UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards Verified, Constant-Time Floating-Point Operations

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Marc Edward Andrysco

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Micheal Holst
Professor Ryan Kastner
Professor Sorin Lerner
Professor Deian Stefan

2019

The dissertation of Marc Edward Andrysco is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thank you so much to everyone who made this PhD possible. I especially want to thank my fiancé, Suzanne Dunai, for all the support throughout the degree. A special thank you to my advisor, Ranjit Jhala; my co-authors Fraser Brown, David Kohlbrenner, Sorin Lerner, Keaton Mowery, Andres Nötzli, Hovav Shacham, and Deian Stefan.

Chapter 2, in full, is a reprint of the material as it appears in ACM Computer and Communications Security (CCS) 2018. Andrysco, Marc; A. Nöetzli; Brown, Fraser; Jhala, Ranjit; Stefan, Deian, 2018. The dissertation author was a primary investigator and a primary author of this paper.

Chapter 3, in part is currently being prepared for submission for publication of the material. Andrysco, Marc; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this material.

VITA

2011	Bachelor of Arts, Mathematics, Case Western Reserve University
2011	Bachelor of Science, Computer Engineering, Case Western Reserve University
2013-2019	Graduate Student Researcher, University of California San Diego
2019	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

M. Andrysco, A. Noetzli, F. Brown, R. Jhala, D. Stefan "Towards Verified, Constant-time Floating Point Operations." ACM Computer and Communications Security (CCS), Oct 2018

M. Andrysco, R. Jhala, S. Lerner "Printing Floating-Point Numbers: An Always Correct Method." Principles of Programming Languages (POPL), Jan 2016

M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham "On Subnormal Floating Point and Abnormal Timing." IEEE Security and Privacy (Oakland), May 2015

ABSTRACT OF THE DISSERTATION

Towards Verified, Constant-Time Floating-Point Operations

by

Marc Edward Andrysco

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Ranjit Jhala, Chair

Floating-point computation exhibits significant runtime variation based on input parameters with some inputs executing over 100 times slower. The timing differences are so severe that attacks have successfully broken privacy guarantees of real systems (e.g. browsers). My thesis presents a defense against floating-point timing variability called CTFP – Constant-Time Floating-Point. The CTFP approach avoids all known fast and slow paths by surrounding every operation with special code that guarantees no dangerous inputs or outputs are observed. CTFP provides five constant-time implementations that trade-off between performance and correctness. Through these implementations, CTFP provides a principled method for defending against floating-point timing attacks.

Chapter 1

Introduction

Side-channels are an evolving and difficult area of computer security. The recent Meltdown and Spectre vulnerabilities demonstration just how difficult it is for security researchers to predict and defend against side-channels [21] [19]. Timing-channels are a common form of side-channels that rely on the execution time of specific pieces of code. Given a vulnerable timing-channel, an attacker may learn private information by measuring how long code takes to run. This presents a challenge to developers who must now reason about the running tmie of any code that interacts with sensitive data. Most attacks have relied on software fast-paths: the program will run faster code based on a secret value [27]. Defending against these attacks have focused on guaranteing that any code running against private information always executes the same instructions, eliminating the fast-path. Unfortunately, these defenses are not sufficient in all cases. Many instructions, such as arithmetic on floating-point numbers, may not execute in constant-time. When operating on certain values, a floating-point instruction may execute over 100 times slower than on normal values [3]. Exploits have been demonstrated that take advantage of floating-point timing variability. The difficulty in defending against these attacks provides an opportunity for newer tools

In this thesis, I provide an analysis of floating-point timing variability and techniques for

eliminating floating-point timing-channels:

- a characterization of floating-point timing variability
- a set of techniques emulating operations on dangeorus values
- a set of tools for defending against floating-point timing attacks
- a technique for improving performance of the above defenses

1.1 IEEE-754: Floating-Point Numbers

IEEE-754 is a widely-supported specification that completely details floating-point numbers and their operations. The standard provides both the ubiquitous 32-bit float s and 64-bit doubles in addition to the less used 16-bit half-precision and 128-bit quad-precision formats. 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, floating-point numbers are encoded with three parts: a sign bit, a w-bit exponant, and a fixed-width significand. Using this encoding, the value of a normal floating-point number is specified by the formula

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

where $bias = 2^{w-1} - 1$ and both the exponent and signficand 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 (NaNs) 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; they are computed according to

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

Most processors that supporting floating-point also support a flag for disable subnormal values. x86-based processors provide a pair of flags: FTZ (flush-to-zero) that replaces subnormal outputs with zero and DAZ (denorms-are-zero) that treats an subnormal input at zero. ARM processors use the `FE_IEEE_FLUSHZERO` flag for disabling subnormals.

