Towards Verified, Constant-time Floating Point Operations

Marc Andrysco
UC San Diego
andrysco@cs.ucsd.edu

Andres Nötzli
Stanford
noetzli@stanford.edu

Fraser Brown
Stanford
mlfbrown@stanford.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Deian Stefan
UC San Diego
deian@cs.ucsd.edu

ABSTRACT

The runtimes of certain floating-point instructions can vary up to two orders of magnitude with instruction operands, allowing attackers to break security and privacy guarantees of real systems (e.g., browsers). To prevent attacks due to such floating-point *timing channels*, we introduce CTFP, an efficient, machine-checked, and extensible system that transforms unsafe floating-point operations into safe, constant-time computations. CTFP relies on two observations. First, that it is possible to execute floating-point computations in constant-time by emulating them in software; and second, that most security critical applications do not require full IEEE-754 floating-point precision. We use these observations to: eliminate certain classes of dangerous values from ever reaching floating-point hardware; emulate floating-point operations on dangerous values when eliminating them would severely alter application semantics; and, leverage fast floating-point hardware when it is safe to do so. We implement the constant-time transformations with our own domain-specific language that produces LLVM bitcode. Since the transformations themselves equate to bit surgery on already complicated floating-point arithmetic, we use a satisfiability modulo theories (SMT) solver to ensure that their behavior fits our specifications. Finally, we find that CTFP neither breaks real world applications nor incurs overwhelming overhead.

ACM Reference Format:
Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. 2018. Towards Verified, Constant-time Floating Point Operations. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243766

1 INTRODUCTION

The IEEE-754 standard was developed for efficient and precise computation over floating-point (FP) values. The standard allows hardware vendors to implement fast paths for common, easy combinations of instructions and operands—addition or multiplication by zero—while falling back to slow paths for rare, complex values—addition or multiplication by special subnormal numbers [18]. This path bifurcation allows hardware developers to make (simple) computations over common-case values orders of magnitude faster than (complex) computations designed to precisely account for special values.

Alas, every well-intentioned fast path eventually becomes an attacker's instrument. For example, Andrysco et al. show that attackers can exploit the timing differences between floating-point computations to exfiltrate the values of sensitive data, breaking Firefox's same-origin policy and the Fuzz database's differential privacy guarantees [4]. For Firefox, they implement Stone's *pixel stealing* attack [38] by measuring page rendering time. To break Fuzz [21]—a system specifically engineered to prevent covert timing attacks—they craft queries that return subnormal results for particular (private) values, and amplify the timing signal into one strong enough to break Fuzz's differential privacy guarantees.

In response to these attacks, system designers have tried to plug floating-point timing channels by using existing hardware features to force operations to take a fixed amount of time. For example, to address floating-point-based SVG filter attacks, Chrome browser developers set the "flush-to-zero" (`FTZ`) and "denormals-are-zero" (`DAZ`) CPU flags before calling the SVG library code [10, 23]. These flags set subnormal outputs and inputs to zero, respectively; since the timing attacks rely on subnormal values, enabling flags before executing the SVG filters prevents pixel stealing [12]. The Escort system [34] proposes a more general defense in which each FP operation is simultaneously executed on both a real input value and a dummy subnormal value via single instruction, multiple data (SIMD) instructions. Since SIMD instructions should execute in parallel and block until *both* results are computed, this tries to ensure that each FP operation takes exactly as long as the worst-case, subnormal running time.

Unfortunately, in relying on hardware alone to resolve timing variabilities, these proposals fall short in several ways. The `FTZ+DAZ` flags are incomplete—they focus exclusively on subnormals, but it turns out that other floating-point values can cause timing differences, too. For example, Kohlbrenner and Shacham show that certain operations speed up on non-subnormal values like zero and four [25]. In this paper, we confirm these timing measurements on additional CPUs and report three new classes of values that show timing variability for certain operations: `NaN`s, `Inf`s, and negatives. Timing channels based on such values are hard to prevent with existing approaches, since existing approaches repurpose existing hardware—and there is a limit to how many kinds of pipes you can fix with a hammer.

Leveraging CPU features also demands a deep understanding of different—almost exclusively proprietary and closed-source—CPU

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243766

details; if you're not Intel you're probably wrong. For example, Kohlbrenner and Shacham observe that certain CPUs execute some portion of the *seemingly*-parallel SIMD operations serially—not, like the name implies, in parallel [25]. Escort's design overlooks this completely unexpected detail, so the system remains vulnerable to timing attacks both on subnormals and on other values like zero.

In this paper, we introduce CTFP: an efficient, machine-checked, and extensible software-based approach to preventing FP timing channels. CTFP is based on two key insights. First, floating-point operations can be executed in constant-time by efficiently emulating them in software. Second, for the vast majority of security-critical applications, the full IEEE-754 precision is unnecessary. These two insights are distilled into program transformations that replace unsafe, elementary floating-point operations (e.g., multiplication and division) with new constant-time variants.

Since fully emulating IEEE floating-point arithmetic is impossibly slow, our constant-time operations are implemented as *decorators* for the existing hardware instructions. These decorators ensure that the hardware floating-point instructions are never called with inputs that may introduce time variability. For example, our decorators for the `sqrt` instruction eliminate, among other values (Section 2.1), subnormal and negative numbers from the input space to avoid FPU slow- and fast-paths, respectively. When encountering such inputs, the decorators instead emulate the operation using constant-time instructions. In all other cases, the decorators simply invoke the corresponding hardware instruction.

CTFP provides two classes of decorators, Full and Restrict, that trade-off precision and performance. Both classes of decorators eliminate subnormal input values from the input space by flushing them to zero, much like `DAZ`. The two decorators, however, differ in how they treat normal, but *dangerous* values that (may) produce subnormal results. Full decorators emulate operations on such values and only flush results that are indeed subnormal, much like `FTZ`. Restrict decorators, on the other hand, altogether eliminate dangerous values from the input space by flushing them (and thus their results) to zero—a precision hit that comes with a dramatic performance boost. Outside these ranges, both Full and Restrict preserve the IEEE-754 semantics.

Implementing constant-time floating-point instructions with correct semantics is hard. We must not only handle the complexity of floating-point arithmetic, but do so differently for each instruction—different floating-point operations are vulnerable to different classes of values—and in an extensible way—as new dangerous classes of values come to light, CTFP must be updated to account for them. To this end, CTFP is implemented as a domain-specific language (DSL) for specifying decorated, verified floating-point instructions. At its core, the DSL provides building blocks for **(1)** defining simple decorators (e.g., flush to zero if subnormal); **(2)** composing simple decorators into decorated instructions or more complex decorators (e.g., to handle multiple classes of values); and, **(3)** specifying the semantics of decorators and decorated instructions as pre- and post-conditions. The DSL compiler, in turn, generates constant-time LLVM bitcode for each decorated instruction, which we mechanically check for safety and precision issues using our verifier.

Machine-checked CTFP is only useful if it does not kill performance or ruin program semantics. We found it encouraging that FP-heavy programs transformed to use our CTFP constant-time instructions did not incur severe overheads—Full imposes an overhead of 3×–29×, while Restrict imposes an overhead of 1.7×–8.1×. For both, the impact on semantics is negligible. Both versions of CTFP pass all the tests from SPECfp, the floating-point performance testing suite from the SPEC 2006 benchmarks. Of the 913 unit tests from the Skia graphics library, Full passes all but 5 and Restrict fails only 7. More importantly, both Full and Restrict pass the entire suite of 654 rendering tests.

2 CONSTANT-TIME FLOATING-POINT

In this section, we give an overview of the classes of values that introduce floating-point (FP) timing channels and describe how CTFP addresses these timing channels. We first give a brief background on floating-point and how timing channels arise when FP operations exhibit different timing behaviors for different classes of values (Section 2.1). Then, we experimentally identify the different classes of values that lead to timing channels (Section 2.2). Finally, we sketch our approach to addressing the timing channels by decorating FP operations to execute in constant-time (Section 2.3).

2.1 Floating-Point and Timing Attacks

IEEE-754 is a standard for floating-point arithmetic that specifies all aspects of a floating-point number system from values (e.g., 32-bit single-precision, 64-bit double-precision, and 128-bit quad-precision), to rounding behavior and error conditions [6]. The standard was created to ensure hardware-software interoperability and, as a result of its popularity, essentially all modern CPUs include a specially designed floating-point unit (FPU) that accelerates floating-point operations.

