FaCT: A DSL for Timing-Sensitive Computation

Sunjay Cauligi[†] Gary Soeller[†] Brian Johannesmeyer[†] Fraser Brown[★] Riad S. Wahby[★]

John Renner[†] Benjamin Grégoire[♠] Gilles Barthe[♣♦] Ranjit Jhala[†] Deian Stefan[†]

[†]UC San Diego, USA [★]Stanford, USA [♠]INRIA Sophia Antipolis, France
[♣]MPI for Security and Privacy, Germany [♦]IMDEA Software Institute, Spain

Abstract

Real-world cryptographic code is often written in a subset of C intended to execute in constant-time, thereby avoiding timing side channel vulnerabilities. This C subset eschews structured programming as we know it: if-statements, looping constructs, and procedural abstractions can leak timing information when handling sensitive data. The resulting obfuscation has led to subtle bugs, even in widely-used high-profile libraries like OpenSSL.

To address the challenge of writing constant-time cryptographic code, we present FaCT, a crypto DSL that provides high-level but safe language constructs. The FaCT compiler uses a secrecy type system to automatically transform potentially timing-sensitive high-level code into low-level, constant-time LLVM bitcode. We develop the language and type system, formalize the constant-time transformation, and present an empirical evaluation that uses FaCT to implement core crypto routines from several open-source projects including OpenSSL, libsodium, and curve25519-donna. Our evaluation shows that FaCT's design makes it possible to write *readable*, high-level cryptographic code, with *efficient*, *constant-time* behavior.

CCS Concepts • **Security and privacy → Cryptography**; • **Software and its engineering → General programming languages**; • **Theory of computation → Operational semantics**.

Keywords domain-specific language, program transformation, cryptography

ACM Reference Format:
Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314605

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314605

1 Introduction

Despite many strides in language design over the past half-century, modern cryptographic routines are still typically written in C. This is good for speed but bad for keeping secrets. Like most general-purpose languages, C gives the programmer no way to denote which data is sensitive—and therefore gives the programmer no warnings about code that inadvertently divulges secrets.

One possible avenue for secret leaks is a *timing side-channel*, wherein code executes for observably different time depending on the value of secret information. For example, a textbook implementation of RSA decryption takes a different amount of time depending on the individual key bits [35]—each '1' bit requires an additional bignum multiplication and thus more time. The cumulative effects of these operations on the running time is large enough for the attacker to reconstruct the value of the secret key. Timing vulnerabilities like these are not merely of academic interest: they have been found in implementations of both RSA [23] and AES [13, 46], where they allowed even remote network attackers to divine the values of secret keys.

The only recourse developers have to avoid timing vulnerabilities is to make their code ugly. Specifically, they use a selection of *recipes* to turn dangerous but readable code into safe but obfuscated code: they re-write potentially secret-revealing constructs like branches into low level sequences of assignments that operate in *constant-time* regardless of the values of secret data. For example, the readable

```
if (secret) x = e
```

which branches on a `secret` bit is replaced by

```
x = (-secret & e) | (secret - 1) & x
```

which, unlike the branch, executes in the same amount of time no matter the value of `secret`.

This is a sorry state of affairs. First, developers apply the recipes in an *ad-hoc* way, and any untransformed computation is left vulnerable to attack. Second, the recipes *obfuscate* the code, making it harder to determine whether the routine is even computing the desired value. Third, it can be tricky for developers to *correctly* apply the recipes. For example,

an attempt to use a recipe to fix a timing attack vulnerability in TLS [40] led to the Lucky13 timing vulnerability in OpenSSL [2], and the purported fix for Lucky13 opened the door to yet another vulnerability [57]!

In this paper, we introduce FaCT, a domain-specific language and compiler for writing *readable* and *timing-secure* cryptographic routines. FaCT lets developers write readable code using high-level control-flow constructs like branches and procedural abstractions, but then automatically compiles this code into efficient, constant-time executables. We develop FaCT via four contributions:

1. Language design. Our first contribution is the design of a language for writing cryptographic code. The language allows programmers to use standard control-flow constructs like `if` and `return` statements. However, the language is equipped with an *information-flow* type system that programmers can use to specify which data are `secret`. The type system prevents leaks by ensuring that `secret`s do not explicitly or implicitly influence the `public`-visible outputs (§3).

2. Public safety. Our second contribution is the observation that not all programs are amenable to constant-time compilation. Specifically, we show that naive application of constant-time recipes can mangle otherwise safe programs, causing memory errors or undefined behavior. We address this problem by introducing a notion called *public safety* that characterizes the source programs that can be compiled to constant-time without introducing errors (§3.2.3).

3. Constant-time compilation. Our third contribution is a compiler that automatically converts (public safe) source programs into constant-time executables. The FaCT compiler is based on the key insight that we can exploit the secrecy types to *automatically* apply the recipes that developers have hitherto applied by hand, and can do so *systematically*, i.e., exactly where needed to prevent the exposure of secrets via timing. We formalize the compiler with two transformations, *return deferral* and *branch removal*, and prove that compilation yields constant-time executables with source-equivalent semantics (§4).

4. Implementation & evaluation. Our final contribution is an implementation of FaCT that produces LLVM IR from high-level sources, and uses LLVM's `clang` to generate the final object or assembly file. We evaluate FaCT's *usability* with a user study, surveying students in an upper-level, undergraduate programming languages course at a U.S. university, where 57% of the participants found FaCT easier to write than C (compared to 15% who found C easier). We evaluate FaCT's *expressiveness* and *performance* by using our implementation to port 7 cryptographic routines from 3 widely used libraries: OpenSSL, libsodium, and curve25519-donna, totaling about 2400 lines of C source. The unoptimized FaCT code—which we *formally* guaranteed to be constant-time—is between 16–346% slower than the C equivalent. The `clang`-optimized FaCT code—which we *empirically* check to be constant-time using dudect [52]—is between 5% slower to 21% *faster* than the C equivalent, showing that FaCT yields readable constant-time code whose performance is competitive with C (§5).

We make all source and data available under an open source license at: https://fact.programming.systems.

2 Background

Some common C constructs—branches, returns, and array updates—can reveal secrets via timing channels. In this section, for each potentially dangerous construct, we explain: (1) how that construct could introduce bugs in real-world projects; (2) how developers must use recipes to avoid the dangerous construct; and, (3) how FaCT allows programmers to forgo recipes and write readable yet safe code.

Branching on secret values. A first class of vulnerability arises from directly branching on the value of a secret—attackers can often reconstruct control flow through a program, and thus secret condition values (e.g., because the `true` branch takes orders of magnitude longer to execute than the `false` branch) [49]. To avoid this type of vulnerability, developers manually translate branching code to straight-line code by replacing `if`-statements with constant-time bitmasks. Consider the following example from OpenSSL (edited slightly for brevity), which formats a message before computing a message authentication code (MAC):

```
for (j = 0; j < md_block_size; j++, k++) {
  b  = data[k - header_length];
  b  = constant_time_select_8(is_past_c, 0x80, b);
  b  = b & ~is_past_cp1;
  b &= ~is_block_b | is_block_a;
  block[j] = b;
}
```

It's hard to tell, but this snippet (1) iterates over plaintext message `data`, (2) terminates the message with standard-defined `0x80`, and (3) pads the terminated message to fill a hash block—all while keeping `data` secret. To this end, even the selection operator `constant_time_select_8(mask, a, b)` is a series of bitmasks: `(mask & a) | (~mask & b)`.

Translating each line of this OpenSSL code to FaCT leads to drastically more readable code:

```
for (uint64 j from 0 to md_block_size) {
  k += 1;
  b = is_past_c ? 0x80 : data[k - (len header)];
  if (is_past_cp1 || (is_block_b && !is_block_a)) {
    b = 0;
  }
  block[j] = b;
}
```

With FaCT, the programmer declares the sensitive variables as used in the conditions as `secret`. After doing so, they are free to use plain conditional expressions and ternary operators to compute the final value of `b`. The FaCT compiler

automatically uses the type annotations to generate machine code equivalent to the C example.

Early termination. Both loops and procedures can terminate early depending on the value of a secret, thereby leaking the secret. A well-known padding oracle attack in older versions of OpenSSL exploits an early function return [61]: a packet processing function would decrypt a packet and then check that the padding was valid, and, in the case of invalid padding, would return immediately. An attacker could exploit this to recover the SSL session key by sending specially crafted packets and use timing measurements to determine whether or not the padding of the decrypted packet was valid. Similarly, if the number of loop iterations in a program depends on a secret, attackers can use timing to uncover the value of that secret (e.g., in the Lucky13 attack [2]).

C programmers again use special recipes, turning idiomatic programs into hard-to-read constant-time code. Consider the following buffer comparison code from the libsodium cryptographic library:

```
for (i = 0; i < n; i++)
  d |= x[i] ^ y[i];
return (1 & ((d - 1) >> 8)) - 1;
```

This snippet compares the first `n` bytes of the arrays `x` and `y`, returning `0` if they are the same, and `-1` otherwise. To avoid leaking information about the contents of the arrays, though, the loop *cannot* simply return early when it detects differing values; instead, the programmer must maintain a "flag" (`d`), and update it at each iteration to signal success or failure. While iterating inside the loop, if the values `x[i]` and `y[i]` are the same, then `x[i] ^ y[i]` will be `0`, leaving `d` unchanged. However, if `x[i]` and `y[i]` are different, then their XOR will have at least one bit set, causing `d` to also have a non-zero value. After the loop, the code uses a complex shift-and-mask dance to collapse `d` into the value `-1` if any bits are set, and `0` otherwise.