1.2 Timing Attacks on Browsers

Browsers are a common and highly valued target for hackers. Users often perform activities in the browser that access sensitive information such as online banking or accessing email and social media. An attacker may host a malicious website that can execute arbitrary Javascript code. Additionally, an attacker may use an iframe to embed a target website containing senstivie information into the malicious website. Despite displaying the malicous website and target website side-by-side, a secure browser will prevent the malicous website from access any data in the target. Unfortunately, enterprising users have repeatedly demonstrated methods for exfiltrating data from the iframe.

1.2.1 History Sniffing

History sniffing is a technique where a malicious website identifies whether or not the user has visit a specific URL. Because links are rendered differently based on whether or not is was visited, early history sniffing would generate a large set of links to website and inspect

the style [18]. Browsers have since protected against this simple version of history sniffing by always returning the unvisited style for a link [6].

Paul Stone demonstrated a method for performing history sniffing by indirectly detecting the link style using a timing-channel [27]. The attack is based off the insight that specific browser features can perform computation on private data and these computations may take a variable amount of time. The execution time is measured using the api requestAnimationFrame that notifies the website when all rendering is cmolpeted. Put together, the attack uses the following steps:

1. Set the style for visited links to be computation intensive (e.g. apply text-shadow)
2. Insert a number of links to the target website
3. Use requestAnimationFrame to time how long each frame takes
4. Analyze timing to determine if the link was visited

The browser first renders the link as unvisited, which is an operation that completes quickly. Once the browser determines the link is visited, it applies the new style, which is an operation that is very slow. A visited link is therefore detected by observing a longer than usual delay before the requestionAnimationFrame callback is executed.

1.2.2 Pixel Stealing via Software Fast-Path

Expanding on the history sniffing attack, Paul Stone also demonstrated a method for extracting pixel information from a target iframe [27]. The attack is based on the fact that certain SVG filters have different execution time depending on the source image. SVG filters are a feature of browsers that allow websites to apply a variety of operations over image data such as gray-scaling or blurring. Despite the fact that a website cannot read inside an iframe, SVG filters allow them to perform computation over the pixel data. Of particular interesting are filters

such as feMorphology that uses a software fast-path to make processing an all white image faster than processing an image of random pixel values.

1. Embed the target website using an iframe
2. Expand a single pixel of the iframe to fill a 100x100 element using SVG filters
3. Use SVG filters to set the element to be completely black or white
4. Multiply the image by an image of random noise, so the image is now either white or noise
5. Apply the feMorphology filter to the iamge of either white or noise
6. Time how long the filter takes to complete using requestionAnimationFrame
7. Analyze the timing with a white image being fast and a noisy image being slow

The above process is repeated for every desired pixel in the iframe until the malicious website has reconstructed the iframe data. The pixel stealing attack were addressed by making all rendering effects (e.g. SVG filters) to not use any software fast-paths.

1.2.3 Pixel Stealing via Data-Depedent Floating-Point

Andrysco et. al. further devloped the attack by Paul Stone by demonstrating that floating-point timing variation can be leveraged to perform timing-channel attacks. They demonstrated a proof-of-concept that stole pixels from cross-origin iframes using only data-dependent timing variation – that is, there no software slow- or fast- paths are used. The timing variation occurs solely due to the execution time of floating-point instructions based on the operands.