In IEEE-754, the encoding of floating-point numbers consists of three different parts—a sign bit, a w-bit exponent, and a fixed-width significand. Under this encoding, the value of a *normal* floating-point number is specified by the following formula:

$$(-1)^{sign} \times 1.significand \times 2^{exponent-bias},$$

where $bias = 2^{w-1} - 1$ and both the exponent and significant are treated as unsigned integers, the latter's bit-representation prefixed with a 1. The range for exponents for normal floating-point numbers is limited to $[1, 2^w - 2]$; exponents outside this range encode special values. For example, zeros (+0 and -0) are encoded by setting both the *exponent* and *significand* bits to 0s; infinities (+`Inf` and -`Inf`) are encoded by setting the *exponent* bits to 1s and the *significand* bits to 0s; and Not-a-Number values (`NaN`s) are encoded by setting the *exponent* bits to 1s and the *significand* to a value above zero. Lastly, subnormals are encoded by setting all the *exponent* bits to 0 and the *significand* to a value above zero. Subnormals (or denormals) are extremely small numbers, useful in retaining precision for computations on small numbers [6]; they are computed according to:

$$(-1)^{sign} \times significand \times 2^{-bias}$$

The running time of floating-point operations can vastly depend on the values of the operands. Some FPU operations exhibit speed-ups on specific operands. Consider, for example, computing the square root of a negative number. This computation can be much faster than the the square root of an average positive number—the FPU can simply short circuit the implementation of `sqrt` by

Operation	Subnormal	Negative	Powers-of-two	Powers-of-four	NaNs, Infs, and zeros
Addition/Subtraction	✗	✓	✓	✓	✓
Multiplication	✗	✓	✓	✓	✓
Division	✗	✓	✗	✗	✗
Square root	✗	✗	✓	✗	✗

Figure 1: Classes of unsafe floating-point values. We distinguish (✓) and unsafe (✗) floating-point operations on these different classes of values. We note that operations that produce subnormal results—even if their inputs are not subnormal—are unsafe.

checking the the *sign* bit of the operand and return `NaN` if it is set. In fact, CPU designers would have to go out of their way to prevent such an "optimization." In contrast, since few applications demand extremely high precision, most FPUs implement operations on subnormals in microcode, rendering these operations far slower than those on normal inputs.

Measurable timing differences can be turned into covert *timing channels*. Attackers can leverage them to learn secret values, even when the rest of an application was engineered to keep instruction and memory access sequences independent of secret data [4]. For example, Andrysco et al. [4] show how subnormals can be exploited to break Firefox's same-origin guarantees by using them to implement Stone's *pixel stealing* attack [38]. Intuitively, the attack works by having the user render pages using using SVG filters that are specifically crafted so that when a pixel is *black* (resp. *white*), the filter results in many fast (resp. slow) $0 \times \text{subnormal} = 0$ (resp. $1 \times \text{subnormal} = \text{subnormal}$) operations. An attacker observing the timing differences can reconstruct the pixels and therefore the contents of the web-page, completely circumventing the browser's same-origin policy.

2.2 Classes of Potentially Dangerous Values

For performance, CTFP relies on the FPU to handle the bulk of the work of floating-point computations—but, since the latency of different computations may vary with arguments, CTFP must only directly use the FPU to execute floating-point operations on safe classes of values. Identifying all safe values is prohibitively expensive. So, beyond designing CTFP to be easily extensible, we follow in the lines of previous work on constant-time FP [4, 25, 34] and seek to identify classes of potentially dangerous values.

To identify unsafe classes of values, we measure the running time of operations (addition, subtraction, multiplication, division, and square root) with different inputs. Figure 2 presents the results of our measurements for different x86 CPUs. (In Section 5.1, we describe our measurement approach in detail.) We conservatively consider classes of values to be unsafe if they are unsafe on any CPU—this ensures that CTFP-transformed x86 is portable and can be safely used across CPUs.

As shown in Figure 1, our measurements establish several classes of unsafe values: subnormal numbers, negative numbers, powers-of-two, powers-of-four, and special values—zeros, `Inf`s, and `NaN`s. While many of these were established by Andrysco et al. [4] and, more recently, Kohlbrenner and Shacham [25], our measurements confirm their observations on different CPUs and identify new classes of unsafe values—negative numbers, `NaN`s, and `Inf`s. In addition, we modify Kohlbrenner and Shacham's tests to distinguish powers-of-twos from powers-of-four and find the two classes to exhibit different timing behavior. Below we describe these different classes in more detail.

Subnormals. Subnormal values induce an order-of-magnitude slowdown on most CPUs for most operations. They differ from other classes of values in not only being unsafe as inputs to FP operations, but also as outputs: operations that produce subnormal results—even if operating on non-subnormals—are extremely slow.

Negative Values. Negative values are unsafe as operands to `sqrt`. Taking the square root of a negative exhibits a speed-up on some CPUs, since the operation amounts to checking a single bit before returning `NaN`.

Powers-of-Two. Powers-of-two are floating-point values that have a zero significand (see Section 2.1). Division with a power-of-two value as the divisor exhibits a speed-up on several CPUs for both single- and double-precision. In contrast to other classes of values, dividing by a power of two is equivalent to performing an addition on the dividend's exponent, a cheap and fast operation.

Powers-of-Four. Powers-of-four are floating-point values where the significand is zero *and* the exponent is even (see Section 2.1). While powers-of-four values exhibit the same behavior as powers-of-two values for division, the two classes differ for square root operations. Specifically, powers-of-four exhibit a speed-up on several CPUs for `sqrt`. This is not surprising since taking the square root of a power-of-four is equivalent to dividing its exponent by two, i.e., a simple bit-shift.

Special Values. Special values include floating-point values of zero, `Inf`, `NaN`, and their negative counterparts. Special values show a speed-up for division and square root on many CPUs. This is not surprising since division and square root on these values have fixed results: zero divided by anything (except zero or `NaN`) is zero, the square root of `NaN` is `NaN`, etc.

2.3 Eliminating Timing Attacks with CTFP

To address attacks that leverage floating-point timing channels, we eliminate timing difference in FP computations by emulating elementary operations (addition, multiplication, etc.) on unsafe values in software, only leveraging the FPU, internally, to perform safe constant-time computations. Our constant-time emulation is not perfect, though. We relax some of the floating-point semantics in return for performance when emulating operations on *deadly* and *dangerous* values.

An argument to a particular FP operation is deadly if it *always* triggers a slow- or fast-path in the FPU (on some processor). For example, subnormal values are deadly values for all FP operations, while negative numbers are only deadly as operands to `sqrt`. An argument to a particular FP operation is dangerous if it can *ever*

Processor	+ subnormal	+ special	× subnormal	× special	÷ subnormal	÷ special	$\div 2^n$	$\div 4^n$	$\sqrt{\text{subnormal}}$	$\sqrt{\text{special}}$	$\sqrt{2^n}$	$\sqrt{4^n}$	$\sqrt{-x}$
						Single-precision operations							
Intel Core i7-7700 (*Kaby Lake*)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-6700K (*Skylake*)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-3667U (*Ivy Bridge*)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Xeon X5660 (*Westmere*)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Atom D2550 (*Cedarview*)	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗
						Double-precision operations							
Intel Core i7-7700 (*Kaby Lake*)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-6700K (*Skylake*)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-3667U (*Ivy Bridge*)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Xeon X5660 (*Westmere*)	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Atom D2550 (*Cedarview*)	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✗	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗

Figure 2: Safe and unsafe single- and double-precision operations for various processors. A checkmark (✓) indicates the operation is safe, and a cross (✗) indicates the operation is unsafe. The notation 2^n and 4^n denote powers-of-two and powers-of-four, respectively.

produce a subnormal output—and therefore trigger a slow-path in the FPU. For example, very small normal values are dangerous as operands to a FP multiplication (if their result is subnormal).

A faithful, but unacceptably slow IEEE-754 implementation would emulate operations on all values (including deadly and dangerous values). A fast, but unacceptably imprecise implementation would eliminate all deadly and dangerous values from the input space. CTFP presents two classes of decorated floating-point instructions that are in the sweet-spot: they are efficient while preserving floating-point semantics relevant to many real-world applications. Our decorators emulate operations on both dangerous and non-subnormal deadly values in software, but perform operations on safe values directly (by calling normal floating-point operations like `fadd` and `fmul`). Full, the more precise version of CTFP, preserves the (almost) full `FTZ+DAZ` floating-point range; Restrict, the less precise version, alters the floating-point range range more significantly but is also significantly faster. We implement CTFP as a DSL that makes it easy to update the Restrict and Full decorators to account for newly discovered unsafe values.

Eliminating Subnormals. CTFP decorates (i.e., wraps) floating-point hardware instructions to eliminate subnormals as both inputs to and outputs from FP operations. Both our Full and Restrict decorators eliminate subnormal operands by flushing them to zero and then performing the intended operation on zero(s).[1] The two classes of decorators differ in how they handle dangerous values.

To ensure that no FP operation produces subnormals, Full emulates operations on dangerous input values—the values that may produce subnormals. This is non-trivial, since operations on dangerous values do not *necessarily* produce subnormals. Full thus checks if the result of the operation would be subnormal, without actually performing the potentially unsafe operation. If the result is guaranteed to be normal, Full performs the floating-point operation on the dangerous value(s); otherwise, it produces zero—of course ensuring that the entire computation runs in constant-time.