FaCT lets programmers avoid the "flag" contortions:

```
for (uint64 i from 0 to n)
  if (x[i] != y[i])
    return -1;
return 0;
```

With FaCT, the programmer can readily express returning `-1` in the case of failure as the compiler automatically generates a special variable for the return value, and uses the `secret` type to translate returns-under-secret conditions into (constant-time) updates to this variable, producing machine code roughly equivalent to the C recipe above.

Memory access. Memory access patterns that depend on secret data can also inadvertently leak that secret data. An attacker co-located on the same machine as a victim process, for example, can easily infer secret memory access patterns by observing their own cache hits and misses [33, 46]; alarmingly, attackers might even learn such information across a datacenter—or even over the Internet [23, 53].

To avoid leaking information via memory access patterns, developers rely on recipes that avoid accessing memory based on secrets. The following C code (from the "donna" Curve25519 implementation), for example, swaps the values of array `a` with array `b` based on the value of a secret (`swap`):

```
for (i = 0; i < 5; ++i) {
  const limb x = swap & (a[i] ^ b[i]);
  a[i] ^= x;
  b[i] ^= x;
}
```

To avoid leaking the value of the secret `swap`, the code *always* accesses both `a[i]` and `b[i]` at each loop iteration, and uses bitmask operations that only change them if `swap` is a mask of all 1-bits.

FaCT, again, makes such subterfuge unneccessary:

```
if (swap != 0) {
  for (uint64 i from 0 to 5) {
    secret uint64 tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
  }
}
```

Once the programmer has marked `swap` as `secret`, the compiler will automatically synthesize masked array reads similar to those from the original Curve25519 code.

3 FaCT

FaCT is a high-level, strongly-typed C-like DSL, designed for writing constant-time crypto code. In this section, we describe the DSL and its type system, one that both disallows certain unsafe programs and specifies how the compiler should transform code to run in constant-time.[1] We describe the type-directed transformations in §4.

3.1 Core language

FaCT is designed to be embedded into existing crypto projects (e.g., OpenSSL), and not to be used as a standalone language. As such, FaCT "programs" are organized as collections of procedures. As shown in Figure 1, developers can export these procedures as C functions to the embedding environment. They can also *import* trusted procedures. This is especially useful when using FaCT to implement error-prone glue code around already known-safe C crypto primitives (e.g., building a block cipher mode that calls an AES primitive).

FaCT procedures are composed of a sequence of statements (e.g., `if` statements, `for` loops, etc.), which are themselves composed of expressions. Both statements and expressions are mostly standard. We only remark on the more

[1] The *surface* language as used by developers is slightly less verbose than the *core* language presented in this section. For example, our surface syntax allows procedures to be called in expressions; FaCT desugars such expressions into core language procedure-call statements. We refer to both the surface and core languages as FaCT.

Procedure definitions

fdef ::=		
|	$f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta$	internal procedure
|	`export` $f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta$	exported procedure
|	`extern` $f(\vec{x} : \vec{\beta}) : \beta$	external procedure

Statements

S ::=		
|	$S; S$	sequence
|	$\beta\ x = e$	variable declaration
|	$\beta\ x = f(\vec{e})$	procedure call
|	$e := e$	assignment
|	`if` $(e)\ \{\ S\ \}$ `else` $\{\ S\ \}$	conditional
|	`for` $(x$ `from` e `to` $e)\ \{\ S\ \}$	range-for
|	`return` e	return

Expressions

e ::=		
|	`true` | `false`	boolean literal
|	n	numeric literal
|	x	variable
|	$\ominus e$	unary op
|	$e \oplus e$	binary op
|	$e[e]$	array get
|	`len` e	array length
|	`zeros`(β, e)	zero array
|	`clone`(e)	array clone
|	`view`(e, e, e)	array view
|	`declassify`(e)	declassify
|	`assume`(e)	assume
|	`ref` e	reference
|	`deref` e	dereference
|	`ctselect`(e, e, e)	constant-time selection

Figure 1. (Subset of) FaCT grammar.

notable language constructs we add to make writing cryptographic code more natural.

First, FaCT includes a number of *array primitives* to capture common idioms in cryptographic routines, and to replace unsafe pointer arithmetic. The operation `len` e returns the length of an array e; `zeros`(β, e) creates an array of zeros of type β of length e; `clone`(e) copies the array e; and `view`(e_1, e_2, e_{len}) returns a *slice* of array e_1 starting at position e_2 and with length e_{len}. We introduce `view`s to make up for the lack of pointers: `view`s allow developers to efficiently compute on small pieces of large buffers.

Second, we provide *vector primitives*: parallel vector arithmetic and vector shuffle instructions. These instructions allow developers to implement crypto algorithms that leverage fast SIMD instructions (e.g., SSE4 in x86_64) without resorting to architecture-specific inline assembly or compiler intrinsics.

Labels

ℓ ::= Pub | Sec

Size

s ::= 8 | 16 | ... | 128

Array size

sz ::= $*$ | 0 | 1 | ...

Mutability

m ::= R | RW

Base types

β ::= Bool_ℓ | (U)Int^s_ℓ | $\text{Ref}_m[\beta]$ | $\text{Arr}^{sz}[\beta]$ | $\left\{\vec{x} : \vec{\beta}\right\}$

Figure 2. FaCT types.

Third, we expose `ctselect`, a constant-time selection primitive. The operation `ctselect`(e_1, e_2, e_3) evaluates to either e_2 or e_3, depending on whether e_1 is `true` or `false`, respectively. The compiler guarantees that `ctselect` compiles to constant-time code (e.g., as a series of bitmasks or the `CMOV` instruction on x86_64). Developers need not use `ctselect` directly; instead, they can use our higher-level `if`-statements, which our compiler transforms to such `ctselect`s (§4).

Lastly, FaCT includes a `declassify` primitive that takes a secret expression as input and returns a public value. Developers can use this primitive to bypass FaCT's typing restrictions (described below) and explicitly make information public. This is useful, e.g., for implementing encryption: a buffer containing a secret message must be treated with care, but if the buffer is encrypted in-place, it is thereafter safe to `declassify` because it contains ciphertext.

3.2 Type system

The most important feature of the FaCT language is its static information-flow type system. We rely on this type system to: (1) provide a way for developers to demarcate the sensitivity of data—whether it is secret or public; (2) reject unsafe programs, i.e., programs that are not information-flow secure or cannot be safely transformed to constant-time code; and (3) direct the compiler when applying transformations. Below, we give an overview of our type system and explain how it fulfills the first two roles; we leave the third for §4.

Like previous information-flow type systems [42, 43, 55, 62], FaCT decorates each base type with a `secret` or `public` *secrecy label*[2]. Figure 2 summarizes our base types; they are largely standard. Reference types wrap another base type and inherit its secrecy label; they are also access controlled, i.e., they can be read-only or read-write. In the FaCT surface syntax, we disallow recursively-typed references—only references of integer and boolean types are expressible. Array types, like references, inherit the secrecy of their base type; arrays have a size which is either a statically known constant

[2]Labels are partially ordered according to $\sqsubseteq$ as usual: Pub $\sqsubseteq \ell$ and $\ell \sqsubseteq$ Sec holds true for any label ℓ. The join of two labels is similarly standard: $\ell_1 \sqcup \ell_2$ is Sec if either label is Sec, and Pub otherwise. For brevity, we also use these operators on types (operating on the underlying label), much like previous work (e.g., [42, 43]).

or dynamically determined (∗). Struct types *do not* carry a secrecy label; instead, each struct field is individually labeled.

Developers explicitly specify labels when they declare variables and procedures. FaCT's type system, in turn, uses these labels to reject unsafe programs and specify how the compiler should transform high-level code that uses seemingly unsafe constructs (e.g., `secret if`-statements) to constant-time code. Below, we walk through our typing rules for expressions, statements, and procedures.

3.2.1 Expression typing

FaCT's expression typing judgment $\Gamma \vdash e : \beta$ states that under the type context Γ, which maps variables to their declared types, the expression e has the type β. We write $x : \beta \in \Gamma$ when variable x maps to type β in the context Γ.

Figure 3 gives the typing rules for the most interesting expressions. The rule for `ctselect`, for example, ensures that (1) the result is at least as secret as all the arguments to `ctselect` and (2) all the arguments can be cast to integers—since, internally, `ctselect` may be implemented as a series of constant-time bitmasks. The typing rules for other constructs similarly preserve secrecy.

The type system also disallows certain kinds of unsafe computations. For example, we reject programs that index memory based on secrets: the rules for T-Arr-Get and T-Arr-View ensure that array indices are `public` and in-bounds. The in-bounds checks are highlighted, and detailed in §3.2.3.