The attack uses the same process as Paul Stone's pixel stealing setup with two changes. First, multiplying by the random noise is unneeded; instead, the final SVG filter is applied directly to the elements that is either all white or all black. Second, the SVG filter feConvolveMatrix

is applied to the element using a specially crafted kernel matrix consisting of subnormal floating-point numbers. The attack relies on the fact that most modern CPUs are exceptionally slow when operating on subnormal operands but remain fast if either operand is zero. If the pixel is black, the value is exactly `0.0`, so the operation `0.0 * subnorm` is fast, taking only a few clock cycles to completel. If the pixel is white, the value is `1.0`, so the operation `1.0 * subnorm` is very slow, taking over 100 clock cycles on most CPUs. Combined with the fact that the multiplication is performed millions of times, the timing difference between the two operations can vary from a single millisecond if the image is black to hundreds of milliseconds if the image is white.

Despite initial efforts to address the floating-point timing-channel attacks, Kohlbrenner and Shacham showed that Chrome, Firefox, and Safari remained vulnerable to pixel stealing attacks [20]. For Firefox, they ported many of the SVG filters, including the vulnerable feConvovleMatrix filter, to a fixed-point that exclusively uses integers. Unfortunately, the lighting filters (e.g. feSpecularLighting) were not ported to fixed-point and therefore remained vulnerable. Chrome attempted to close the timing channel using the FTZ+DAZ flags to eliminate the slowdown caused by subnormal numbers. Under normal circumstances, a filter is either applied by the CPU with the FTZ+DAZ flags set or by the GPU where subnormal timing variablity is not present or useful. By using a very large kernel matrix with feConvolveMatrix, Chrome would abort processing on the GPU and fall back to computing the filter on the CPU. This specific fallback would not enable the FTZ+DAZ flags and therefore failed to defend against the pixel stealing attack. Safari remained vulnerable to a reworked version of the original floating-point pixel stealing attack.

1.3 Prior Defenses against Timing Variability

Existing defenses again floating-point timing variability have focused on eliminating the difference when operating on subnormal numbers versus normal numbers. Defenses either aim to eliminate subnormal numbers to prevent the slowdown or they seek to slow down all normal operations to the speed of subnormals.

1.3.1 Using FTZ+DAZ Flags

The x86 family of processors include a pair of flags called FTZ+DAZ that eliminate subnormal numbers. The FTZ (flush-to-zero) flag instructs the CPU to round any operation that would *output* a subnormal number to instead round the result to zero. The DAZ (denorms-are-zero) flag instructs the CPU to consider any subnormal *input* to be zero. Combining both of the flags eliminates the possibility that the CPU will compute on subnormal numbers, and therefore the program will not experience any slowdown from their presence.

1.3.2 SIMD Packing with Subnormals

Rane et. al. demonstrated their system Escort for reducing timing variability by slowing down all normal operations to the speed of subnormals [25]. Every floating-point operation was instrumented such that the desired operation is packed into a SIMD register with subnormal operands. If the original data inputs are normal, the operation will be slowed down to the speed of the subnormal operands. While the timing difference is substantially reduced, Kohlbrenner et. al. was able to show that at least some portion of the instruction is still computed serially and exhibits a timing difference [20].

1.3.3 Timing Variability of Special Values

Kohlbrenner et. al. was also able to demonstrate that certain special values produce small but measureable *speedup* of floating-point instruction [20]. In particular, they observed speedups in division in the following cases:

- Dividing with any special value (`0.0`, Inf, NaN)
- Dividing by powers-of-two
- Square root of any special value (`0.0`, Inf, NaN)

These speedups are observed both when FTZ+DAZ flags are set and when present in SIMD operations. Therefore, neither FTZ+DAZ or Escort are sufficient to protect against special values.

Chapter 2

Constant-Time Floating-Point

2.1 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 [14]. 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 [3]. For Firefox, they implement Stone's *pixel stealing* attack [27] by measuring page rendering time. To break Fuzz [15]—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 [11, 17]. 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 [10]. The Escort system [25] 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 [20]. 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 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 [20]. 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 2.2.1), 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.

2.2 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 2.2.1). Then, we experimen-

tally identify the different classes of values that lead to timing channels (Section 2.2.2). Finally, we sketch our approach to addressing the timing channels by decorating FP operations to execute in constant-time (Section 2.2.3).