Our range-restricted decorators, Restrict, do not emulate operations on dangerous values. Instead, Restrict eliminates dangerous input values altogether by flushing them to a zero or infinity (depending on the operation). Compared to Full, this approach is dramatically faster (see Section 5). Of course, it also significantly alters the semantics of several FP operations—for example, our Restrict decorators for division essentially eliminate half the input space. In practice, though, this seems to not damage the behavior of many applications (see Section 5).

Emulating Operations on Deadly Values. Both classes of decorators emulate operations on other deadly input values by special-casing each class of values—powers-of-twos, zeros, `NaN`s, etc. At a high-level, a CTFP decorated instruction **(1)** replaces each deadly operand with a dummy, safe value; **(2)** performs the operation on the dummy value(s); and, **(3)** returns the actual result, as if were computed on the deadly value(s). For example, consider executing $4 \times \infty$ using our constant-time `fmul`. Instead of running `fmul` directly on the two inputs, the decorator first checks if either argument is an infinity. Since one of the arguments is ∞, CTFP replaces that value with a dummy value that causes no timing variation for `fmul`. Then, CTFP executes `fmul` on 4 and the dummy value,

[1] Of course, since zero is a deadly value for certain operations, the intended operation may itself be further decorated.

and overwrites the result with infinity—all in constant-time. In Section 3, we describe the implementation of the Full and Restrict decorated instructions in detail.

Correct and Extensible Decorators. Decorating floating-point instructions is error-prone: decorators have to account for all the intricacies of floating-point arithmetic. Moreover, decorated instructions must be extensible—in a month or a year or five years, someone may discover a whole new class of unsafe floating-point values. To account for these issues we implement CTFP as a DSL, so developers can specify new decorated instructions using generic high-level building blocks. Then, they can ensure that their new instructions are correct using our verifier, and compile them into optimized LLVM bitcode. Section 3 and Section 4 respectively describe the DSL and verifier in detail.

Threat Model. We assume an attacker capable of running floating-point computations (e.g., as SVG filters [38]) on sensitive data in an attempt to leak the data. These computations, however, are restricted to our decorated constant-time floating-point operations and cannot use the hardware instructions directly. To this end, CTFP provides a program transformation that rewrites existing operations with our constant-time variants (see Section 3.4). We consider attacks that abuse other CPU instructions (e.g., branching) outside the scope of this work—instructions such as conditional branches can trivially introduce time-variability, however, and must be addressed in practice. We believe techniques that address these broader concerns (e.g., [8, 28, 33]) are complimentary to CTFP.

3 THE CTFP DSL

CTFP ensures that floating-point operations execute in constant-time using carefully crafted bitmasking operations. In this section, we describe how to create the bitmasking operations using the CTFP domain specific language (embedded in Haskell). Then, we show how CTFP goes from DSL implementations of Restrict and Full to LLVM bitcode.

At its core, CTFP consists of basic floating-point *operations* `op`, and *decorators* `tx`. Each `op` represents—and is compiled to—a low-level LLVM function. Each decorator takes as input an operation and returns a new operation that "wraps" or "transforms" the input operation; we can use the decorators to transform unsafe operations into their safe counterparts. Both decorators and operations are represented in the DSL "meta-language," Haskell. We develop CTFP by composing multiple decorators $\texttt{tx}_1, \texttt{tx}_2, \ldots$, each of which accounts for a subset of unsafe inputs. Inspired by Python [36], we write `tx @ op` for the result of applying the decorator `tx` to the operation `op`, and $\texttt{tx}_1$ `@` $\ldots$ `@` $\texttt{tx}_n$ `@ op` for the result of composing the decorators $\texttt{tx}_1, \ldots, \texttt{tx}_n$. Below, we describe the building blocks of our DSL and the four decorator "strategies" that we use to implement the Restrict and Full operations.

Primitives. Our DSL has the following primitives that correspond to the machine operations that test and manipulate floating-point values.

- Floating-point and integer literals.
- Floating-point arithmetic functions, including addition (`fadd`), subtraction (`fsub`), multiplication (`fmul`), division (`fdiv`), and square-root (`fsqrt`).
- Bitwise functions, including `and`, `or`, `xor`, and `not`.
- Comparison functions, including ordered equality (`oeq`), ordered less than (`olt`), ordered greater than (`ogt`), unordered less than or equal to (`ule`), unordered greater than or equal to (`uge`), unordered-is-negative check (`isUneg`), and a `NaN` check (`isNaN`).
- Sign functions, including `abs`—which computes the absolute value of a floating-point value—and `copySign`—which creates a value with a given magnitude and sign.

Conditionals. Since conditional branching can introduce timing variabilities, our DSL does not provide general conditional branching primitives [1]. Instead, we provide a function `ite` that can be used to select between two values: `ite b x y` evaluates to `x` if `b` is true and `y` otherwise. Internally, `ite` is implemented using constant-time bitvector operations:

```
ite b e1 e2 = (b `and` e1) `or` (not b `and` e2)
```

We ensure that the condition variable (`b`) is always true (i.e., all `1`s) or false (i.e., all `0`s). In the former (resp. latter) case, `ite` returns `e1` (resp. `e2`).

Calls. Recall that each decorator `tx` is a meta-function that takes as input an `op` function and returns a new function. The new, decorated function may call `op`—or any other primitive function—via the DSL primitive `call`.

Generic Blinding Decorator. We express all CTFP decorators using the generic *blinding* decorator shown below:

```
txBlind isUnsafe blind fix op = λvs ->
  let unsafe   = isUnsafe vs
      safeVs   = ite unsafe (blind vs) vs
      res      = call op safeVs
      fixedRes = ite unsafe (fix vs res) res
  in fixedRes
```

This decorator takes as input four functions:[2]

- `isUnsafe`: tests if any input value is unsafe,
- `blind`: replaces the unsafe inputs with safe values that can be used as inputs to `op`,
- `fix`: produces a correct output given the (potentially) unsafe inputs and the result of the operation, and
- `op`: the operation to be decorated.

Given these input functions, the decorator returns a new "wrapped" operation.

3.1 Core CTFP Decorators

We use the binding decorator `txBlind` to implement the four high-level decorators for all Full and Restrict decorated instructions.

Dummy Value Decorator. For several floating-point operations, emulating the computation on certain unsafe values amounts to returning a constant value. For example, the square-root of `NaN` or a negative number—both unsafe—is `NaN`. To this end, our DSL provides the *dummy value* decorator:

[2] We loosely use the term "decorator" to refer to these high-order meta-functions that take several values, not just the operation to be transformed. Indeed, these high order functions can be thought of as "decorator templates", i.e., functions that produce decorators.

```
txDummy badIn badOut safeIn =
  txBlind (λv   -> oeq v badIn) -- isUnsafe
          (λ_   -> safeIn)      -- blind
          (λ_ _ -> badOut)      -- fix
```

This decorator addresses the timing variability of an operation op (without altering its semantics) by computing on dummy `safeIn`—instead of deadly or dangerous `badIn`—and returning `badOut`, the value that would be returned if the unsafe computation were actually performed. The decorator `txDummies` generalizes `txDummy` to blind multiple deadly inputs at once; this allows our DSL to generate more efficient code than would be possible by repeatedly applying `txDummy`.

Underflow and Overflow Decorators. Restrict and Full manipulate the range of floating-point numbers to eliminate subnormal numbers from the input and output space. CTFP accomplishes this by implementing underflow and overflow in software:

```
txUnderflow lim =
  txBlind (λv     -> olt (abs v) lim) -- isUnsafe
          (λv     -> copySign Zero v) -- blind
          (λ_ res -> res)             -- fix
txOverflow lim =
  txBlind (λv     -> ogt (abs v) lim) -- isUnsafe
          (λv     -> Inf)             -- blind
          (λ_ res -> res)             -- fix
```

Decorator `txUnderflow` takes as input a threshold value `lim` and returns a decorator that replaces all inputs under the threshold with positive or negative zero. `txOverflow` replaces all inputs over the threshold with the special value `Inf`. Both decorators return the result of op on the zero or infinity.

Predict Decorator. Since computations that produce subnormal outputs exhibit time variability, CTFP must *predict* these outputs without performing the dangerous operation:

```
txPredict shift lim safeIn =
  txBlind (λv     -> olt (abs (shift v) lim)  -- isUnsafe
          (λ_     -> safeIn)                  -- blind
          (λv res -> copySign r (shift v))    -- fix
```

This decorator performs a safe, constant-time computation on modified inputs in order to predict whether the output (on the original values) is deadly or not. It takes three inputs:

- a function `shift` that, given the input `v` returns a shifted value `shift v` in lieu of the underlying operation,
- a threshold `lim` that is an upper bound on the magnitude of deadly outputs for the underlying operation,
- and a safe input `safeIn` that should be used if the output is indeed deadly.

Given these inputs, we use `txBlind` to return a decorated operation that executes on `safeIn` if the shifted (output) value falls within the (shifted) deadly region denoted by `lim`, and executes unchanged otherwise.

