.—do not always compile down to constant-time code. For example, mbedTLS developers deemed a function get_zeros_and_len_padding unsafe because a logical-expression (involving ||) was compiled to a branch [50].

To avoid traps like this, some crypto libraries (e.g., OpenSSL and libsodium) forgo C's built-in operators and, instead, rely on helper functions that implement the operators in constant time. This approach requires reimplementing almost all of C's operators. Moreover, it can require transforming even public expressions into different representations, ones that the rewritten functions expect.

Other libraries (e.g., mbedTLS, libsignal, and Crypto++) only transform logical and relational expressions when necessary. To this end, their developers inspect the assembly code and, if the generated assembly contains branches, they rewrite the culprit C code into a series of expression-statements using bitwise operators. This approach is brittle because it requires developers to examine assembly generated by different compilers, compiler versions, and optimization levels [20]. Some transformations are necessary for certain architectures but not others; some are only necessary when using certain compilers, compiler versions, or when certain optimizations are enabled. Finally, these selective rewrites often sacrifice readability, a trade-off that developers don't always find worthwhile [18].

Memory access Developers must also ensure that memory access patterns do not reveal information about secret data. Consider, for example, the mbedTLS get_zeros_padding function. This function takes a buffer of secret data, whose length is public, and computes the number of elements in the buffer before the all-0x00 suffix. Even a function this simple can leak secret information. For example, the original mbedTLS implementation [43] scanned the input buffer from the tail end until it found first non-zero element, exposing the length of the data in the process:

```
/* mbedtls/library/cipher.c */
static int get_zeros_padding( unsigned char *input,
                    size_t input_len, size_t *data_len ) {
        unsigned char *p = input + input_len - 1;
        ...

        while( *p == 0x00 && p > input )
            --p;

        *data_len = *p == 0x00 ? 0 : p - input + 1;
        ...
    }
```

By measuring the duration of the function, an attacker can determine how many iterations the **while** loop performs—the length of the padding—and therefore the secret length of the data. This is possible because the loop ends immediately after it identifies the first non-zero byte.

In general, accessing memory based on secret data is unsafe. To safely express memory access computations, developers must remove any dependency on secret values. Most often, this means transforming the memory access code to instead depend on publicly computable bounds that encompass the secret. For example, the mbedTLS developers rewrote the buggy function above to iterate over all elements of the buffer [51]. The function now updates the secret data length every iteration, using the done and prev_done variables to mask the proper values:

```
  /* mbedtls/library/cipher.c */
    static int get_zeros_padding( unsigned char *input,
                    size_t input_len, size_t *data_len ) {
        size_t i;
        unsigned char done = 0, prev_done;
        ...

        *data_len = 0;
        for( i = input_len; i > 0; i-- )
        {
            prev_done = done;
            done |= ( input[i-1] != 0 );
            *data_len |= i * ( done != prev_done );
        }
        ...
    }
```

This is not an isolated example: BoringSSL implements Base64 encoding and decoding similarly [31].

Variable-time, data-dependent operations Developers must also avoid using C language features and standard library functions whose running time varies according to input values [7]. For example, C operators like division and modulo often get compiled to instructions whose cycle time depends on the input values. On some architectures (e.g., ARM and PowerPC), even multiplication can be variable-time [8]! Similarly, some standard library buffer comparison functions like memcmp return early depending on the values of their input buffers [7]. An attacker can often leverage these variable-time operations to learn the contents of individual words or, sometimes, whole buffers [28].

Developers avoid variable-time operations, once again, by using transformations. For example, they translate divisions and modulo operations into explicit bit shifts. Similarly, they rewrite string and buffer comparisons to constant-time alternatives: some crypto libraries rely on comparisons implemented by an underlying library, while others (e.g., mbedTLS) resort to manually transforming uses of functions like memcpy to constant-time memory access patterns. This transformation is both easy to forget and easy to get wrong [29].