3.2.2 Statement and procedure typing

FaCT allows developers to write code whose control flow depends on sensitive data. Unfortunately, not all such code can be safely or efficiently transformed. For example, to safely allow writes to arrays using a secret-dependent index we must (transform the code to) write to *all* indices [39, 47, 51]; such a transformation would be expensive, and FaCT instead disallows such computations. As such, typing rules for statements and procedures rely on a *secrecy context*, which comprises a pair of secrecy labels *pc*, *rc* called the *path* and *return* context, respectively.

The path context label *pc* for a statement is `secret` if the statement is contained within—i.e., is control-dependent upon—a `secret` branch. Since a procedure caller's path context must persist through to the callee's path context, the initial path context label of a procedure is `secret` if it is ever called from a `secret` context; otherwise the initial path context label is `public`. We use ω to map procedures to their initial path context labels.

The return context label *rc* for a statement is `secret` if the statement *may* be preceded by a `return` statement that is itself control-dependent on a `secret` value. A procedure's return context label is always initially public. Thus, the *secrecy context* ($pc \sqcup rc$) for a statement represents whether the flow of control (to get to the statement) can be influenced by `secret` values. For example, if the conditional expression of an `if` statement is `secret`, then the statements of each branch are judged with $pc = \textsc{Sec}$, and are thus typed under a `secret` context.

Statement typing. FaCT's statement typing judgment is of the form $\omega, pc, \beta_r \vdash S : \Gamma, rc \rightarrow \Gamma', rc'$, where β_r is the return type of the procedure containing the statement S. This judgment states that, given a type- and security- context defined by ω, pc, β_r and initial Γ, rc, the statement S: (1) can be safely compiled into constant-time code, and (2) yields a new updated type context Γ' and return context rc'. This typing judgment accounts for new variable declarations and ensures that the secrecy context influences subsequent statements. For example, if a `return` statement resides within a `secret` branch, then all statements executed after that branch must also be typed under a `secret` context, since their execution now depends on the `return`.

Figure 4 shows the most interesting statement typing rules. For example, (T-Asgn) checks that when updating a reference, the current secrecy context does not exceed the secrecy label of the value e_2 being assigned. This ensures that `secret` data cannot be leaked via control flow. Rules (T-If) and (T-Ret) account for such secret contexts; the latter additionally ensures that the procedure cannot return a value more sensitive than specified by the procedure return type.

Rule (T-For) is more restricting: it ensures that `secret`s do not influence the running time of `for` loops by requiring that the loop bounds—and therefore the number of iterations—be `public`. The updated return context rc' must both be a fixpoint of the loop, and must be no lower than the original return context rc. In practice, our type checker only assigns rc' to be `secret` if it cannot assign it to be `public`.

The typing for procedure calls given by (T-Call) is slightly more complex. In particular, this rule ensures that procedures can only be called with suitable inputs and checks that the output type is *compatible* with the variable being assigned. To this end, we ensure that if the procedure f has visible effects, then its initial path context $\omega(f)$ must be at least the label of the calling context. This, in effect, ensures that in a `secret` context we *cannot* call procedures that (1) modify `public` parameters, i.e., take mutable public references as input parameters; (2) are `extern`ally defined and so possibly have `public`ly visible side-effects; or (3) are `export`ed (top-level) procedures.

Procedure typing. Figure 5 shows rules for typing procedure definitions. FaCT's procedure typing judgment is of the form $\omega \vdash f(\vec{x} : \vec{\beta}) \{ S \} : \beta_r$, which states that under ω, the procedure f with named parameters $\vec{x}$ of types $\vec{\beta}$ has return type β_r. Procedures in FaCT may only return simple types (i.e., boolean values or integers), but there are no such restrictions on the types of parameters. When typing procedures, the initial type context Γ is formed from the procedure's parameters, and the initial path context pc is given by $\omega(f)$. The

T-Ct-Sel

$$\frac{\beta \text{ is numeric or Bool} \qquad \Gamma \vdash e_1 : \text{Bool}_\ell \qquad \Gamma \vdash e_2 : \beta \qquad \Gamma \vdash e_3 : \beta}{\Gamma \vdash \texttt{ctselect}(e_1, e_2, e_3) : \beta \sqcup \ell}$$

T-Arr-Get

$$\frac{\Gamma \vdash e_1 : \text{Arr}^{sz}[\beta] \qquad \Gamma \vdash e_2 : \text{UInt}^s_{\text{Pub}} \qquad \Gamma \Rightarrow e_2 < \texttt{len}\ e_1}{\Gamma \vdash e_1[e_2] : \beta}$$

T-Arr-View

$$\frac{\Gamma \vdash e_1 : \text{Arr}^{sz}[\beta] \qquad \Gamma \vdash e_2 : \text{UInt}^s_{\text{Pub}} \qquad \Gamma \vdash e_{len} : \text{UInt}^s_{\text{Pub}} \qquad sz' = szOfExpr(e_{len}) \qquad \Gamma \Rightarrow e_2 < \texttt{len}\ e_1 \qquad \Gamma \Rightarrow e_{len} \leq \texttt{len}\ e_1 - e_2}{\Gamma \vdash \texttt{view}(e_1, e_2, e_{len}) : \text{Arr}^{sz'}[\beta]}$$

Figure 3. (Subset of) FaCT expression typing rules.

return context rc always starts as Pub, as the procedure body S (vacuously) has no preceding `secret`-dependent `return` statements. The return type β_r is taken from the procedure definition. If the body S is well-typed under these initial contexts, then the procedure itself is considered well-typed.

3.2.3 Public safety

The FaCT type system ensures that procedures can be transformed using constant-time recipes without giving up safety. Naively applying recipes can inadvertently *introduce* safety and security vulnerabilities while making the code constant-time. Consider the following procedure:

```
void potential_oob( secret mut uint32[] buf
                  , public uint64 i
                  , secret uint64 secret_index ) {
  assume(secret_index <= len buf);
  if (i < secret_index)
    buf[i] = 0;
  ...
}
```

This code is memory safe as the branch condition ensures that we only update `buf[i]` when `i` is within bounds. However, the update is predicated upon a `secret` condition. To make the above code constant-time, we must ensure that the access to `buf[i]` happens regardless of that condition, or else the memory access pattern will reveal the secret. Consequently, the constant-time recipes—that we discuss in §4—would compile the code into:

```
cond   = (i < secret_index);
buf[i] = ctselect(cond, 0, buf[i]);
```

Such a naive transformation introduces a potential *out-of-bounds* access. In other cases it can introduce yet different kinds of undefined behavior.

Public safety. We avoid the above problem with the key observation that for a program to be amenable to constant-time compilation, the source must be *publicly safe*: It must

T-Call

$$\frac{\omega \vdash f(\vec{\beta}) : \beta \qquad hasEffects(f) \Rightarrow pc \sqcup rc \sqsubseteq \omega(f) \qquad \Gamma \vdash e_i : \beta_i \qquad \Gamma' = \Gamma, x : \beta}{\omega, pc, \beta_r \vdash \beta\ x = f(\vec{e}) : \Gamma, rc \to \Gamma', rc}$$

T-Asgn

$$\frac{\Gamma \vdash e_1 : \text{Ref}_W[\beta] \qquad \Gamma \vdash e_2 : \beta \qquad pc \sqcup rc \sqsubseteq \beta}{\omega, pc, \beta_r \vdash e_1 := e_2 : \Gamma, rc \to \Gamma, rc}$$

T-If

$$\frac{\Gamma \vdash e : \text{Bool}_\ell \qquad \omega, pc \sqcup \ell, \beta_r \vdash S_1 : \Gamma \wedge e, rc \to \Gamma_1, rc_1 \qquad \omega, pc \sqcup \ell, \beta_r \vdash S_2 : \Gamma \wedge \neg e, rc \to \Gamma_2, rc_2}{\omega, pc, \beta_r \vdash \texttt{if}\ (e)\ \{\ S_1\ \}\ \texttt{else}\ \{\ S_2\ \} : \Gamma, rc \to \Gamma, rc_1 \sqcup rc_2}$$

T-For

$$\frac{\Gamma \vdash e_1 : \text{UInt}_{\text{Pub}} \qquad \Gamma \vdash e_2 : \text{UInt}_{\text{Pub}} \qquad \Gamma' = \Gamma, x : \text{UInt}_{\text{Pub}} \wedge e_1 \leq x < e_2 \qquad rc \sqsubseteq rc' \qquad \omega, pc, \beta_r \vdash S : \Gamma', rc' \to \Gamma'', rc'}{\omega, pc, \beta_r \vdash \texttt{for}\ (x\ \texttt{from}\ e_1\ \texttt{to}\ e_2)\ \{\ S\ \} : \Gamma, rc \to \Gamma, rc'}$$

T-Ret

$$\frac{\Gamma \vdash e : \beta_r \qquad pc \sqcup rc \sqsubseteq \beta_r}{\omega, pc, \beta_r \vdash \texttt{return}\ e : \Gamma, rc \to \Gamma, pc \sqcup rc}$$

T-Assume

$$\frac{\Gamma \vdash e : \text{Bool}_\ell \qquad \Gamma' = \Gamma \wedge e}{\omega, pc, \beta_r \vdash \texttt{assume}(e) : \Gamma, rc \to \Gamma', rc}$$

Figure 4. (Subset of) FaCT statement typing rules.