2.2.1 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 [5]. 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 [5]; 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 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 [3]. For example, Andrysco et al. [3] show how subnormals can be exploited to break Firefox's same-origin guarantees by using them to implement Stone's *pixel stealing* attack [27]. 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.

2.2.2 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

Operation	Subnormal	Negative	Powers-of-two	Powers-of-four	`NaN`s, `Inf`s, and zeros
Addition/Subtraction	✗	✓	✓	✓	✓
Multiplication	✗	✓	✓	✓	✓
Division	✗	✓	✗	✗	✗
Square root	✗	✗	✓	✗	✗

Figure 2.1: 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.

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 [3, 25, 20] 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 2.2 presents the results of our measurements for different x86 CPUs. (In Section 2.5.1, 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 2.1, 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. [3] and, more recently, Kohlbrenner and Shacham [20], 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 2.2.1). 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 2.2.1). 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.

Processor	$+$ subnormal	$+$ special	$\times$ subnormal	$\times$ special	$\div$ subnormal	$\div$ 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 2.2: 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.

2.2.3 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* 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 instruc-

tions 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 3.6). 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 3.6).

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

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

`fmul` on 4 and the dummy value, and overwrites the result with infinity—all in constant-time. In Section 2.3, 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 2.3 and Section 2.4 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 [27]) 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 2.3.4). 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, 22, 24]) are complimentary to CTFP.

2.3 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 $\texttt{tx}_1$, $\texttt{tx}_2$, ..., each of which accounts for a subset of unsafe inputs. Inspired by Python [26], we write `tx @ op` for the result of applying the decorator `tx` to the operation `op`, and $\texttt{tx}_1$ `@ ... @` $\texttt{tx}_n$ `@ 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 1s) or false (i.e. all 0s). 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.

[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.

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

2.3.1 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:

```
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.

Table 2.1: 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.

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

2.3.2 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.

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 2.1 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 [20] 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 2.3.

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.

- 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

[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 2.3: RESTRICT-division uses decorators that blind large divisors, small dividends,

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.

2.3.3 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 $|a \texttt{ op } b| \geq M$, 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 2.4 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

[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 2.4: FULL addition, subtraction (omitted), and multiplication (omitted) underflow subnormal inputs and any normal inputs that produce subnormal outputs.

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 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 subnor-

mal 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 2.3.2 since the square-root of a normal number is guaranteed to be normal.

2.3.4 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 `ctOp` = $\texttt{tx}_1$ `@` ... `@` $\texttt{tx}_n$ `@ 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.

2.4 Verifying CTFP with SMT

As demonstrated by the decorators in Section 2.3, 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 $a = \mathit{true}, b = \mathit{false}$ 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 [13] as our SMT solver because it is one of the few solvers that currently support the floating-point theory.

2.4.1 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 [16]. Next, we show how we can use contracts to specify the two key properties that CTFP ensures

- *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

```
;; 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 2.5: Contract for the primitive 32-bit addition.

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 2.5. 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 2.6 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)
```

```
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 2.6: LLVM code for RESTRICT addition.

```
                    (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 `underflow`ed (to zero) inputs. We precisely define the semantics of underflowing as the SMT predicate:

```
(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 Table 2.1.

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.

2.4.2 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.

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 2.5. 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-

[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.

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 $a \diamond b \vee$ `isNaN`$(a) \vee$ `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 $\infty - \infty =$ `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 2.3.3).

- 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. Table 2.2 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.

Table 2.2 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 val-

Table 2.2: 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.

	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

ues 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 dectorators 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.

2.5 Evaluation

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

- (Section 2.5.1) execute in constant-time?
- (Section 2.5.2) change the input-output ranges as specified?

- (Section 2.5.3) break the behavior of applications?
- (Section 2.5.4) 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.

2.5.1 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,`

[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 [25]. The *Subnormals* column in Figure 2.7 gives a rough estimate of the best-case overhead for Escort.

`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 2.2 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.

2.5.2 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 pre-

[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`**.

serve the semantics of the underlying operation when neither its inputs nor its output are sub-normal (Section 2.4). 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).

2.5.3 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 [20]. 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.

2.5.4 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 2.5.1.[8] The first two columns of Figure 2.7 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.