B. Compiler optimizations may introduce vulnerabilities

Like the C language itself, C compilers do not account for data sensitivity, and may therefore introduce new vulnerabilities during optimization passes. In other words, compilers may take C code that *appears* to be constant-time—code that does not include problematic constructs from the previous section (e.g., conditional, relational, or logical statements)—and still generate assembly that is vulnerable to timing attacks. Consider, for example, the following C code that implements the ternary operator according to a constant-time recipe [7]:

```
/* Return either x or y depending on whether bit is set */
uint32_t ct_select_u32(uint32_t x, uint32_t y, bool bit) {
    // Generate a mask: 0xFFFFFFFF if bit != 0, 0 otherwise
    uint32_t mask = ct_mask_u32(bit);
    return (x&mask) | (y&~mask);
}
```

When compiled with clang versions 3.9.1 or 4.0.0 using flags -O2 -m32 -march=i386, the generated code is *not* constant-time: the compiler generates code that contains conditional jump instructions.

Compilers may not only introduce dangerous operations like conditional jumps, but also optimize away seemingly superfluous code—code that exists to prevent information leaks. For example, in Crypto++, the compiler removed unused array reads that developers had added on purpose. These reads were meant to preload a cache line so that memory accesses at secret indices did not leak the indices via cache access patterns. Unsurprisingly, since the compiler optimized away the reads, the library was vulnerable to a timing attack [11].

C. Compilers may not generate optimal constant-time code

Today's approach to writing constant-time in a subset of C is not only extremely difficult and error-prone, but also comes at the cost of performance. Specifically, when transforming high-level idiomatic C code to a series of low-level arithmetic and bitwise operations developers also introduce a semantic gap that makes makes it more difficult for compilers to generate optimal code. This semantic gap is often *purposeful*: by transforming conditional statements into bitwise operations, developers conceptually "trick" the compiler into generating straight-line assembly. Unfortunately, when maneuvering the compiler away from dangerous instructions like conditional jumps, developers also maneuver away from safe, more efficient instructions. Below we describe some of these underutilized hardware instructions.

Conditional move (*cmov*) Intel's recent family of constant-time [36] conditional move instructions, cmov, are a prime example of underutilized hardware. These instructions check the CPU status flags and perform a move depending on flags. In practice, most of the constant-time transformations for conditional statements simulate a cmov using arithmetic or bitwise operations. Unfortunately, in our testing, only the arithmetic-based transformations compile to `cmov` instructions. Neither `clang` nor `gcc` optimize bitwise transformations to their equivalent `cmov` operations.

Byte swap (`bswap`) The family of byte swap instructions (`bswap`)—instructions that change the endianness of a value in constant time [23]—are another example of hardware features that can benefit crypto libraries. Unfortunately, as with `cmov`, most libraries implement equivalent computations using a series of bitwise operations, in turn, making it difficult for compilers to leverage `bswap`.

Add-with-carry (`adc`) Almost all processors have add-with-carry (`adc`) instructions. Since C does not expose these instructions at the language level, though, libraries like mbedTLS and wolfSSL manually recompute carry bits—via several bit shifting operations—when implementing "bignum" arithmetic. This reimplementation not only hurts readability of already hard-to-read constant-time code, but also leads to less efficient code—multiple instructions are used in place of a single `adc`.

Vector operations Most modern processors are also equipped with *single instruction multiple data* (SIMD) instructions that allow developers to perform operations on multiple, packed operands in parallel. Crypto libraries take advantage of these instructions by using elaborate macro definitions that boil down to either intrinsic functions [12, 19]—compiler-provided assembly-instruction wrappers—or inline assembly [20]. For example, the Crypto++ implementation of the Galois counter mode algorithm uses a set of preprocessor directives that check the compiler, compiler version, and architecture type to generate appropriate *macros* for accessing packed operations as such. Unfortunately, this approach is not only error-prone—macros that generate macros of inline assembly—but also does not scale: while some library developers can take on the job of a compiler to generate SIMD instructions for certain architectures, other libraries simply do not leverage these hardware features.

III. FaCT: Flexible and constant-time

We propose to solve the hazards outlined in the previous section by designing a new, flexible and constant-time (FaCT) programming language. FaCT is *not* a general purpose language. Rather, it is a domain specific language (DSL) explicitly designed to be used within existing, larger C crypto projects (e.g., wolfSSL or mbedTLS). Developers can continue to write the majority of their crypto code in C, and only need to use FaCT for the low-level functions that must to be resilient to timing attacks.

FaCT is designed to: (1) allow developers to easily write idiomatic code that runs in constant time, (2) be flexible enough to express real-world crypto code, (3) interoperate with C code, (4) produce fast assembly code, and (5) be verified to be resilient against timing attacks. In the rest of this section we describe how FaCT addresses these design points.