T-Fn

$$\frac{pc = \omega(f) \qquad \Gamma = \{\vec{x} : \vec{\beta}\} \qquad \beta_r \text{ is numeric or Bool} \qquad \omega, pc, \beta_r \vdash S : \Gamma, \text{Pub} \to \Gamma', rc'}{\omega \vdash f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r}$$

T-Fn-Extern

$$\frac{\omega(f) = \text{Pub} \qquad \beta_r \text{ is numeric or Bool}}{\omega \vdash \texttt{extern}\ f(\vec{x} : \vec{\beta}) : \beta_r}$$

Figure 5. (Subset of) FaCT procedure typing rules.

be memory-safe and free from buffer overflows and undefined behavior using only `public`-visible information, i.e., the code must be safe even after removal of `secret`-dependent control-flow. We formalize the notion of public safety in FaCT's type system by extending the type environment Γ to track `public`-visible path conditions, using these conditions to check safety. In Figures 3 and 4 these public safety extensions are highlighted.

Public views. We first define the judgment $\Gamma \vdash_i e$ to mean that e is *immutable* in Γ; that is, e is only composed of constants, immutable variables, array lengths, or operations

thereon. Next, we define the operation $\Gamma \wedge e$, which *conjoins* Γ with a *public view* of the condition e: if e is a `public bool` ($\Gamma \vdash e : \text{Bool}_{\text{Pub}}$) and e is immutable ($\Gamma \vdash_{\text{i}} e$), then $\Gamma \wedge e$ represents the environment Γ with the additional assumption that e is true. Otherwise, $\Gamma \wedge e = \Gamma$, i.e., conjoining Γ with a `secret` condition does not add any new assumptions to Γ. Rules T-If and T-For in Figure 4 show how we propagate public views, tracking (`public`) conditions and loop ranges to use when type checking statements.

For cases where the public safety checker is incomplete, we allow developers to add assumptions directly to the environment Γ with the `assume` primitive (Figure 1). This is useful for aiding the checker by, e.g., adding preconditions to a procedure.

Checking public safety. Finally, we define $\Gamma \Rightarrow e$ to mean that the conditions in Γ *imply* e. This is checked via an SMT solver. We use this predicate in the expression typing rules T-Arr-Get and T-Arr-View (Figure 3) to check that memory accesses are never out of bounds. In the example program given earlier, since the expression `i < secret_index` is of type Bool_{Sec}, it is not added to Γ; thus the predicate $\Gamma \Rightarrow$ `i < len buf` does not hold when typing the expression `buf[i]`, and the program (correctly) does not type check.

The FaCT type system also prevents undefined behavior from invalid operand values (not shown in Figure 3). For example, integer division has the additional requirement $\Gamma \Rightarrow e_2 \neq 0$, and the left- and right-shift operators have the requirement $\Gamma \Rightarrow 0 \leq e_2 < s$ where s is the bitwidth of e_1.

4 Front-end compiler

The FaCT compiler consists of two passes. The first pass is a source to source transformation—it compiles well-typed code into semantically equivalent FaCT constant-time code whose observable timing is `secret`-independent. The second pass is straightforward—it takes the `secret`-independent code and generates LLVM bitcode. In the rest of the section, we thus only describe and formalize FaCT's transformation pass.

Since our type checker (§3.2) already ensures that memory accesses, loop iterations, and variable-time instructions are `secret`-independent, the transformations need only make procedure returns and branches `secret`-independent. FaCT does this in two steps, *return deferral* and *branch removal*.

The first step replaces `secret`-dependent `return` statements by (1) creating a boolean that represents whether the procedure has returned and (2) conditioning all later code on that boolean to prevent statements from executing after the original procedure would have terminated. That is, return deferral converts control flow in terms of `secret returns` into control flow in terms of `secret if`s.

The second step turns all `secret`-dependent conditional branches into straight-line code. This includes both `secret if` statements in the original source as well as those generated by return deferral. Thus, by eliminating `secret if`s—the last source of `secret`-dependent timing—this transformation yields constant-time code.

$$\textsc{Tr-Ret-Dec}\quad \frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \qquad \Phi, \omega(f), \text{Pub} \vdash S \longrightarrow S'}{\begin{array}{c}\omega \vdash f(\vec{\beta})\ \{\ S\ \} : \beta_r \ \longrightarrow \\ f(\vec{\beta})\ \{\ \ \text{Ref}_{\text{RW}}[\beta_r]\ \mathit{rval} = \mathit{init}(\beta_r); \\ \text{Ref}_{\text{RW}}[\text{Bool}_{\text{Sec}}]\ \mathit{notRet} = \texttt{true}; \\ S';\ \ \texttt{return}\ \mathit{rval}\ \ \}\ : \beta_r\end{array}}$$

$$\textsc{Tr-Ret-Guard-Pub}\quad \frac{\Phi, pc, \text{Pub} \vdash S \longrightarrow S'}{\Phi, pc, \text{Pub} \vdash S \rightsquigarrow S'} \qquad \textsc{Tr-Ret-Guard-Sec}\quad \frac{\Phi, pc, \text{Sec} \vdash S \longrightarrow S'}{\Phi, pc, \text{Sec} \vdash S \rightsquigarrow \texttt{if}\ (\mathit{notRet})\ \{\ S'\ \}}$$

$$\textsc{Tr-Ret}\quad \frac{pc \sqcup rc = \text{Sec}}{\Phi, pc, rc \vdash \texttt{return}\ e \longrightarrow \mathit{rval} := e; \mathit{notRet} := \texttt{false}}$$

$$\textsc{Tr-Ret-Seq}\quad \frac{\Phi = (\omega, \Gamma, \beta_r) \qquad \omega, pc, \beta_r \vdash S_1 : \Gamma, rc \longrightarrow \Gamma', rc' \qquad \Phi, pc, rc \vdash S_1 \longrightarrow S_1' \qquad \Phi, pc, rc' \vdash S_2 \rightsquigarrow S_2'}{\Phi, pc, rc \vdash S_1; S_2 \longrightarrow S_1'; S_2'}$$

$$\textsc{Tr-Ret-For}\quad \frac{\Phi = (\omega, \Gamma, \beta_r) \qquad rc \sqsubseteq rc' \qquad \omega, pc, \beta_r \vdash S : \Gamma, rc' \longrightarrow \Gamma', rc' \qquad \Phi, pc, rc' \vdash S \rightsquigarrow S'}{\begin{array}{c}\Phi, pc, rc \vdash \texttt{for}\ (x\ \texttt{from}\ e_1\ \texttt{to}\ e_2)\ \{\ S\ \} \longrightarrow \\ \texttt{for}\ (x\ \texttt{from}\ e_1\ \texttt{to}\ e_2)\ \{\ S'\ \}\end{array}}$$

Figure 6. Return deferral transformation rules.

4.1 Return deferral

As previously mentioned, early returns that depend on secrets often leak information. Consider the following snippet:

```
if (sec) { return 1; }
// long-running computation ...
```

Here, an attacker can determine whether `sec` is `true` by observing a quick computation, or `false` by observing a slow computation.

FaCT prevents code from leaking such secrets by *deferring* `return`s to the end of each procedure. For example, the compiler transforms the above code to:

```
secret mut uint32 rval = 0;
secret mut bool notRet = true;
if (sec) { rval = 1; notRet = false; }
if (notRet) {
  // long-running computation ...
}
return rval;
```

The new `notRet` variable tracks whether or not the procedure *would have* returned, and any statement that could be executed after the `return` is guarded by the `notRet` variable.

Finally, the actual `return` occurs at the very end of each procedure, returning the value stored in `rval`.

Transformation rules. We formalize return deferral using three kinds of rewrite rules, shown in Figure 6. The first *procedure-transformation* rule $\omega \vdash f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r \longrightarrow f(\vec{x} : \vec{\beta})\ \{\ S'\ \} : \beta_r$ is used to rewrite the body S of a procedure f into a `secret`-independent body S'. (This is accomplished using the other two rewrite rules.) The second *guarded-execution* rule $\Phi, pc, rc \vdash S \rightsquigarrow S'$ transforms a statement S, given a secrecy context pc, rc, into S' by making implicit control flow (due to secret returns) explicit. Finally, the *return-elimination* rule $\Phi, pc, rc \vdash S \longrightarrow S'$ transforms S into S' by replacing all secret returns with assignments. Below, we walk though some of these rules in detail.

1. Procedure transformation. The Tr-Ret-Dec rule declares two special (mutable) variables `notRet` and `rval` that respectively hold the `secret`-dependent return state and the value to be returned. The return state `notRet` is set to `true`, while the return value `rval` is initialized to a default value for its type. The rule then eliminates all secret returns from S and inserts a (deferred) `return` after, as the very last statement of the transformed body S'.

2. Guarded execution. Rules Tr-Ret-Guard-Pub and Tr-Ret-Guard-Sec are used to transform statements that appear *after* any secret returns. Both of these rules first eliminate secret returns from S to obtain S'. If the original statement S is typed with $rc = \text{Sec}$, i.e., S may be preceded by a secret return, then the rule Tr-Ret-Guard-Sec additionally *guards* the execution of S' with the condition `notRet`.

3. Return elimination. The bulk of the transformation is done by the remaining rules in Figure 6. We omit rules where we either do not transform the statement, or simply recursively transform any sub-statements. Rule Tr-Ret replaces secret returns by updating `rval` with the (deferred) return value and setting `notRet` to `false`, to signal that subsequent code should *not* be executed.