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 2.7 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

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

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

Table 2.3: 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.

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

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 2.7 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.

2.6 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,

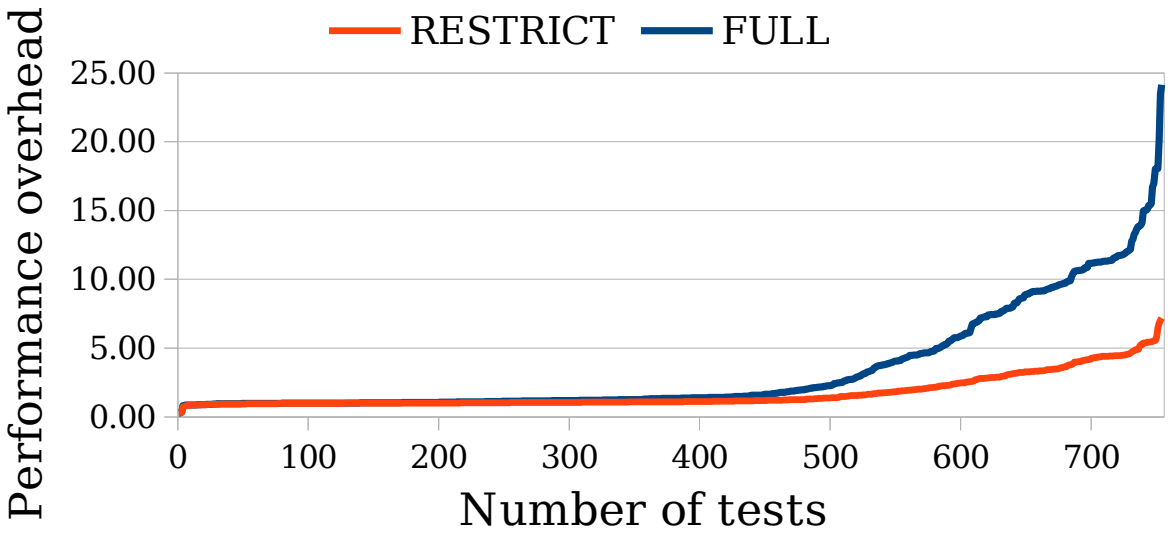

Figure 2.7: Overhead using 755 CPU performance tests for Skia. The chart plots the normalized performance overhead (y-axis) against the number of tests (x-axis). The blue line charts the overhead of FULL, the red line charts the overhead of RESTRICT.

potentially leveraging some of the techniques from [24, 25].

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 2.4). 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 [25]).

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.

2.7 Acknowledgements

Chapter 2, in full, is a reprint of the material as it appears in ACM Computer and Communications Security (CCS) 2018. Andrysco, Marc; A. Nöetzli; Brown, Fraser; Jhala, Ranjit; Stefan, Deian, 2018. The dissertation author was a primary investigator and a primary author of this paper.

Chapter 3

Optimizing CTFP

3.1 Introduction

Modern computing approximates real-valued computations through floating-point computation. The IEEE-754 specification details the concrete format for floating-point numbers (e.g. 32-bit `float` and 64-bit `double`) and operations on these values (e.g. multiplciation and square root) [12]. Hardware implementors have dutifully followed the standard implementing IEEE-754 floating-point in virtually all common CPU designs. Although the standard provides explicit detail of *what* results should be given, it does not specify exactly *how*, or more directly *when*, to give a result.

Hardware designers have therefore elected to optimize for the most common cases, often heavily sacrificing performance on edge cases. A special class of very small floating-point numbers called subnormals are particularly unoptimized, executing about 50x slower than typical inputs. When operating on sensitive data, it is possible for attackers to supply subnormal numbers in order to leak information through a timing side-channel [3]. Leaking of sensitive information was demonstrated on browsers through cross-domain iframes [2] and the security conscious database Fuzz [15].

Work on defending against leaking information via floating-point timing-channel has focused on five areas.

Use Integers/Fixed-Point. The entire issue of floating-point timing variability may be sidestepped by using an alternative real-value approximations (e.g. fixed-point). Unfortunately, integer arithmetic itself is not guaranteed to be constant-time and may exhibit its own timing variation. Furthermore, fixed-point values cannot represent the extremely large range that is provided by floating-point.