Easy to write constant-time code FaCT allows developers to write idiomatic, C-like code while ensuring that any code that computes on sensitive data is compiled to assembly that runs in constant time. To accomplish this, FaCT forces developers to specify the sensitivity of function arguments and return values—either **`public`** or **`secret`**—alongside their types. Using these **`public`** and **`secret`** labels, the FaCT compiler frontend infers the types and labels of intermediate values with an information flow control (IFC) type system. The compiler ensures that developers cannot express computations that would leak **`secret`** data [44, 52, 56]. For example, if a code snippet tries to access array elements based on a **`secret`** index, the compiler will report an error: the code snippet will not compile. The compiler also ensures that any variable-time operations (e.g., division or modulo) are restricted to **`public`** operands and that loops are bounded by **`public`** expressions. These restrictions force developers to reason about data sensitivity and write code that is resilient to timing leaks.

FaCT also allows developers to safely and idiomatically write conditional, logical, and relational expressions on secret values. The FaCT compiler will transform this idiomatic code into constant-time assembly. For example, with FaCT, we can write the `get_zeros_padding` mbedTLS function using standard programming language constructs:

```
public uint32 get_zeros_padding( secret uint8[] input,
                                 secret mut size_t data_len ) {
    ...
    for ( size_t i from len input to 1 by -1 ) {
        if ( input[i-1] == 0 )
            data_len = i;
    }
    ...
}
```

The FaCT compiler, as shown in Fig. 1, automatically transforms this code (e.g., by executing both branches of a secret conditional and merging stores, as in [44, 52]) to generate straight-line, constant-time assembly.

Finally, the FaCT system can do more than just code transformation to make constant-time code easy to write. The compiler can ensure that developers do not violate memory safety by using an SMT solver (e.g., Z3 [37]) to statically check that memory accesses are within bounds. Furthermore, FaCT provides a debug mode the allows developers to use otherwise unsafe operations (e.g., C's `printf`), to generate code that has not been transformed to constant-time, and to interact with their implementation using a read-eval-print loop (REPL) tool.

Express real-world crypto operations To ensure that FaCT is expressive enough to capture practical, real-world constant-time crypto code, we manually analyzed several popular cryptographic libraries and codebases [5, 9, 13–15, 17, 22]. Beyond standard features (e.g., functions, arrays, loops, mutable variables, etc.), FaCT has several built-in features specific to crypto code, features that we outline in Fig 2. For example, FaCT has built-ins for manipulating, testing, and packing bits, bytes, and words, since many hash functions and encryption algorithms rely on such operations. FaCT also exposes types

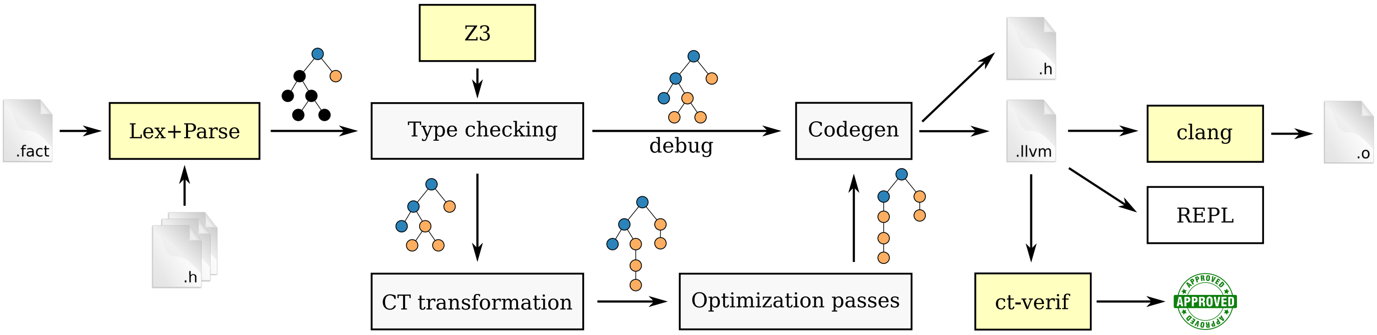

Fig. 1. The FaCT compiler consists of several phases. The type checking phase entails type and label inference as well as information flow and memory safety checks. The CT transformation phase applies a constant-time recipe over the typed AST before public and secret optimizations occur. The codegen phase generates label-annotated LLVM IR as well as the necessary C header files. Using ct-verif, the LLVM IR is verified to be constant-time before being sent to clang to produce an object file. Finally, when used in debug mode, the REPL will use the LLVM MCJIT execution engine to compile and execute FaCT code on the fly. We highlight the components that are trusted to be correct in yellow.

and operators for add-with-carry and vector operations, as we have observed many real libraries abuse the C macro system to accomplish a similar goal.