Rule Tr-Ret-Seq handles sequenced statements $S_1; S_2$ by guarding the execution of instructions in S_2 against possible `secret` returns in S_1. The rule first eliminates the secret returns from the *first* block to get S_1'. Next, it extracts the secrecy context rc' produced by type checking S_1. Finally, the rule uses rc' to derive a guarded version of the *second* statement S_2'.

The Tr-Ret-For rule handles secret returns inside loops. As control flow can jump back to the beginning of a loop, a secret return inside a loop body S can affect the execution of the entire body, as in the following example:

```
for (uint32 i from 0 to 5) {
  b[i] = 1;
  if (i == sec) { return i; }
  a[i] = 2;
}
```

Tr-Br-Dec

$$\frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \qquad \omega(f) = \text{Pub} \qquad \Phi, \texttt{true} \vdash S \longrightarrow S'}{\omega \vdash f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r \ \longrightarrow\ f(\vec{x} : \vec{\beta})\ \{\ S'\ \} : \beta_r}$$

Tr-Br-Dec-Sec

$$\frac{\Phi = (\omega, \{\vec{x} : \vec{\beta}\}, \beta_r) \qquad \omega(f) = \text{Sec} \qquad \Phi, \mathit{callCtx} \vdash S \longrightarrow S'}{\omega \vdash f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r \ \longrightarrow\ f(\vec{x} : \vec{\beta}, \mathit{callCtx} : \text{Bool}_{\text{Sec}})\ \{\ S'\ \} : \beta_r}$$

Tr-Br-If

$$\frac{\Phi = (\omega, \Gamma, \beta_r) \qquad \Gamma \vdash e : \text{Bool}_{\text{Sec}} \qquad \text{FRESH } m_t,\ m_f \qquad \Phi, (p \,\&\, m_t) \vdash S_1 \longrightarrow S_1' \qquad \Phi, (p \,\&\, m_f) \vdash S_2 \longrightarrow S_2'}{\begin{array}{l}\Phi, p \vdash \texttt{if}\ (e)\ \{\ S_1\ \}\ \texttt{else}\ \{\ S_2\ \} \ \longrightarrow \\ \quad \{\ \ \text{Bool}_{\text{Sec}}\ m_t = e; \\ \quad\ \ \text{Bool}_{\text{Sec}}\ m_f = \neg m_t; \\ \quad\ \ S_1';\ \ S_2' \qquad\qquad\ \ \}\end{array}}$$

Tr-Br-Assign

$$\frac{p \neq \texttt{true}}{\Phi, p \vdash e_1 := e_2 \ \longrightarrow\ e_1 := \texttt{ctselect}(p, e_2, e_1)}$$

Tr-Br-Call

$$\frac{\omega(f) = \text{Sec}}{\Phi, p \vdash \beta\ x = f(\vec{e}) \ \longrightarrow\ \beta\ x = f(\vec{e}, p)}$$

Figure 7. Rules for branch removal.

Here, if `i == sec` becomes `true`, the program must stop overwriting the elements in both `a` *and* `b`. The rule accounts for `return`s in the body S by using the secrecy context rc' from type checking the body, and in turn, uses this to derive the guarded form of the body S'. In our example, the `secret`-dependent `return` makes the return context $rc' = \text{Sec}$, and so the entire body is guarded by `notRet`, to obtain the transformed program:

```
for (uint32 i from 0 to 5) {
  // for-loop rule
  if (notRet) {
    b[i] = 1;
    if (i == sec) { rval = i; notRet = false; }
    // sequencing rule
    if (notRet) { a[i] = 2; }
  }
}
```

4.2 Branch removal

Return deferral eliminates `secret` returns by introducing `secret`-dependent branches. In this section we eliminate `secret`-dependent control flow as the final step towards producing constant-time code.

To this end, FaCT replaces secret branches with constant-time selections. Consider the following snippet:

```
if      (sec1) { a[1] = 3; }
else if (sec2) { a[2] = 4; }
```

The updates to `a[1]` and `a[2]` are guarded by the `secret` values `sec1` and `sec2` and, therefore, produce memory access patterns that can reveal the values of those secrets when left untransformed—this is the classic *implicit flows* problem [55]. We eliminate the implicit flow in two steps. First, we track the *control predicates* that correspond to (the conjunction of) the `secret`-conditions. Then, we perform both memory writes, but use `ctselect` to preserve conditional semantics:

```
a[1] = ctselect( sec1       , 3, a[1]);
a[2] = ctselect(~sec1 & sec2, 4, a[2]);
```

Our general strategy is to transform each conditional array assignment into a re-assignment to a conditional (`ctselect`).

Transforming code that calls procedures is less straightforward: if a procedure takes a mutable parameter, the procedure may update that parameter's value in a way that is visible to the caller. For example:

```
void foo(secret mut uint32 x) { x = 5; }
...
if (sec) {
  foo(x);
  // x is now 5
}
```

The transformation of this code must ensure that updates to `x` only occur if `sec` is `true`. We do so using a *call-context* parameter passed to callee `foo`; this parameter is the caller control predicate—in this case, `sec`—which we use to guard updates in `foo`. Our compiler converts the above into semantically equivalent constant-time code:

```
void foo(secret mut uint32 x,
         secret bool callCtx) {
  x = ctselect(callCtx, 5, x);
}
...
foo(x, sec);
// x is 5 only if sec is true
```

Transformation rules. We formalize branch removal using two kinds of rules, shown in Figure 7. The *procedure transformation* rule $\omega \vdash f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r \longrightarrow f(\vec{x}' : \vec{\beta}')\ \{\ S'\ \} : \beta_r$ transforms the body S of the procedure f to S', much like for `secret`-return removals. This rule, however, additionally extends f's set of parameters $\vec{x}$ to include the extra *call-context* parameter *callCtx*. The *statement transformation* rule $\Phi, p \vdash S \longrightarrow S'$, transforms S to S' given context Φ and control predicate p. We walk though some of the rules below.

1. Procedure transformation rule. Both Tr-Br-Dec and Tr-Br-Dec-Sec remove branches from procedures. Tr-Br-Dec transforms procedures that do not depend on `secret` contexts by transforming each procedure's body S into S' using the initial control predicate `true`. Tr-Br-Dec-Sec, on the other hand, transforms a procedure f if $\omega(f) = \text{Sec}$, i.e., where f depends on the caller's `secret` context. The rule adds a new parameter `secret bool callCtx` that holds the control predicate at each call-site, and then transforms the body S starting with the initial control predicate `callCtx`.

2. Branch elimination. The remaining rules in Figure 7 remove branches from statements. Rule Tr-Br-If, for example, eliminates `secret`-dependent conditional branches by saving the condition (resp. its negation) in the variable m_t (resp. m_f). The "then" statement S_1 (resp. "else" statement S_2) is then transformed after conjoining m_t (resp. m_f) to the control predicate p. To prevent name collision when transforming nested conditionals, the FRESH metafunction guarantees that all m_t and m_f variables have unique names. The declarations of m_t, m_f and transformed branches S_1', S_2' are sequenced to obtain the final result.

Rule Tr-Br-Assign handles side-effecting assignment statements, using the control predicate to `ctselect` the old or new values. But, if the assignment occurs under the trivial control predicate (i.e., the literal `true`), the assignment is left unchanged.

Finally, rule Tr-Br-Call handles calls to ω-Sec procedures f by explicitly passing the control predicate p as the call-context parameter. This ensures that updates within f only occur according to the caller's control flow.

4.3 Compiler correctness and security

In this section, we prove that our compiler preserves semantics and outputs constant-time procedures. To formalize these claims, we define an instrumented semantics that describes procedure behavior and *leakage*, i.e., the sequence of branches taken, the memory addresses accessed, and the operands to variable-time instructions. Intuitively, a procedure is constant-time if its leakage is not influenced by any secret values [9].

In particular, we consider a big-step semantics of the form $F : (\vec{v}, h) \xrightarrow{\psi} (v, h')$ where F is shorthand for a procedure $f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta_r$, the term $\vec{v}$ represents the values of parameters, h and h' are *heaps* mapping pointers to values, v is the final value of the procedure, and ψ is the leakage. The semantics is parametrized by an allocation function, and the proofs of the claims below rely on several (minor) assumptions on this function. We give these assumptions, formal definition, and complete proofs in [25].

We first prove the *correctness* of our compiler, using the notation $\omega \vdash F \longrightarrow F'$ to denote the combined return deferral and branch removal transformations. Compiler correctness states that the compiler preserves the meaning of well-typed statements. To account for new references and variables that are introduced by the compiler pass itself, we show *equivalence* of the final heaps h' and h'', i.e., for any pointer p' in h', there is an equivalent pointer p'' in h'' such that $h'(p')$ and $h''(p'')$ are either equal values, or are themselves equivalent pointers.

Theorem 4.1 (Compiler correctness). *If $\omega \vdash F \longrightarrow F'$ and F' is well-typed, then $F : (\vec{v}, h) \xrightarrow{\psi} (v, h')$ implies that $F' : (\vec{v}, h) \xrightarrow{\psi'} (v, h'')$ and h' and h'' are equivalent.*

Proof sketch. By induction on the derivation. □

Note that our compiler correctness theorem does not make any claim about leakage. We separately prove that the compiler produces constant-time procedures. To this end, we first define the notion of a constant-time procedure.