3.2 Restrict Decorated Instructions

In this section, we show how to use our DSL to implement the constant-time Restrict floating-point operations. Restrict exploits the fact that there is a range of normal floating-point values that are guaranteed to also produce safe, normal outputs for a given floating-point operation. Restrict uses the dummy, underflow, and overflow decorators to flush values outside of this safe range to zero and infinity.

Op	Expression	Float	Double
Add/Sub Min	MIN * 2^SIG - ULP	9.86e-32	1.00e-292
Mul/Div Min	sqrt(MIN)	1.08e-19	1.49e-154
Div Max	1 / sqrt(MIN)	9.22e+18	6.70e+153
Sqrt Min	MIN	1.18e-38	2.22e-308

Figure 3: Threshold values used by Restrict decorators. The constant MIN is the smallest, normal floating-point for a format and SIG is the number of bits in the significand (e.g., MIN = 1.175e-38 and SIG=23 for `float`). For brevity, the table shows constants rounded to three digits.

Each floating-point operation has its own safe input range. For example, for multiplication, an input range [0.1, 10.0] is sufficient to prevent subnormal outputs since multiplying the smallest possibles value of 0.1 with itself yields 0.01 (a value far from any subnormal numbers). To minimize the impact on floating-point precision, however, we want the *largest* safe input range possible; Figure 3 summarizes the cutoffs that CTFP uses for the different operations.

Addition, Subtraction, and Multiplication. To transform addition, we restrict the input range so that any values below `addMin` (9.86e-32) are flushed to zero. The following code demonstrates the Restrict version of addition:

```
restrictAdd = txUnderflow1 addMin
            @ txUnderflow2 addMin
            @ fadd
```

Subtraction is nearly identical to addition, except that we change the base operation to `fsub`. For multiplication, we underflow values smaller than `mulMin` (1.08e-19).

Division. A safe division operator must account for three classes of deadly values. First, *out-of-range* values like very large divisors or small dividends can lead to subnormal outputs. Second, *special* values like `NaN` or `Inf` or `Zero` can trigger special cases that return early (and thus take less time than a "typical" division). Third, some implementations of division enjoy a small but exploitable speedup [25] when the divisor is a power-of-two. We address these three classes of values with three decorators (`txRange`, `txSpecial`, and `divPowers2`), and compose them to implement the Restrict division shown in Figure 4.

First, the `txRange` decorator underflows dividends smaller than `divMin` (1.08e-19) to zero, and overflows divisors larger than `divMax` (9.22e+18) to infinity to ensure that the outputs are not subnormal. We chose these cutoffs to roughly balance the number of floating-point values below the lower cutoff and above the upper cutoff.[3]

Once we eliminate the out-of-range values, we call the `safeDiv` function, a decorated version of division that is safe whenever the inputs and outputs are normal. We implement `safeDiv` by wrapping the base operation `fdiv` with two decorators that respectively account for special values and powers-of-two.

[3]Users can choose their own (application-specific) cutoffs, too.

```
-- | Restrict division ---------------------------------
restrictDiv   = txUnderflow1 divMin
              @ txOverflow2  divMax
              @ safeDiv
-- Decorated division that is safe for normal inputs
safeDiv       = txSpecial
              @ divPowers2
              @ fdiv
-- Emulate division on special values
txSpecial     = txDummies nans  (λ_ -> NaN)  Dummy
              @ txDummies infs  (λ_ -> Inf)  Dummy
              @ txDummies zeros (λ_ -> Zero) Dummy
-- Emulate division on powers-of-two
divPowers2    = divByParts
              @ txDummy2  1.0   (λ(v,_) -> v)    Dummy
              @ txDummies infs  (λ_     -> Inf)  Dummy
              @ txDummies zeros (λ_     -> Zero) Dummy
-- Split division into two steps
divByParts    = txBlind
                  (λ _     -> True)
                  (λ (n,d) -> (fdiv n (getExp d), getSig d))
                  (λ _ r   -> r)
```

Figure 4: RESTRICT-division uses decorators that blind large divisors, small dividends,

- The `txSpecial` decorator emulates division on special inputs—**NaN**s, **Inf**s and **Zero**s—using the `txDummies` decorator to perform constant-time division on a **Dummy** (1.5), normal value.
- `divPowers2` handles powers-of-two divisors. This decorator exploits the insight that a power-of-two divisor can be split into two parts $d = s \times e$ where the significand s lies in the range $[1, 2)$ and the exponent e is exactly a power of two. To this end, our decorator emulates the division computation by performing *division by parts*, i.e., in two steps $n/d = (n/e)/s$ where dividing by e is always "fast" and dividing by s is always "slow". When splitting division into parts, we use `txDummy` to account for the special degenerate case where $s = 1.0$. Also, since the intermediate result of n/e may itself underflow or overflow to **Zero** or **Inf** we again use `txDummies` to handle these safely—recall that **Zero** and **Inf** are special values for division.

Square Root. The RESTRICT implementation of square root must account for four classes of deadly values. First, as with other operations, square root is unsafe when the input is subnormal. Second, as with division, square root exposes a timing variability due to the special values **Zero**, **Inf** and **NaN**. Third, negative input values are deadly for square root. Finally, some implementations of square root are faster on powers-of-four inputs.

We account for these cases by composing a sequence of transforms. We use the `txUnderflow` decorator to eliminate subnormal inputs (values below `fltMin`) and flush them to zero. As with division, we use `txDummy` to emulate the operation on special values **NaN**, **Inf** and **Zero** (in the last case, we return `v` to preserve the sign). The `sqrtNeg` decorator—implemented using `txBlind`—emulates square root on negative input values by performing the computation on a dummy value and returning **NaN**.

Finally, `sqrtPowers4` emulates square root on powers-of-four using `txBlind`. This decorator checks if the value is a power-of-four[4] and, if so, blinds the input by adding a single bit to it (i.e., `sqrtPowers4` computes `sqrt(input + ULP)`, where `ULP` is the unit of least precision). It fixes the result by removing (via a bit-mask) the at-most single incorrect bit from the output.

3.3 FULL Decorated Instructions

Next, we show how to use our DSL to implement the FULL floating-point operators. FULL instructions preserve IEEE-754 semantics for all non-subnormal floating-point values. As with RESTRICT, we flush subnormal inputs to zero using the `txUnderflow` decorators. Unlike RESTRICT, however, we do not flush normal inputs that *may* result in a subnormal output. Instead, our FULL operations only flush values that actually produce subnormal outputs—a decision we perform at run-time. For an operation op on values a and b, we check if $|a \text{ op } b| \geq M$, where M is the smallest normal value. Since this is exactly the timing-variable computation that we wish to avoid, though, our decorated operations rely on the `txPredict` decorator to perform this computation on scaled but timing-safe values, in order to determine whether they are safe.

Addition, Subtraction, and Multiplication. Figure 5 shows the FULL version of addition. To decorate addition, we first apply the underflow decorators to eliminate subnormal inputs. Next, we use the `predictAdd` decorator to emulate addition, accounting for inputs that may result in subnormal outputs. Internally the `predictAdd` decorator multiplies both inputs with a constant C (`addC` in the figure) so that $C \cdot a + C \cdot b$ is *normal* for all possible inputs a and b. It then tests the result against $C \cdot M$ (`addLim` in the figure), i.e., it checks if $|C \cdot a + C \cdot b| \geq C \cdot M$, to determine if the output of the computation is subnormal. If the output is subnormal, the decorator replaces the inputs with **Zero**s; otherwise it leaves the inputs intact. It then performs the floating-point addition on the safe inputs. Subtraction is implemented the same, except the base operation **fadd** is replaced by subtraction **fsub**.

For the decorated multiplication we use the `predictMul` decorator to emulate multiplication after subnormal inputs are eliminated. This decorator performs a *scaled* multiplication to determine if $a \times b$ is subnormal, and if so, sets the inputs to **Zero**. Unlike the `predictAdd` decorator, `predictMul` uses a single multiplication (by `mulC`) to scale the operation outside of the subnormal range, after which the result is compared against the threshold `mulLim` to determine if the output is subnormal.