C interoperability FaCT is a DSL intended to be used within a larger C codebase. The FaCT compiler produces both an object file and a C header file (see Fig. 1). Developers can link the object file into an existing project and include the header files in order to call exposed FaCT functions as C functions.

We designed FaCT to interoperate with C code. To this end, all datatypes in FaCT have the same memory layout as their corresponding C datatypes. FaCT functions that are exposed to C code end up having very similar type declarations. Consider, for example, the following FaCT function:

```
export public int16 fn( public int32 value,
                        public mut uint32 modifiable_value,
                        secret mut uint8[8] fixed_len_array,
                        secret uint8[] var_len_array ) {
    ...
}
```

For this function, the FaCT compiler will generate a C header file with the following C function declaration:

```
// generated C declaration
/* public */ int16_t fn(
        /* public */ int32_t  value,
        /* public */ uint32_t *  modifiable_value,
        /* secret */ uint8_t  fixed_len_array[8],
        /* secret */ const uint_8 *  var_len_array,
        /* public */ size_t  _var_len_array_len );
```

Given the general similarity between FaCT and C function types, we only highlight some of the notable differences. First, `public` and `secret` labels have no meaning in C; when generating C declarations, however, FaCT leaves them in as comments for documentation. Simple types are passed by value, while `mut` types are passed by address. Structs and arrays are also passed by address. More subtly, FaCT functions that accept arrays with dynamically specified length are silently transformed to accept an additional, implicit length parameter. The corresponding C declaration reveals this extra parameter, so that C functions can provide the proper information upon invocation.

While we encourage developers to use FaCT for all low-level crypto code, we acknowledge that developers may need to call external assembly or C functions (highly tuned hash function implementations, for example). To this end, FaCT provides the `unsafe_foreign` function declarator that allows developers to omit function definitions and only declare the types and labels of their arguments and return values. While using such functions is generally considered unsafe, we can leverage verification tools like ct-verif [27] to ensure that foreign functions cannot leak sensitive data. This allows FaCT to retain its security guarantees even when calling C or assembly code.

Fast assembly code By bridging the semantic gap between the language features exposed to developers and the underlying hardware, FaCT can not only ensure security more easily, but also generate faster code. For example, because FaCT developers do not unnecessarily transform conditional statements into arithmetic or bitwise operations, the FaCT compiler can often emit fast instructions such as `cmov` to implement conditionals. Since FaCT exposes several language features (see Fig. 2) that have direct instruction-level analogues, the FaCT compiler can simply emit instructions to leverage these hardware features on architectures like x86-64. On architectures that do not natively have analogous instructions, the FaCT compiler can easily generate code that has the equivalent functionality.

In addition to using fast, constant-time instructions, the FaCT compiler can optimize code in a way that preserves the security guarantees of the IFC type system. Specifically, since we propagate sensitivity labels to the optimizer, the FaCT compiler can aggressively optimize the parts of the program that operate on `public` data while applying less aggressive (but safe!) optimizations to the `secret` parts (as suggested by [38]).

Resilient against timing attacks The FaCT compiler preserves label information through different compilation stages and eventually generates label-annotated LLVM. This label-annotated LLVM is then directly fed to the ct-verif tool,

Feature	Description	Example
Add-with-carry	Add two integers, producing carry bit in addition to sum	`sum, carry = value1 + value2;`
Byte packing	Treat multiple smaller words as one larger word	`large_word = pack(a, b, c, d);`
Byte unpacking	Split one large word into multiple smaller words	`a, b, c, d = unpack(large_word);`
Bit extraction	Extract the value of a single bit from a word	`bit_value = get_bit(word, index);`
Bit setting	Set the value of a single bit in a word	`set_bit(word, index, new_value);`