Definition 4.2. A procedure F where $\omega \vdash F$ is *constant-time* iff for every pair of executions $F : (\vec{v}_1, h_1) \xrightarrow{\psi_1} (v_1, h_1')$ and $F : (\vec{v}_2, h_2) \xrightarrow{\psi_2} (v_2, h_2')$, we have $\vec{v}_1, h_1 \equiv \vec{v}_2, h_2$ implies $\psi_1 = \psi_2$, where $\equiv$ is a suitably parametrized notion of equivalence (e.g., `public` or "low" equivalence [5, 9, 62]).

Much like CT-Wasm [62], we cannot prove that *all* FaCT procedures are constant-time—FaCT allows procedures to declassify secret data and call external procedures over which it has no control. We can, however, provide guarantees for a safe subset of *declassify-free* procedures, i.e., procedures that do not contain any `declassify` statements nor call other procedures unless they too are declassify-free (and not `extern`).

Theorem 4.3 (Compiler security). *If F is declassify-free and $\omega \vdash F \longrightarrow F'$, then F' is constant-time.*

Proof sketch. We define two additional type systems that impose stricter constraints on programs, and prove type-preservation for return deferral and branch removal. We then conclude by proving that the final type system guarantees that programs are constant-time. It is important to note that these type systems are merely proof artifacts, i.e., type checking is not performed again after transformations.

Informally, the two type systems are incremental restrictions on the FaCT type system. The first type system, which we denote by $\vdash_{\text{rd}}$, rejects programs that contain secret returns; the second type system, denoted $\vdash_{\text{ct}}$, rejects programs that branch on secrets.

We then establish type-preservation for return deferral and branch removal:

- If $\omega \vdash F$ and $\omega \vdash_{\text{rd}} F \longrightarrow F'$ then $\omega \vdash_{\text{rd}} F'$.
- If $\omega \vdash_{\text{rd}} F$ and $\omega \vdash_{\text{ct}} F \longrightarrow F'$ then $\omega \vdash_{\text{ct}} F'$.

Both are proved by induction on derivations, using adequate ancillary statements for the induction to go through.

We conclude by proving that $\vdash_{\text{ct}}$ guarantees that programs are constant-time. The proof follows from a "locally preserves" unwinding lemma, stating that equivalent states yield equivalent final configurations and equal leakage. □

5 Implementation and evaluation

We implement a prototype compiler for FaCT in ~6000 lines of OCaml. The compiler transforms FaCT source code into LLVM IR, which it passes to `clang` (version 6.0.1) to generate assembly or object code. The compiler uses the Z3 SMT solver [29] to check public safety assertions (§3.2.3).

We evaluate FaCT by asking the following questions:

- Is FaCT expressive enough to implement real-world cryptographic algorithms?
- Does FaCT produce constant-time code?
- What is FaCT's performance overhead?
- Compared to C, does FaCT improve non-experts' experience reading and writing constant-time code?

We answer the first three questions with case studies in which we integrate FaCT into real-world projects (§5.1). We find that FaCT is expressive enough to implement a range of cryptographic primitives. We use dudect [52] to empirically check that our implementations, including compiler optimizations, are constant-time. We find that, compared to optimized C code, unoptimized FaCT code runs 16–346% more slowly, while optimized FaCT code ranges from 5% slower to 21% faster.

We answer the fourth question with a study comparing user experiences reading and writing FaCT and C (§5.2). In sum, a plurality of participants found FaCT easier to read than C, and a majority found FaCT easier to write.

5.1 Case studies

We integrate FaCT into three popular open source libraries by porting pieces of these libraries from C to FaCT:

- The NaCl `secretbox` API for symmetric-key authenticated encryption and decryption. We port the entire libsodium (version 1.0.16) [30] `secretbox` API, including the two underlying primitives, the Poly1305 message authentication code (MAC) and the XSalsa20 stream cipher.
- The Curve25519 Elliptic-Curve Diffie-Hellman (ECDH) primitive for asymmetric key exchange. We port Adam Langley's curve25519-donna library [36] in whole.
- The OpenSSL [45] `ssl3_cbc_digest_record` function used to verify decrypted SSLv3 messages. At its core, this function computes the MAC of a padded message without revealing the padding length. Our implementation invokes OpenSSL's SHA-1 hash primitive as an `extern` (§3.1).
- The OpenSSL `aesni_cbc_hmac_sha1_cipher` function used in the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) construction. This function performs AES-CBC decryption and then verifies the MAC and padding of the decrypted message. Our implementation invokes OpenSSL's AES and SHA-1 primitives as `extern`s.

We choose these functions because they (1) are complex enough to exercise all of the FaCT language features; (2) implement a range of algorithms; and (3) demonstrate that FaCT can be used in different settings, from implementing individual procedures to large portions of libraries.

Method. We port in three steps. First, we port the C code to FaCT by translating C constructs to their corresponding FaCT counterparts. During this translation process, we label sensitive messages, keys, etc. as `secret`, and add `assume` and

Table 1. Case study summary: lines of code (per `cloc`) and uses of `assume` (#A), `declassify` (#D), and `extern` (#E).

Case study	Lines of code C	Lines of code FaCT	#A	#D	#E
libsodium `secretbox`	984	1068	16	1	0
curve25519-donna	310	342	0	0	0
OpenSSL record validate	92	91	3	0	2
OpenSSL MEE-CBC	201	219	10	1	4

`declassify` statements as appropriate to ensure the code typechecks (§5.1.1); we also replace "bit hacks" (§2) with high-level FaCT constructs (e.g., `if`). Second, we check the correctness of our ports using each library's test harness, and we empirically check that the ports are constant-time using dudect (§5.1.2). Finally, we use each library's benchmarking suite to compare our ports to the C implementations (§5.1.3).

5.1.1 Expressiveness

Table 1 summarizes our ports. FaCT implementations are at worst ~10% longer than the corresponding C code. Much of the extra length is because FaCT does not have a macro system; instead, we translated macro definitions and then manually expanded them. (We note that it would be straightforward to instead use the C preprocessor with FaCT.) FaCT code is also more verbose than C when processing buffers: since FaCT has no pointer arithmetic, FaCT code must use extra variables to track offsets into arrays.

Our ports make sparing use of `extern`, `declassify`, and `assume`. For example, our ports use `assume` to help the public safety verifier track values through memory and reason across procedure and language boundaries. We `declassify` in two cases: in libsodium `secretbox` decryption and in OpenSSL MEE-CBC verification; these declassifications are permitted by the libraries' respective attacker models [19, 27, 37]. Finally, we use `extern` to invoke existing primitives (e.g., OpenSSL's SHA-1 implementation).

5.1.2 Security

We prove that FaCT's transformations produce constant-time code (§4.3), but this applies only to the *unoptimized* LLVM IR produced by the FaCT compiler.[3] Since we use `clang` to generate optimized object code, an LLVM optimization pass might break FaCT's constant-time guarantees.

To empirically check that our case study implementations run in constant-time, even after optimization, we use the dudect [52] analysis tool. At a high level, dudect tests for constant-time execution by running the code under test for a large number of iterations and collecting timing information using the CPU's cycle counters. It then tests the collected timing information for statistically significant variation in execution time that are correlated with changes to secret inputs. In our evaluation, we configure dudect to collect 50 million measurements for each benchmark. It finds no statistically significant timing variation.

Several other works concerned with constant-time crypto implementation [8, 52, 56, 62] have reported using dudect. In our testing, we found the tool to quickly and reliably find timing differences in buggy code. We note, however, that dudect is only a check—not a proof—of constant-time behavior; we discuss further in Section 6.

Table 2. Overhead of FaCT ports compared to optimized C, for each benchmark. `secretbox` results are for encryption and decryption overhead, respectively.

Benchmark	% Overhead of FaCT Unoptimized	% Overhead of FaCT Optimized
`secretbox` (reference)	345.57/373.49%	-20.92/-14.56%
`secretbox` (vectorized)	427.21/427.09%	-6.54/-4.99%
curve25519-donna	144.42%	2.21%
OpenSSL record validate	30.13–35.16%	0.64–4.62%
OpenSSL MEE-CBC	16.15–31.97%	-2.56–4.16%

5.1.3 Performance

Table 2 shows the performance cost of porting C to FaCT. We benchmark each implementation on an Intel i7-6700K at 4GHz with 64GB of RAM using `clang` 6.0.1. We compare both unoptimized and optimized FaCT implementations with C implementations that are compiled at the corresponding project's default optimization level.[4] Our optimized FaCT code uses the same optimization flags as the C code.

For libsodium and curve25519-donna, we use the library's benchmarking suites. We measure the mean of $\sim 2^{24}$ and $\sim 2^{17}$ iterations, respectively, and report the median of five such measurements. For the OpenSSL implementations, we use OpenSSL's `s_server` and `s_client` commands to measure throughput when transferring 256MB, 1GB, and 4GB files. We compute the median throughput of five transfers at each file size, and report the minimum and maximum result; overhead was uncorrelated with file size.