Division. As with the previous operations, we use `txUnderflow` decorators to eliminate subnormal inputs and a predict decorator, `predictDiv`, to remove normal inputs that produce a subnormal output. The predict decorator for division is more complex, though, since extreme operands may (1) require intermediate computation on values larger than `FLT_MAX` or (2) overflow intermediate values

[4] The binary representation of a power-of-four has all zeros in the significand and an odd exponent—the odd exponent translates to a power-of-four due to the exponent bias. Thus, to determine if the input is a power-of-four, the blinding condition extracts the significand bits of the input and the last bit of the exponent and compares the result to verify all significand bits are zero and the exponent ends in a one bit.

```
-- | Full addition ----------------------------------
fullAdd = txUnderflow1 fltMin
        @ txUnderflow2 fltmin
        @ predictAdd
        @ fadd
-- Try an addition and replace inputs with zeros if output is subnormal
predictAdd = txPredict
               (λ (a,b) -> fadd (fmul a addC) (fmul b addC))
               addLim
               (Zero, Zero)
```

Figure 5: Full addition, subtraction (omitted), and multiplication (omitted) underflow subnormal inputs and any normal inputs that produce subnormal outputs.

to infinity. For the first issue, consider a/b when a is 1.18e-38 and b is 3.4e38. The result of the shifted division using `divC` (8.5e37) will produce the subnormal 2.9e-39. Unfortunately, we cannot address this by simply increasing `divC` to the necessary value of 3.4e38, since this number exceeds the maximum size of single-precision floats. For the second issue, consider when a is 5e31 and b is 2e-07. When `divByParts` splits b into 1.2e-7 and 1.68 such that the first division results in 4e38, a value that incorrectly overflows to infinity. Consequently, we must account for the possibility of the first division—the division by the exponent—overflowing without giving up on precision. For both cases—when the trial division is subnormal or when the intermediate value in `divByParts` overflows—we use the decorator `extremeDiv` to shift the value of `b` up or down to avoid subnormals and overflow.

Square Root. The Full square root operator is identical to the Restrict one shown in Section 3.2 since the square-root of a normal number is guaranteed to be normal.

3.4 Implementation

CTFP generates LLVM code from our DSL by generating a new LLVM function for each decorated term, yielding constant-time equivalents for each floating-point operation. Given an DSL term $\texttt{ctOp} = \texttt{tx}_1 \texttt{ @} \ldots \texttt{@ tx}_n \texttt{ @ op}$ CTFP generates a sequence of functions $f_1 \ldots f_{n+1}$ where each f_{n+1} is just `op` and each f_i is the result of applying the decorator $\texttt{tx}_i$ to f_{i+1}.

We use Clang version 6.0.0 on Linux to generate constant-time machine code targeting Skylake processors (`-march=skylake`). We manually verify this machine code is free of unsafe instructions (e.g., conditional branches).

Vectorized CTFP Transformations. For each instruction, our tool supports `floats` and `doubles` as either scalars or vectors of sizes 2, 4, 8, or 16. Supporting vectorized CTFP transformations is crucial for performance. Without vector support, the CTFP transformations must run before the vectorization passes of LLVM, and the CTFP code's complexity prevents the future vectorization passes from succeeding. By supplying vector versions of the CTFP transformation, the pass is able to run after vectorization and take full advantage of the resulting SIMD instructions.

Source Code. The full source code for CTFP is publicly available at http://ctfp.programming.systems. The repository includes the Haskell DSL for generating Restrict and Full floating-point operations, the LLVM plugin for applying CTFP as a optimization pass in Clang, and our verifier.

4 VERIFYING CTFP WITH SMT

As demonstrated by the decorators in Section 3, floating-point computations have various subtle corner cases, and it is quite easy to get transformations wrong. To address this issue, we implemented LLVC, a verifier for the LLVM fragment used by CTFP. LLVC converts LLVM programs and specifications into logical formulas called *verification conditions* (VCs) whose *validity* **(1)** can be automatically checked by SMT solvers, and **(2)** implies that the LLVM program adheres to the given specification. Next, we describe LLVC and how we use it to verify that CTFP's decorated operations indeed eliminate inputs and outputs that could lead to timing variability and produce results that are equivalent to the IEEE-754 behaviors for the respective Restrict- or Full-value ranges. During development, we found and fixed six bugs using verification, from tricky cases involving special values to semantic differences caused under certain rounding modes.

SMT Solvers. SMT solvers are automatic decision procedures that can determine whether a formula is *satisfiable*, i.e., whether a formula evaluates to *true* for *some* assignment of values to the variables in the formula. The formula is *unsatisfiable* if no such assignment exists. Dually, we say a formula is *valid* if the formula evaluates to *true* for *all* assignments of values to the variables. Note that a formula is valid if its negation is unsatisfiable. For example, when queried with formula $a \vee b$, an SMT solver will return a satisfying assignment $a = true, b = false$ that makes the formula evaluate to *true*. When queried with $a \wedge \neg a$, the SMT solver will return `unsat`, indicating that no satisfying assignment exists. In other words, the negation of the above, namely $\neg(a \vee \neg a)$ is valid.

Reasoning about Floating-Point Arithmetic with SMT. Users can express relatively high-level concepts about program structures as SMT formulas by using functions and predicates that belong to *theories*. Theories include integers, arrays, and bit-vectors. We encode the semantics of CTFP transformations using the theories of bit-vectors and floating-point arithmetic [9] formalized in SMT-LIB [7], an initiative for standardizing theories and input languages across different solver implementations. The floating-point theory is a formalization based on the IEEE-754 standard, and includes basic arithmetic operations, `sqrt`, the five different rounding modes, and special values. We use commit `95963f7` of Z3 [16] as our SMT solver because it is one of the few solvers that currently support the floating-point theory.

4.1 Specification

Function Contracts. In LLVC, we model *all* operations i.e., LLVM primitives like comparisons, bitwise operations, `select` and arithmetic, and of course, `call`—as function calls. In LLVC, users can specify desired properties as Floyd-Hoare style function contracts comprising a *precondition* that callers must establish on the function's inputs, and a *postcondition* that the function will guarantee about the function's output [22]. Next, we show how we can use contracts to specify the two key properties that CTFP ensures

```
;; precondition for addition
(define_fun fadd32_requires ((a F32) (b F32)) Bool
   (not (fadd32_deadly a b)))
;; postcondition for addition
(define_fun fadd32_ensures ((ret F32) (a F32) (b F32)) Bool
   (= ret (fp.add rm a b)))
;; deadly inputs for addition
(define_fun fadd32_deadly ((a F32) (b F32)) Bool
   (or (fp.isSubnormal a)
       (fp.isSubnormal b)
       (fp.isSubnormal (fp.add rm a b))))
```

Figure 6: Contract for the primitive 32-bit addition.

- ▶ *Safety:* The decorators must prevent any unsafe, timing-sensitive values from being passed as inputs to or returned as outputs from floating-point instructions, and
- ▶ *Equivalence:* The decorators must preserve floating-point semantics for safe values so that the end-to-end application behavior is unchanged.

Safety Contracts. The safety contracts are the same regardless of Restrict or Full mode. For each operator we *require* that their inputs be *safe*, i.e., not any of the special timing-sensitive values, and that they do not produce timing sensitive values as output. Dually, for each operator we *ensure* that the output is exactly the result of applying the respective operator to the two inputs.

For example, for the `fadd` operation, we specify the contract shown in Section 6. The precondition `fadd32_requires` is a predicate over the two inputs `a` and `b` that says that the `fadd` must be called with inputs that are normal, and which produce normal outputs. The postcondition `fadd32_ensures` is a predicate over the two inputs and the output **`ret`** which says that the operator returns exactly the result of the floating-point addition of its two inputs.

Equivalence Contracts for Restrict. We specify the *equivalence* property for an operator `op` by writing contract for the top-level decorated function corresponding to `op`. This contract specifies a precondition that the inputs of the function can be called with *any* input because the decorated functions must properly account for all possible floating-point input values. However, the contract specifies that the output value is only correct for *non-dangerous* inputs that fall outside the threshold specific to the operator.

For example, for `fadd` we specify the contract for the top-level function `@restrictAdd1` from Figure 7 that says that **(1)** the inputs must satisfy the precondition *true* and **(2)** the function return value (**`ret`**) satisfies the postcondition:

```
(define_fun restrictAdd_ensures ((ret F32) (a F32) (b F32))
   (= ret (fp.add rm (underflow a addMin)
                     (underflow b addMin))))
```

This postcondition ensures that outputs preserve floating-point semantics *if* the inputs are not dangerous, i.e., the inputs are larger than `addMin`; otherwise, the function returns the result performing `fadd` on `underflowed` (to zero) inputs. We precisely define the semantics of underflowing as the SMT predicate:

```
define weak float @restrictAdd1(float %a, float %b){
  ;@ requires true
  ;@ ensures  (restrictAdd_ensures %ret %a %b)
  %1 = call float @llvm_fabs(float %a)
  %2 = fcmp olt float %1, 0x3980000000000000
  %3 = select i1 %2, i32 -1, i32 0
  %4 = xor i32 %3, -1
  %5 = bitcast float %a to i32
  %6 = and i32 %4, %5
  %7 = bitcast i32 %6 to float
  %8 = call float @llvm_copysign(float %7, float %a)
  %9 = call float @restrictAdd2(float %8, float %b)
  ret float %9
}
```

Figure 7: LLVM code for Restrict addition.

```
(define_fun underflow ((val F32) (lim F32)) F32
   (ite (fp_lt (fp_abs val) lim) (copysign zero val) val))
```

The exact threshold—here, `addMin`—depends on the type of the operation `op` as laid out in Figure 3.