Bit rotation	Shift bits, adding the shifted bits back on the other end	`rotate_l = word <<< n;`
		`rotate_r = word >>> n;`

Vector operations	Parallel arithmetic on packed values	`vec1 += vec2;`
		`vec1 ^= vec2;`

Vector operations with saturation	Parallel arithmetic on packed values, without carrying (modular polynomial operations)	`vec1 .+= vec2;`
		`vec1 .*= vec2;`

Fig. 2. Non-standard language features in FaCT.

which verifies that the generated code does not leak any `secret` data [27]. An alternative approach to verifying the compiler output would be to verify the compiler itself—by proving noninterference [56] and translation validation [47]—and ensure that it *always* generates constant-time assembly. While potentially interesting, our current approach of using ct-verif is drastically simpler and more flexible. Indeed, ct-verif allows us to modify parts of the compiler to, say, add potentially dangerous optimization passes without giving up on security or having to prove anything: if a particular optimization leads to unintended leakages, ct-verif will catch it, and the FaCT compiler can fall back to less risky ones.

IV. Related Work

There are several approaches to verifying whether code is constant-time, from static analysis [27, 55], to dynamic taint tracking [6] and fuzzing [54]. These verification tools are useful for ensuring that neither the developer nor the compiler have accidentally introduced timing variations in their code. However, they do not save developers from having to write or understand abstruse constant-time C code.

Many proposed solutions to producing code free of timing side channels perform source-to-source transformations [30, 46, 48]. Other systems operate on annotated C code, but because C's relaxed type system complicates IFC typing, these systems are conservative and transform more code to constant-time than necessary [36, 46, 53]. To address timing differences in conditional statements, some of these approaches use transactions to execute both sides of a branch while only committing the "correct" branch, similar to our approach [30, 53]. Unfortunately, code generated by the above tools is still subject to general compiler optimizations. Other systems rely on an assembly-like language [16], or otherwise operate directly on annotated assembly [34] to avoid compiler pitfalls, giving up on the portability of writing code in a high-level language.

HACL [61] allows developers to write cryptographic functions in F* [58], a high level functional language, and verify that these functions do not leak information via covert-channels, such as timing. Their system, however, does not perform any automatic program transformations, leaving developers to manually write their code using the same constant-time recipes as in C.

The problem of compiler optimizations adversely impacting security is well known and goes beyond introducing timing attack vulnerabilities [38]. For example, dead store elimination—an optimization which removes stores that are never read from again—has been known to remove sensitive data scrubbing operations. This optimization alone has littered bug reports going as far back as 2002 [1] and still plagues developers with vulnerabilities through at least January 2016 [10]. In response, many developers reputedly devise source code level, compiler-specific "solutions" that trick the compiler into leaving the security relevant code unoptimized [60]. Other systems propose to introduce new LLVM passes that instrument the compiler itself and prevent it from making unsafe optimizations (e.g., [42, 59]).

Acknowledgements

We thank David Kohlbrenner, Ariana Mirian, Hovav Shacham, and the anonymous reviewers for their insightful comments and suggestions. We give special thanks to Riad S. Wahby for his superb skill in prettifying our code snippets and formatting our paper. This work was supported in part by NSF grant CNS-1514435.

References

[1] 8537 – Optimizer removes code necessary for security. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537.

[2] 868948 – A patch for NSS: a constant-time implementation of the GHASH function of AES-GCM, for processors that do not have the AES-NI/PCLMULQDQ. https://bugzilla.mozilla.org/show_bug.cgi?id=868948.

[3] Added more constant-time code / removed biases in the prime number generation routines. https://github.com/ARMmbed/mbedtls/pull/182.

[4] AES timing attack countermeasures. https://github.com/weidai11/cryptopp/commit/c8e2f8959414846031634477b2a0614434843ca3.

[5] BearSSL. https://bearssl.org/gitweb/?p=BearSSL.

[6] Checking that functions are constant time with valgrind. https://github.com/agl/ctgrind/.

[7] Coding rules. https://cryptocoding.net/index.php/Coding_rules.

[8] Constant-time MUL. https://www.bearssl.org/ctmul.html.

[9] Crypto++. https://github.com/weidai11/cryptopp.

[10] CVE-2016-0777. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.