For most benchmarks, we find that optimized FaCT is comparable to C: the overhead is never more than 5%. Notably, the FaCT implementation of libsodium `secretbox` is 15-20% *faster* than the C reference implementation. We attribute this speedup to vectorization: inspecting the XSalsa20 assembly code, we find that `clang` generates vector instructions for the FaCT implementation, but not for C. To explore this discrepancy, we measure performance of `secretbox` with XSalsa20 explicitly vectorized (using vectors in FaCT, intrinsics in C). In this case, FaCT is still 5-6% faster than C, but this speedup appears to be an artifact of LLVM's applying different optimizations to different code.

[3] And to procedures that do not use `declassify`.

[4] For OpenSSL, `-O3`; for other projects, `-O2`.

5.2 User study

We evaluate the usability of FaCT by conducting a user study as part of an upper-level, undergraduate programming languages course at UC San Diego.[5] Prior to the study, we dedicated three lectures to timing side-channels, constant-time programming in general, and constant-time programming specifically in C and FaCT. As an optional assignment, students were asked to (1) *explain* the behavior of constant-time code written in C and FaCT, and (2) *implement* constant-time algorithms in both C and FaCT. Of the 129 enrolled students, 77 completed the study over a nine-day period. We describe methods and conclusions below; in [25], we give further lessons from the study, e.g., compilation errors participants ran into frequently.

Method. The user study is a sequence of web-based tasks. For each task, the participant is first given a warm-up code comprehension question, whose answer is subsequently revealed. The participant is then given a second, related question. This question is repeated twice, in C and in FaCT; we randomize the order of the languages per participant, i.e., half the participants' tasks are in C and then FaCT, and vice-versa. On a given question, participants can repeatedly check partial answers for correctness; once finished, the participant *submits* a final answer, which can no longer be viewed or revised. A task is *complete* if the participant submits a final answer for both C and FaCT; we discard incomplete tasks.

The user study was built on an earlier version of FaCT which did not enforce public memory safety. Nevertheless, we believe the results largely translate to the version presented in this paper, because the surface language did not change significantly.

5.2.1 Understanding constant-time code

To evaluate participants' understanding of C and FaCT code, we asked them to describe the behavior of two functions. The first function takes two input buffers—a header and a message—and copies the header and message to an output buffer and adds padding up to a fixed size. The second function implements long division: it computes a quotient and remainder, writes each to an output buffer, and returns a status code indicating success or failure.

We graded participants on their ability to correctly describe each function's behavior. In both cases, we find that participants showed slightly better understanding of FaCT than of C: for the first function, the mean score was 57% for FaCT and 53% for C; for the second, it was 40% for FaCT and 32% for C. Participants also reported a slight preference for FaCT; specifically, 31 participants found FaCT easier to understand compared to 10 that found C easier and 28 that reported similar difficulty.

[5] Our study was reviewed and exempted by the IRB.

Table 3. Number of participants (out of 77) that submitted correct and constant-time solution for each task. The `check_pkcs7_padding` task was misconfigured, and marked variable-time code as constant-time (16 submissions); we report these numbers for completeness (§5.2.2).

Programming task	FaCT	C
`remove_secret_padding`	62	49
`check_pkcs7_padding`	35	32 (16)
`remove_pkcs7_padding`	34	24

5.2.2 Writing constant-time code

To evaluate participants' ability to write constant-time code in FaCT and C, we had them implement three functions:

- `remove_secret_padding`: given a buffer and secret length, this function removes any secret padding, i.e., sets every element of the buffer past the length to zero.
- `check_pkcs7_padding`: this function checks whether a supplied buffer contains a valid PKCS#7 [34] message.
- `remove_pkcs7_padding`: this function removes padding from a supplied buffer, if it contains a valid message.

Participants could compile their code, run a test suite, and, for C code, check constant-time correctness with ct-verif [5]. They could also give up on a task and move to the next one.

Table 3 summarizes our findings. Of the 68 participants that completed the first task, 62 submitted correct and constant-time FaCT code, and 49 submitted correct and constant-time C code. For the third task, 34 participants submitted correct, constant-time FaCT code compared to 24 participants for C. In the survey, 40 participants reported finding FaCT easier to write, 11 found C easier, and 18 found them similar.

We cannot draw conclusions from `check_pkcs7_padding`, because the task had a bug that incorrectly marked variable-time code as constant-time; only 16 of the 32 C submissions marked "correct" were constant-time. The bug was limited to this task, but because `check_pkcs7_padding` is required for `remove_pkcs7_padding`, some participants needed to correct their code to pass the third task.

6 Limitations and future work

We think FaCT makes it easier to write constant-time code, but it is not perfect. Limitations and future work include:

The type system. The type system lacks polymorphism and flow sensitivity [43, 55], which reduces both expressivity and performance. For example, our type system cannot express a program that branches on a buffer's `public` contents and then decrypts the buffer in-place, upgrading its label to `secret`. We leave such extensions to future work.

The public safety checker. FaCT's public safety checker does not reason about mutable variables or properties across function calls. For example, indexing an array based on a mutable variable requires `assume`-ing the index is in bounds.

The brittleness of constant-time behavior. FaCT's compiler only guarantees constant-time behavior for the LLVM IR that it produces. Crucially, this means that LLVM's optimization passes and lowering to assembly can introduce variable-time behavior. Though many optimizations *do* preserve constant-time property [10], FaCT relies on dudect to empirically check that a piece of code is constant-time.

Sound, *symbolic* verification of constant-time behavior using ct-verif [5] would give much stronger guarantees. Unfortunately, ct-verif currently has limited support for declassification and vector instructions. Extending ct-verif to support these primitives and applying it to optimized FaCT code is future work.

The evaluation. Our evaluation of FaCT is preliminary and thus incomplete. For example, we relied on `extern` versions of SHA-1 and AES (§5.1) because we preferred to focus on porting higher-level OpenSSL functions with a history of timing attacks. Moreover, some of the low-level primitives we ported (XSalsa20, Poly1305, and Curve25519) were explicitly designed for ease of constant-time implementation [14, 15, 17]. Future work is expanding FaCT's repertoire with potentially more challenging algorithms.

Finally, our user study has limited scope and involves only non-expert users; remedying these issues is also future work.

7 Related work

This work supersedes an initial design we previously described in [24]. In particular, we present a design and implementation of a DSL for writing constant-time crypto, provide a formal semantics and security guarantees for FaCT, and evaluate FaCT on several dimensions; in [24] we outlined the vision for such a DSL. Our implementation and formalization efforts revealed insights previously missed in [24]—e.g., the need for *public safety* (§3.2.3) and challenges with using ct-verif [5] to verify code with inline declassifications. At the same time, in this paper, we did not explore parts of the design space outlined in [24]—e.g., we do not expose some hardware-specific instructions like add-with-carry, which could simplify asymmetric-key crypto implementations.

Domain-specific languages. There are several efforts designing DSLs for implementing cryptographic primitives and protocols. Bernstein's qhasm is a low-level portable assembly for writing high-speed crypto routines [16]; it does not distinguish `secret` data from `public` data, so does not prevent information leaks by construction.

Vale [21] and Jasmin [3] are DSLs for writing and verifying high-performance assembly code. Vale developers write platform-independent assembly code and specify the target architecture; the Vale compiler uses Dafny to verify semantics and non-interference. Jasmin allows developers to use architecture-specific instructions alongside higher-level code, and the verified Jasmin compiler rejects non-constant-time programs. Low* is a higher-level, embedded (in F*) DSL that compiles to verified constant-time C [50]. The verified NaCl [18] library, HACL* [68], is written in Low*. CT-Wasm [62] is a formally verified extension to the WebAssembly language [63] for writing crypto code in the browser. CT-Wasm uses a strict label-based type system to enforce its constant-time policy. These languages provide support for high-level control flow constructs and procedures, but they require developers to manually write constant-time code.

Constant-Time Toolkit (CTTK) is a C library [48] that follows recipes in [28, 49] to provide functions—including low-level constant-time primitives—for crypto libraries, but developers must compose these low-level blocks.

Verification. There is a growing body of work on both building verified cryptographic implementations and verifying existing libraries. Bhargavan et. al verify an implementation of TLS, including low-level cryptographic primitives [20]. Barthe et. al [9] verify constant-time properties of various PolarSSL implementations. Ye et. al [65] verify the mbedTLS implementation of HMAC-DRBG. Appel [7] and Beringer et. al [12] respectively verify OpenSSL's implementation of SHA-256 and HMAC. Tsai et. al [60] verify core parts of X25519. Almeida et. al [4] verify AWS Lab's s2n MEE-CBC implementation (after identifying a vulnerability); they also verify security properties of NaCl libraries [6]. Erbsen et. al [32] synthesize and verify elliptic curve implementations from high-level descriptions. Almeida et. al develop ct-verif [5] and verify constant-time properties of several cryptographic algorithms. Many of these verification efforts are specific to the projects being analyzed. Additionally, developers still bear the burden of manually writing constant-time code, which FaCT aims to alleviate.