Equivalence Contracts for Full. As before, we specify the equivalence contract for Full as a contract for the top-level operator, where the precondition is *true*. However, this time, the contract specifies that the output value is correct for all *non-deadly* inputs and underflows otherwise:

```
(define_fun fullAdd_ensures ((ret F32) (a F32) (b F32)) Bool
   (= ret (underflow (fp.add rm (underflow a fltmin)
                                (underflow b fltmin))
                     fltmin)))
```

Optimality. In addition to checking for safety and equivalence, we use the SMT solver to determine whether our choice of thresholds used in the Restrict decorators is *optimal*, i.e., the threshold C chosen for `op` is as small as possible.[5] To check optimality, we change the code and conditions to use the next smaller floating-point value for C and query the SMT solver to prove that they admit a deadly computation. For example, for Restrict addition, we check that the threshold `addMin` is optimal by checking that the SMT solver finds a counterexample that violates the safety contracts when we change `addMin` to `addMin` – ULP.

4.2 Verification

Next, we describe how LLVC converts LLVM programs and contract specifications into VCs whose validity formally guarantees that the generated code enjoys the safety and equivalence properties, i.e., proves the constant-time and correctness properties *for all* inputs. If the VC is *invalid*, i.e., its negation is satisfiable, the solver returns a *counterexample*: a concrete set of floating-point inputs on which some property is violated, which we found invaluable in hunting down tricky bugs in our CTFP implementation.

[5] Division has two separate thresholds for the dividend and the divisor, which we translate to two separate properties. The threshold for the divisor is special because we want it to be as large as possible.

Contracts for LLVM Primitives. Recall that in LLVC, the semantics of all LLVM primitives are modeled as contracts. This modeling is eased by the fact that many floating-point LLVM instructions have a corresponding function or predicate in the SMT-LIB floating-point theory. For example, the LLVM instruction `fadd float %1, %2` is modeled by the contract in Figure 6. The postcondition describes the output of the operation and is modeled as a single SMT operation, `fp.add`. For some instructions, we have to take care to precisely model the semantics by combining other SMT-LIB predicates. For example, the LLVM not-equals comparison `fcmp une float %a, %b` returns `true` if any of the arguments of the comparison are **NaN**. We model these by turning the original comparison $a \diamond b$ into $a \diamond b \vee \texttt{isNaN}(a) \vee \texttt{isNaN}(b)$, via the following postcondition:

```
(define_fun une_ensures ((ret F32) (a F32) (b F32)) Bool
  (= ret (or (fp.isNaN a)
             (fp.isNaN b)
             (not (fp.eq a b)))))
```

Finding bugs in CTFP. The verification tool found, and helped us fix, several subtle errors in our design of the Restrict and Full operations:

- An early version of Full missed the fact that $\infty - \infty = \texttt{NaN}$; consequently, it used an ordered (instead of an unordered comparison), which led to large values underflowing to zero.
- An early version of Full was wrong when rounding towards zero. In this rounding-mode there is no overflow to ∞, which breaks addition (as described in Section 3.3).
- The trial comparison in Full multiplication was off by 1 ULP.
- Our optimality check allowed us to improve the cutoff thresholds for the Restrict addition. In earlier versions, we found that the threshold was too conservative in that it unnecessarily rounded values down.
- Our safety check discovered that an earlier version of the Restrict division was using a threshold that was too aggressive: it was possible to generate inputs that yielded subnormal *output values*, thereby violating CTFP's constant-time guarantees.
- Finally, the computation in `predictDiv` produced subnormal *intermediate* values. As a result, we developed the `extremeDiv` decorator to handle specific cases of large and small values.

In addition to the fixed issues, the verification tool discovered the limitation that the Full multiplication and division are not exact when the result is `FLT_MIN` or `DBL_MIN`.

We remark that every one of these issues involved tricky floating-point semantics and special values. This made them hard to detect manually ahead of time—LLVC's SMT-based verification proved invaluable for implementing CTFP.

Verifying Restrict and Full Transformations. While the SMT solver was able to find the above bugs in a few seconds, proving the safety and equivalence took much longer. Figure 8 lists the operations and the time taken for Z3 to verify each one. As expected, verifying Restrict transformations is significantly faster than their Full counterparts. The verification times also show that division is, by far, the most complex operation to verify; this is not surprising—division consists of many decorators and the underlying operation itself is complex to reason about. To improve performance, we manually introduce binary clauses of the form `(or x (not x))` and ask Z3 to split them into subgoals. We find this effective when we use it on the values of the initial conditions of some of the decorators. Intuitively, this is because most of our decorators check for some bad value at the start of the function. Depending on the outcome of that check, the possible register values are often very different (e.g., a certain register may always be zero if the check succeeds), so treating the two outcomes as separate subgoals can be useful—the solver does not have to discover the variable to split on by itself.

	Verification Time		
Operation	Float	Double	# Assumes
Restrict add	0.7s	1.1s	0
Restrict sub	0.7s	1.3s	0
Restrict mul	1.4s	6.4s	0
Restrict div	120.7s	—	1
Full add	10.1s	45.6s	0
Full sub	10.2s	48.7s	0
Full mul	11.2s	49.0s	1
Full div	573.7s	—	6
Full sqrt	15.9s	122.5s	0

Figure 8: The time taken for Z3 to verify each operation. The last column indicates the number of assumptions that we added. We use the same assumptions for both bit-widths.

Figure 8 also highlights the number of assumptions we require for verification to complete—without them, the solver does not finish within a reasonable time limit. The assumptions that we rely on largely help Z3 reason about scaled values (e.g., when determining whether the output of an operation could be subnormal). Our Restrict decorators require a single assumption: for the Restrict division we assume the outcome of `divByParts` is correct for the given range. Full is slightly more demanding in requiring eight assumptions. Two of the Full division assumptions are directly analogous to our Restrict assumption, one for each `divByParts` performed. In addition, one of the `divByParts` decorators requires an assumption that the values passed to the following decorator fall into a specified range. The preconditions are slightly different in both applications of the decorator, so the reasoning about the possible values is different. Two assumptions help Z3 reason about the `extremeDiv` decorator; the decorator avoids extreme output values by scaling the inputs, our assumptions say that the scaled computations do not change the result. The remaining assumptions state that our subnormal predition decorators are accurate. For example, we assume that `predictMul` accurately predicts whether the result is subnormal or not (except when the result is `FLT_MIN`/`DBL_MIN`). We note that to our 64-bit division remains unverified—verifying 64-bit divisions in a reasonable amount of time requires many more assumptions, defeating the purpose of using verification.

5 EVALUATION

We evaluate CTFP on three fronts—security, correctness, and performance—by answering the following questions: does the CTFP-decorated code

- (Section 5.1) execute in constant-time?
- (Section 5.2) change the input-output ranges as specified?
- (Section 5.3) break the behavior of applications?
- (Section 5.4) perform with acceptable overheads?

We find that CTFP indeed eliminates exploitable timing vulnerabilities; i.e., the decorated floating-point operations execute in constant-time over the range of inputs that we tested. Furthermore, both the Full and Restrict operations do not break Skia's rendering tests or the SPECfp benchmark suite. In our experiments, Restrict CTFP adds an overhead of 1.7–8.0x, while Full CTFP adds an overhead of 3.5x–29x.[6] We compiled all benchmarks with Clang 6.0.0 and ran them on a machine with an Intel Core i7-7700 (Kaby Lake) CPU.

5.1 CTFP Executes In Constant Time

We time the CTFP floating-point operations with different inputs to determine if they actually exhibit any exploitable timing variability.

Inputs and Operations. We choose normal input values and a selection of special values from the following categories: subnormal, zero, one, powers-of-two, powers-of-four, infinity, and `NaN`. We test all combinations of these inputs on all original floating-point operations and their Restrict and Full versions. We use this method because the input space—all combinations of all floating-point numbers—is too large to test exhaustively. Similarly, picking random values to test would rarely yield known problematic inputs since, for example, single-precision floating-point numbers have 2^{32} possible values and only 2^{23} of those are subnormals.

Testing Method. We use the `rdtsc` instruction to measure the running time of executing an operation 32 times. To minimize the effects of out-of-order execution and pipelining, we introduce an artificial data dependency by reassigning the input: `in = m_xor_d(m_xor_d(out, res), in)`.[7] We repeat this process a million times to overcome timing inconsistencies like context switches or hyper-threading and average the median 50% of measurements.

Results. Figure 2 summarizes the results of running the *original* single-precision floating-point operations on different CPUs. All CPUs exhibit timing differences in multiple cases. An operation on a value category is marked unsafe if its runtime differs from that on normal values. Because timing differences can be very small, we consider any consistent performance difference unsafe. In contrast, for the CTFP versions of the floating-point operations, we found that the execution time for all value categories on each instruction (e.g., addition for all values) varies by less than 1%, which falls well below the threshold of a single clock cycle. In other words, our results show that CTFP eliminates any exploitable timing variability from these FP instructions.