[11] CVE-2016-3995. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3995.

[12] ”Jumbo” update for crypto/modes. https://github.com/openssl/openssl/commit/f472ec8c2f354314d278e11be567b43630acf090.

[13] libsodium. https://github.com/jedisct1/libsodium.

[14] mbed TLS. https://github.com/armmbed/mbedtls.

[15] OpenSSL. https://github.com/openssl/openssl.

[16] qhasm: tools to help write high-speed software. https://cr.yp.to/qhasm.html.

[17] Signal protocol C library. https://github.com/WhisperSystems/libsignal-protocol-c.

[18] Simplify some constant-time code. https://github.com/ARMmbed/mbedtls/commit/2ee8d24ca273487caa0b9b75e8791db75a77f51e.

[19] speed up GCM key setup. https://github.com/weidai11/cryptopp/commit/35820c22c226ac9b25a995efafadee708ca3c27f.

[20] Why constant-time crypto? https://www.bearssl.org/constanttime.html.

[21] Why not use `<`, `>` or `==` in constant time comparison? https://crypto.stackexchange.com/a/39432.

[22] wolfSSL. https://github.com/wolfSSL/wolfssl.

[23] Intel® 64 and IA-32 architectures software developer’s manual. *Volume 2: Instruction Set Reference, A-Z*, 2016.

[24] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *2nd ACM Symposium on Information, Computer, and Communications Security*, pages 312–320. ACM, 2007.

[25] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *Cryptographers’ Track at the RSA Conference*, pages 225–242. Springer, 2007.

[26] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *34th IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[27] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium*. USENIX Association, 2016.

[28] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *36th IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[29] P. Bakker. Check HMAC in constant-time in crypt_and_hash. https://github.com/ARMmbed/mbedtls/commit/424cd69.

[30] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[31] D. Benjamin. Implement base64 in constant-time. https://boringssl-review.googlesource.com/c/15047/.

[32] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[33] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[34] 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 *26th USENIX Security Symposium*. USENIX Association, 2017.

[35] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[36] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy,*, pages 45–60. IEEE, 2009.

[37] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[38] V. D’Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops, 2015 IEEE*, pages 73–87. IEEE, 2015.

[39] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–13. IEEE/ACM, 2016.

[40] I. ISO. IEC 9899-2011: Programming languages—C. *ISO Working Group*, 14, 2012.

[41] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*, pages 104–113. Springer, 1996.

[42] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *23rd ACM Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[43] Manuel Pégourié-Gonnard. Add zero padding. https://github.com/ARMmbed/mbedtls/commit/0e7d2c0f9537d66007a1a914f4c7e5b064f4c6ac.

[44] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF)*. IEEE, June 2012.

[45] B. Moeller et al. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. 2004.

[46] 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 *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[47] G. C. Necula. Translation validation for an optimizing compiler. In *ACM sigplan notices*, volume 35, pages 83–94. ACM, 2000.

[48] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *38th IEEE Symposium on Security and Privacy*, pages 710–728. IEEE, 2017.

[49] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[50] M. Pégourié-Gonnard. Make get_zeros_and_len_padding() constant-time. https://github.com/ARMmbed/mbedtls/commit/d17df51277d74ba6f487b3e72c4c0bb4ba55eb9c.

[51] M. Pégourié-Gonnard. Make get_zeros_padding() constant-time. https://github.com/ARMmbed/mbedtls/commit/e68bf171eb9ac92404d42c54984ea90101899430.

[52] J. Planul and J. C. Mitchell. Oblivious program execution and path-sensitive non-interference. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 66–80. IEEE, 2013.

[53] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, pages 431–446. USENIX Association, 2015.

[54] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? In *2017 Design, Automation & Test in Europe Conference & Exhibition*, pages 1697–1702. IEEE, 2017.

[55] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *25th International Conference on Compiler Construction*, pages 110–120. ACM, 2016.

[56] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[57] J. Somorovsky. Curious padding oracle in OpenSSL (CVE-2016-2107). https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html, 2016.

[58] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[59] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *24th ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[60] Z. Yang, B. Johannesmeyer, A. Trier Olesen, S. Lerner, and K. Levchenko. Dead store elimination (still) considered harmful. In *26th USENIX Security Symposium*. USENIX Association, 2017.

[61] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library.