General techniques for eliminating timing-channels. FaCT uses an information flow control type system to eliminate programs that may introduce information leaks or are otherwise inefficient (or impossible) to transform to constant-time. Our label-based type system is a standard IFC type system [55] that borrows explicit mutability from ownership-based systems [26]. Previous solutions have also relied on type- and static-analysis techniques (e.g., [9, 31, 54, 59, 66]) to address timing leaks. FaCT automatically transforms secret sub-computations into constant-time straight-line code. Our approach follows several previous efforts on eliminating timing channels via source code transformations [1, 11, 41, 44, 47, 51]. Most similar in ethos is SC-Eliminator [64]. This system takes as input a program and a list of secrets, and uses tag propagation to transform LLVM IR into its constant-time equivalent. Though both projects perform transformations, they use orthogonal approaches: SC-Eliminator repairs already-existing code, while FaCT is a language for writing such code from the start. Finally, many other efforts employ system-level techniques to eliminate and detect timing-channels [22, 33, 38, 52, 58, 67].

Acknowledgments

We thank the anonymous PLDI and PLDI AEC reviewers and our shepherd Limin Jia for their suggestions and insightful comments. We thank the participants of the Dagstuhl Seminar on Secure Compilation for early feedback on this work, especially Tamara Rezk. We thank Ariana Mirian for handling the IRB for our user study, Shravan Narayan for his help in understanding the subtleties of LLVM, and Joseph Jaeger and Jess Sorrell for helping us understand elliptic curve implementations. We also thank the CSE 130 TAs for their help in testing our user study, and the CSE 130 students for participating in the user study. This work was supported in part by gifts from Fujitsu and Cisco, by the National Science Foundation under Grant Number CNS-1514435, by ONR Grant N000141512750, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Johan Agat. 2000. Transforming out timing leaks. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

[2] Nadhem J. Al Fardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *34th IEEE Symposium on Security and Privacy*. IEEE.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption*. Springer.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th USENIX Security Symposium*. USENIX Association.

[6] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming* (2013).

[7] Andrew W. Appel. 2015. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems* (2015).

[8] Jean-Philippe Aumasson and Yolan Romailler. 2017. Automated testing of crypto software using differential fuzzing. *Black Hat USA* (2017).

[9] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[10] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *Computer Security Foundations Symposium*.

[11] Gilles Barthe, Tamara Rezk, and Martijn Warnier. 2006. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science* (2006).

[12] Lennart Beringer, Adam Petcher, Q. Ye Katherine, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*.

[13] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES*. Technical Report. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[14] Daniel J. Bernstein. 2005. The Poly1305-AES message-authentication code. In *Fast Software Encryption*. IACR.

[15] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*. Springer.

[16] Daniel J. Bernstein. 2007. qhasm: tools to help write high-speed software. https://cr.yp.to/qhasm.html

[17] Daniel J. Bernstein. 2008. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*. Springer.

[18] Daniel J. Bernstein. 2009. *Cryptography in NaCl*. Technical Report. http://cr.yp.to/highspeed/naclcrypto-20090310.pdf.

[19] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer.

[20] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*. IEEE.

[21] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium*. USENIX Association.

[22] Benjamin A. Braun, Suman Jana, and Dan Boneh. 2015. Robust and efficient elimination of cache and timing side channels. (2015). arXiv:1506.00189

[23] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* (2005).

[24] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *Secure Development Conference (SecDev)*. IEEE.

[25] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. *FaCT: A DSL for Timing-Sensitive Computation*. Technical Report. https://fact.programming.systems/FaCT_extended.pdf

[26] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 48–64.

[27] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*. IEEE.

[28] Cryptography Coding Standard. 2016. Coding Rules. Retrieved June 9, 2017 from https://cryptocoding.net/index.php/Coding_rules

[29] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[30] Frank Denis. [n. d.]. libsodium. Retrieved November 17, 2018 from https://github.com/jedisct1/libsodium

[31] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security* (2015).

[32] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2018. *Systematic Generation of Fast Elliptic Curve Cryptography Implementations*. Technical Report. https://people.csail.mit.edu/jgross/personal-website/papers/2018-fiat-crypto-pldi-draft.pdf

[33] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2018).

[34] Burt Kaliski. 1998. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315.

[35] Paul Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*. Springer.

[36] Adam Langley. [n. d.]. curve25519-donna. Retrieved November 17, 2018 from https://github.com/agl/curve25519-donna

[37] Adam Langley. 2013. ImperialViolet - Lucky Thirteen attack on TLS CBC. Retrieved November 13, 2018 from https://www.imperialviolet.org/2013/02/04/luckythirteen.html

[38] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory trace oblivious program execution. In *IEEE 26th Computer Security Foundations Symposium*. IEEE.

[39] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. 2012. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF)*. IEEE.

[40] Bodo Moeller. 2004. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. https://www.openssl.org/~bodo/tls-cbc.txt

[41] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology*. Springer.

[42] Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

[43] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif: Java information flow. Retrieved November 15, 2018 from http://www.cs.cornell.edu/jif

[44] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and synthesizing constant-resource implementations with types. In *38th IEEE Symposium on Security and Privacy*. IEEE.

[45] The OpenSSL Project. [n. d.]. OpenSSL. Retrieved November 17, 2018 from https://github.com/openssl/openssl

[46] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer.

[47] Jérémy Planul and John C. Mitchell. 2013. Oblivious program execution and path-sensitive non-interference. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*. IEEE.

[48] Thomas Pornin. [n. d.]. Constant-Time Toolkit. Retrieved November 15, 2018 from https://github.com/pornin/CTTK

[49] Thomas Pornin. 2016. Why Constant-Time Crypto? Retrieved November 15, 2018 from https://www.bearssl.org/constanttime.html

[50] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *Proceedings of the ACM on Programming Languages* (2017).

[51] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution.. In *24th USENIX Security Symposium*. USENIX Association.

[52] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is my code constant time?. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM.

[54] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *25th International Conference on Compiler Construction*. ACM.

[55] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* (2003).

[56] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *3rd IEEE European Symposium on Security and Privacy*. IEEE.

[57] Juraj Somorovsky. 2016. Curious Padding oracle in OpenSSL (CVE-2016-2107). Retrieved November 15, 2018 from https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html

[58] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. 2013. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*. Springer.

[59] Josef Svenningsson and David Sands. 2009. Specification and verification of side channel declassification. In *International Workshop on Formal Aspects in Security and Trust*. Springer.

[60] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[61] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS.... In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer.

[62] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proceedings of the ACM on Programming Languages* (2019).

[63] WebAssembly Community Group. 2018. WebAssembly. Retrieved November 15, 2018 from http://webassembly.org

[64] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-channel Leaks Using Program Repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 12.

[65] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[66] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. *ACM SIGPLAN Notices* (2015).

[67] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[68] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

A FaCT Grammar

PROGRAM

program ::= $\left[\ fdef \mid sdef\ \right]\ldots$

STRUCTURE DEFINITION

sdef ::= `struct` *name* $\{\ \beta\ x; \ldots\ \}$

PROCEDURE DEFINITIONS

fdef ::=

|	$f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta$	internal procedure
|	`export` $f(\vec{x} : \vec{\beta})\ \{\ S\ \} : \beta$	exported procedure
|	`extern` $f(\vec{x} : \vec{\beta}) : \beta$	external procedure

STATEMENTS

S ::=

|	$\{\ S\ \}$	block
|	$S; S$	sequence
|	$\beta\ x = e$	variable declaration
|	$\beta\ x = f(\vec{e})$	procedure call
|	$f(\vec{e})$	void procedure call
|	$e := e$	assignment
|	$e \oplus= e$	binop assignment
|	`if` $(e)\ \{\ S\ \}$ $\left[\ \texttt{else if}\ (e)\ \{\ S\ \}\ \right]\ldots$ $\left[\ \texttt{else}\ \{\ S\ \}\right]$	conditional
|	`for` $(x$ `from` e `to` $e)\ \{\ S\ \}$	range-for
|	`return` e | `return`	return

UNARY OPS

$\ominus$ ::=

|	`!`	boolean not
|	`-`	negate
|	`~`	bitwise not

BINARY OPS

$\oplus$ ::=

|	`+` | `-` | `*` | `/` | `%`	arithmetic
|	`==` | `!=`	equality
|	`<` | `<=` | `>` | `>=`	comparison
|	`&&` | `||`	logical
|	`&` | `|` | `^`	bitwise
|	`<<` | `>>`	bitshift
|	`<<<` | `>>>`	bit rotate

EXPRESSIONS

e ::=

|	(e)	parentheses
|	`true` | `false`	boolean literal
|	n	numeric literal
|	x	variable
|	$\ominus\ e$	unary op
|	$e \oplus e$	binary op
|	$e\ ?\ e : e$	ternary op
|	`ctselect`(e, e, e)	constant-time selection
|	$\text{UINT}^s(e)$ | $\text{INT}^s(e)$	numeric cast
|	$[e, \ldots]$	array literal
|	$e[e]$	array get
|	`len` e	array length
|	`zeros`(β, e)	zero array
|	`clone`(e)	array clone
|	`view`(e, e, e)	array view
|	`ref` e	reference
|	`deref` e	dereference
|	$\langle n, \ldots\rangle$	vector literal
|	$e\langle n, \ldots\rangle$	vector select/shuffle
|	$\{x : e, \ldots\}$	struct literal
|	$e.x$	struct access
|	$f(\vec{e})$	procedure expression
|	`declassify`(e)	declassify
|	`assume`(e)	assume