[6]Depending on the operation, these overheads are either similar or much lower than Escort's, since Escort effectively slows all operations to the speed of their subnormal variants [34]. The *Subnormals* column in Figure 9 gives a rough estimate of the best-case overhead for Escort.

[7]This is a no-op because the output `out` XORed with the expected output `res` is 0 and that XORed with the input `in` is simply `in`.

5.2 CTFP Changes Ranges as Specified

Both the Restrict and Full decorations change the semantics of floating-point operations: the former only guarantees IEEE-754 semantics for input values in the safe range, and the latter, for all computations that do not consume or produce deadly values. Next, we check that the decorated code returns the same output as the original code in cases where we intend to preserve semantics. Similarly, we test that the decorated code changes semantics correctly when intended, e.g., that subnormal values are indeed replaced by zeroes. To validate the known timing channels, we test each operation using every combination inputs from each value class. In order to find other bugs, we enumerate one million pairs of random inputs and verify that each operation produces the correct result and every primitive operation never observes a deadly value on input or output.

Full Correctness. We formally verify that our decorated floating-point operations preserve the semantics of the underlying operation when neither its inputs nor its output are subnormal (Section 4). As an additional sanity check, we run micro-benchmarks to confirm that CTFP instructions exactly match the IEEE-754 specification with flush-to-zero and denormals-are-zero flags enabled, on various combinations of inputs. Specifically, we test IEEE-75 compliance for all operations over all special values in addition to one million randomly generated pairs of inputs.

Restrict Correctness. The Restrict decorated operations directly evaluate the corresponding machine operations on values that fall within the safe range, and hence, trivially produce equivalent results. Thus, we just test that CTFP zeroes out values that fall outside of the safe range (for a given operation).

5.3 CTFP Preserves Application Behavior

CTFP changes the semantics of floating-point operations. Hence, we evaluate whether these changes are benign or if they can adversely affect real applications. To this end, we evaluate CTFP on the SPEC benchmarks and the Skia 2D graphics engine. Evaluating SPEC and Skia did not require changing their code (and optionally included changing only a few lines of their tests). We just linked against a version of the `MUSL-libc` math library, which we created using an LLVM pass to replace normal floating-point operations with CTFP operations.

SPEC Benchmarks. Both the Restrict and Full operations pass all C and C++ tests from the SPECfp benchmark suite. The SPEC tests demonstrate that restricting the range of floating-point numbers still leaves enough power to complete many scientific computing tasks: physics calculations, molecular dynamics, linear programming, ray tracing, fluid dynamics, and speech recognition.

Skia Graphics Library. We test whether CTFP works on the Skia library, since this library is widely used (e.g., in Google Chrome, Chrome OS, Chromium OS, Firefox, Android, Firefox OS, and Sublime Text 3) and has been the target of a floating-point timing attack [25]. After compiling Skia with both versions of CTFP, we test Skia's default configuration of 1565 tests (excluding those for the

GPU). These tests comprise 913 unit tests and 654 render tests which check the end-to-end functionality of everything from fonts to drawing primitives. When we compiled Skia with the `-ffast-math` flag, the test suite would not complete: it kept failing assertions and looping infinitely. This indicates that Skia is very sensitive to changes in floating-point semantics, which makes it a particularly useful stress test for CTFP.

Full Skia Tests. Comparing the CTFP results to those of the original reference build, we found that Full-Skia passes the same set of rendering tests, but fails on five floating-point-specific unit tests. Three of the unit tests explicitly use subnormal values to test low level vector and matrix operations. Another unit fails in the function `SkFloatToHalf` due to a multiplication by a very small constant that may produce a subnormal result. And the final unit test fails due to an infinite loop that assumes subnormal numbers are treated as non-zero.

Restrict Skia Tests. Restrict-Skia fails seven unit tests but passes all rendering tests. The failing unit tests match the failures of Full caused by the removal of subnormals. The remaining two unit tests fail due more limited range of Restrict.

Fixing Tests. A six line change to the `SkFloatToHalf` function (using an alternate, equivalent computation) removes one failure leaving only four failures for Full and six failures for Restrict. Furthermore, a four line change eliminates the infinite loop found in the last failure, safely discarding inputs that underflow to zero. While this change removes the infinite loop and allows the test to complete, it still reports an error because CTFP cannot handle subnormal numbers. However, the fact that Skia passes all the rendering tests demonstrates that CTFP's semantic changes provide enough to power a robust rendering library.

5.4 CTFP Has Acceptable Overheads

Micro-benchmark on Normals. We measure the overhead of CTFP calls compared to traditional floating-point operations on normal operands. We run each operation with two normal operands, and measure the time that each takes using the method from Section 5.1.[8] The first two columns of Figure 9 illustrate CTFP's overhead for each operation. Restrict CTFP is approximately twice as fast as the Full version for all operations except square root. This is because the implementation of square root is identical between the two versions as range restriction is unnecessary. Division is much slower than the other operations because it requires far more special cases.

In general, we find that the vectorized versions of each operation (except square root on doubles) outperforms the scalar version by up to 15% despite performing additional computation. This speedup is due to poor code generation by the compiler backend—values are often passed back and forth between SIMD registers and integer registers instead of computing exclusively on SIMD registers. To take advantage of this, all scalar operations call the vectorized version by leaving the remainder of the register uninitialized. Although it may seem unsafe to operate on uninitialized values, CTFP will safely protect every value.

[8]CTFP should take the same amount of time on both normals and subnormals, but traditional floating-point computations are considerably slower on subnormals.

Op	Restrict	Full	Subnorm
Float add	4.0x	9.8x	1.0x
Double add	4.0x	9.9x	1.0x
Float mul	4.1x	10.1x	40.2x
Double mul	4.2x	10.1x	40.4x
Float div	14.5x	31.3x	20.0x
Double div	12.2x	26.7x	17.1x
Float sqrt	9.5x	9.5x	15.1x
Double sqrt	9.5x	9.5x	14.8x

Bench	Restrict	Full
milc	2.0x	4.9x
namd	4.0x	12.6x
soplex	1.7x	3.5x
povray	3.3x	8.4x
lbm	8.0x	28.8x
sphinx3	4.0x	10.7x

Figure 9: Overhead introduced by CTFP on micro-benchmarks (left) and Spec CPU2006 benchmarks (right). The *Subnorm* column shows the overhead of baseline floating-point computations on subnormal inputs compared to normal inputs.

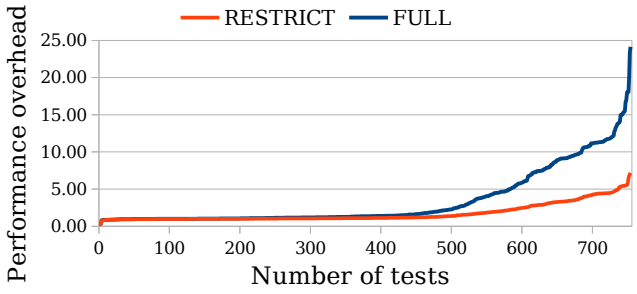

Figure 10: Overhead using 755 CPU performance tests for Skia. The chart plots the normalized performance overhead (y-axis) against the number of tests (x-axis). The blue line charts the overhead of Full, the red line charts the overhead of Restrict.

Micro-benchmark on Subnormals. We also compare CTFP's performance to baseline floating-point operations on subnormal inputs. [9] This comparison asks how the overhead of subnormals in traditional floating-point operations compares to the overhead of eliminating them these operations in CTFP. As Figure 9 shows, for addition, multiplication, and square root, the overhead of computing with subnormals is *far worse* than eliminating them. However, the overhead introduced for division is roughly 50% slower than operating on subnormals. Even so, the division operation protects against values beyond subnormals (e.g., zero and `NaN`).

Macro-benchmark. We compare CTFP's performance to baseline performance on the SPECfp benchmarks. The performance penalty on Spec sometimes exceeds that on micro-benchmarks because CTFP's decorations hinder optimizations and efficient machine code generation. Even so, the Restrict decorations add an overhead of between 1.7x-8.1x on floating-point intensive code, and the Full decorations add an overhead of 3x to 29x.

We also compare the performance overhead of CTFP using all 654 of Skia's CPU rendering tests from the `nanobench` tool. Figure 10 presents the performance of CTFP using a CDF that shows the overhead (normalized to base runtime) against the number of tests that fall below that performance threshold. Most tests (> 90%) incurred less than 3.7x slowdown with Restrict and less than 10x slowdown with Full.

[9]CTFP's performance on subnormals is unchanged, since CTFP is constant-time.

6 LIMITATIONS AND FUTURE WORK

Our CTFP approach has several limitations. Some of these are inherent to our approach, while others can be addressed in future work.

Control Flow Related Timing Channels. CTFP only applies to handling individual FP operations that have exploitable timing variability across different operands; it does not consider the orthogonal problem of information leakage via control flow and memory access. A complete protection mechanism would have to combine solutions for both kinds of timing channels, potentially leveraging some of the techniques from [33, 34].

Scaling Verification. While the SMT solver proved to be invaluable in unearthing tricky corner cases, we were unable to use it to verify the two most complex transformations without resorting to assumptions: multiplication and division (Section 4). Additionally, it would be desirable to verify all the 64-bit versions of the transformations. SMT solvers have only recently started supporting floating-point decision procedures, which ultimately use "bit-blasting" to reduce all queries down to propositional SAT formulas. We are optimistic that with more work, the solvers will be able to scale up to handle more complex transformations.

Subnormal-Sensitive Applications. While we have demonstrated that CTFP preserves the functionality of real-world applications like SPEC and Skia, there are many, many more applications—some of which may require exact IEEE-754 semantics. In future work, it would be interesting to support these applications with a new version of CTFP that transforms code so that all FP operations safely operate over subnormals (like the Escort system [34]).

Full Truncation of `FLT_MIN`/`DBL_MIN` to Zero. Due to double rounding in Full, multiplication and division may truncate an output of `FLT_MIN`/`DBL_MIN`—the smallest normal floating-point number—to zero. Newer CPUs support the FMA (fused multiply-add) instruction that could eliminate the undesired double rounding and restore correct calculation of all, non-subnormal operations.

Optimizing CTFP Output. The current code generated by CTFP contains a number of optimization opportunities. For example, when Full is applied the expression `a+1`, both the operands `a` and `1` are replaced by zero if they are subnormal—however, the constant value `1` will never be subnormal and therefore the check may be removed. Further work may safely optimize CTFP code while still protecting against timing channels.

7 RELATED WORK

Next, we survey recent related work on timing channels and their prevention, particularly focusing on systems related to floating-point computations.

Timing Channels. Kocher's seminal paper [24] was one of the first to consider the security implications of covert timing channels, in particular timing channels that arise due to data-dependencies. Since then, many attacks against real world systems have leveraged such timing attacks [4, 17, 19, 25, 26, 30, 32, 38].

FP Timing Channels and Defenses. Andrysco et al. were the first to demonstrate the dangers of data-dependency of floating-point operations by carrying out timing attacks against Firefox and Fuzz [4]. To address these timing attacks, they presented a fixed-point library, `libfixedtimefixedpoint`(FTFP), implemented in software. FTFP has significant performance overhead, but was mechanically verified to be constant time [3]

In the context of crypto systems, Coppens et al. [15] and Cleemput et al. [13] introduced the idea of using program transformations to *pad* variable-latency instructions, or to run them in parallel with other instructions to force variable-latency instructions to run in constant-time. The Escort system of Rane et al. [34] builds upon this padding scheme to address the FP attacks of [4]. Escort executes potentially deadly FP operations (which may involve subnormals) in a parallel SIMD pipeline with a dummy subnormal operation. Assuming the SIMD instruction retires only when *both* operations are complete, Escort ensures that all FP operations are effectively slowed down to operate at the (fixed) speed of subnormal computations.

Kohlbrenner and Shacham [25] demonstrated the danger of Escort's hardware-based padding by finding new special values that affect the latency of instructions, thereby circumventing the Escort system. Intel's flush-to-zero (`FTZ`) and denormals-are-zero (`DAZ`) CPU flags can be turned on to disable computations on subnormals; Chrome, for example, relies on these flags when calling into the Skia graphics library to address timing attacks [11]. Unfortunately, these CPU flags need to be very carefully enabled and disabled on a per library (function) basis. Kohlbrenner and Shacham [25] demonstrate that managing the flags can be error-prone, leaving the system vulnerable to attacks. Moreover, like the Escort's hardware-based method, `FTZ` and `DAZ` cannot prevent attacks that use other (non-subnormal) special values. Nevertheless, as a defense-in-depth, CTFP can be easily be modified to enable these CPU flags during our program transformation pass.

Other Covert Digital Channels. To prevent attacks that leverage timing channels, it is insufficient to solely ensure that instructions are constant-time. Timing-attack vulnerabilities can arise from program control flow, memory access patterns, and so on. Previous solutions address these attacks via program transformations [2, 8, 15, 28, 33, 34], clever usage of hardware, e.g., Intel's hardware transactional memory [20, 35], performance counters [14, 37], or partitioned caches [31, 32]) even black-box, whole system mitigation techniques [5]. While many of these techniques are orthogonal—Askarov et al.'s technique being the one exception that can be used to tackle timing attacks more generally—we believe that CTFP would be especially well-suited to be used alongside the control flow and memory transformations of Racoon [33] and Escort [34]. Both Racoon and Escort tackle leaks via control flow and memory access patterns, but rely on constant-time elementary floating-point operations.

SMT Verification of FP Optimizations. Alive-FP [27] and LifeJacket [29] both verify floating-point optimizations at the LLVM IR level using an SMT solver. Both focus on proving total semantic preservation. We, in contrast, translate our LLVM Restrict and Full operations to SMT formulas in order to verify *safety*: that no deadly values appear as the inputs or outputs of FP operations, and (partial) *equivalence*: that semantics are preserved only for non-dangerous or non-deadly values respectively. Escort [34] uses an SMT solver to determine which of a program's FP operations

are guaranteed to operate on non-subnormal values. These operations may be safely left unmodified. While they only consider subnormals, it would be natural to extend their tool to additionally consider other unsafe special values. Thus, we envision an extension to CTFP that uses symbolic execution to automatically eliminate the decorators when it is safe to do so.

8 CONCLUSION

Many systems rely on floating-point operations to run in constant time for security. Unfortunately, hardware floating point units exhibit timing variability for various input classes (e.g., subnormals, zeros, infinities, and `NaN`s) and solutions that repurpose various hardware features (e.g., CPU flags such as `FTZ` and `DAZ`) have not held up. We present CTFP, a software-based approach that addresses floating-point timing channels in an extensible and mechanically verified fashion. CTFP translates floating-point computations to use our constant-time operations (e.g., `fadd` and `fmul`), which either eliminate unsafe classes of values or emulate computation on such values in software, in constant time. CTFP breaks IEEE-754 semantics to ensure reasonable performance. But, importantly, CTFP does not break the semantics of real-world applications.

ACKNOWLEDGEMENTS

This work was supported in part by gifts from Cisco and Mozilla, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the National Science Foundation under Grant Number CNS-1514435.

REFERENCES

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

[2] J. Agat. Transforming out timing leaks. In *POPL*, 2000.

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

[4] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *SP*, 2015.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM CCS*, 2010.

[6] I. S. Association. IEEE Standard for Floating-Point Arithmetic. Technical report, 2008.

[7] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[8] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *ENTCS*, 2006.

[9] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *ARITH*, 2015.

[10] Chromium. Issue 615851. https://bugs.chromium.org/p/chromium/issues/detail?id=615851, 2016.

[11] Chromium. Issue 615851, security: Timing attack on denormalized floating point arithmetic in svg filters circumvents same-origin policy. https://bugs.chromium.org/p/chromium/issues/detail?id=615851, 2016.

[12] Chromium. Issue 2179003003. https://codereview.chromium.org/2179003003, 2016.

[13] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *TACO*, 2012.

[14] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: An empirical study of timing channels on sel4. In *ACM CCS*, 2014.

[15] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *SP*, 2009.

[16] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*.

[17] C. P. García and B. B. Brumley. Constant-time callees with variable-time callers. In *USENIX Security*, 2017.

[18] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 1991.

[19] J. Großschädl, E. Oswald, D. Page, and M. Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC*, 2009.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 2017.

[21] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In D. Wagner, editor, *USENIX Security*, 2011.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[23] Intel. Setting the FTZ and DAZ flags. https://software.intel.com/en-us/node/678362.

[24] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[25] D. Kohlbrenner and H. Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, 2017.

[26] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *ACM CCS*, 2013.

[27] D. Menendez, S. Nagarakatte, and A. Gupta. Alive-FP: Automated verification of floating point based peephole optimizations in LLVM. In *International Static Analysis Symposium*, 2016.

[28] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005.

[29] A. Nötzli and F. Brown. LifeJacket: verifying precise floating-point optimizations in LLVM. In *International Workshop on State Of the Art in Program Analysis*, 2016.

[30] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *CryptographersâĂŹ Track at the RSA Conference*, 2006.

[31] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR*, 2005.

[32] C. Percival. Cache missing for fun and profit, 2005.

[33] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[34] A. Rane, C. Lin, and M. Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *USENIX Security*, 2016.

[35] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[36] K. D. Smith, J. J. Jewett, S. Montanaro, and A. Baxter. Pep 318 – decorators for functions and methods. https://www.python.org/dev/peps/pep-0318/, 2003.

[37] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*, 2013.

[38] P. Stone. Pixel perfect timing attacks with html5. *Context Information Security (White Paper)*, 2013.