Enabled FTZ+DAZ Flags. x86-based processors provide a pair of CPU flags that treat subnormals as zero avoiding the extreme slowdown. These flags replace subnormal inputs with zero and returns a zero if the output would be subnormal. Because the flags are statefully stored by the CPU, the program must carefully ensure that the flags are always set. Chrome relies on the FTZ+DAZ flags to defend against timing variations in SVG filters, but their system was vulnerable in a certain corner case where Skia's hardware exeleration falls back to software rendering in a context where the flags were no longer set. Additionally, FTZ+DAZ does not protect against the small timing variations from other special values such as zero and infinity.

Move Computation to the GPU. Besides avoiding timing variation, moving comptuation to the GPU can greatly improve the speed of floating-point computation. While timing variation on multiplication is not present on an Nvidia GeForce GT 430, other operations (e.g. division) are not constant-time [20].

Slowing Computation using SIMD. Rane et. al. demonstrated a technique where all floating-point operations are performed in SIMD with subnormals packed beside the desired computation. Because the SIMD register contains a subnormal, the entire instruction is slowed down to handle the special case. Although this technique greatly reduces the amount of timing variability, there are still very small variations on the order of a single clock cycle. The resulting code runs much slower with an overhead of 32x on the Spec CPU2006 benchmarks.

Emulate Unsafe Operations. Andrysco et. al. demonstrated a pair of systems that elim-

inate operations on unsafe values via emulation. Both techniques use constant-time bitmasks to detect dangerous values ahead of time and replace them with safe alternatives. Replacement is suitable for special values (e.g. zero), but there are too many subnormal numbers to handle efficiently. Thus, the two constant-time systems aim to deal with subnormals by trading accuracy for speed. The `RESTRICT` implementation aggressively underflows small numbers to zero such that the output is always non-subnormal. The `FULL` implentation detects subnormal outputs ahead of time, and if the output is predicted to be subnormal, replaces the inputs with zero.

Our work builds on the emulation of unsafe operations by expanding the techniques used for subnormal numbers. Although both `RESTRICT` and `FULL` eliminate timing channels, both incur a significant overhead of up to 8x and 29x on Spec CPU2006. Our work is summarized in three contributions:

- `BASIC`: a simplified defense that only protects special values. Because handling subnormals incurs signicant overhead, `BASIC` achieves far better performance than other defenses. However, a proper protection against floating-point timing variability must combine `BASIC` with another techinque for handling subnormals. In this manner, `BASIC` serves as the foundation for building complete systems. The following two techniques build on top of `BASIC` to completely eliminate timing variations.

- `FAST`: an optimized version of range restriction for preventing subnormals. Values are aggressively underflowed, more so than `RESTRICT`. In doing so, future operations may be statically proven and not require protection. Eliminating the need to underflow provides a substantial performance benefit.

- `FLAGS`: a protection scheme for subnormals that forcefully asserts the FTZ+DAZ flags found in x86 CPUs. To prevent third-party code from disabling FTZ+DAZ, the flags are reset at every control flow into every function. The use of FTZ+DAZ allows programs to be fully protected from floating-point timing variability without limiting the range of

allowable values.

3.2 Background

In this section we give an overview of floating-point timing channel attacks and techniques for defending against such attacks. First, we provide an overview of floating-point numbers and their operations. Second, we characterize the timing-channels present in floating-point hardware and how these lead to exploitable attacks. Third, we cover previous work to defend against floating-point timing-channel attacks using emulation. Last, we give an overview of our new approach that improves on the performance and usability of previous methods.

3.2.1 Floating-Point Standard

The IEEE-754 standard details the semantics for floating-point numbers and their operations. Each floating-point number is built from three pieces: a sign bit, a u-bit significand, and a w-bit exponent. *Normal* numbers take the form

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

The significand is a strictly positive number in the range $[2^u, 2^{u+1}-1]$ (the most-significant bit is always one and not explicitly stored). The bias is fixed to $2^{w-u-1}-1$. The exponent falls in the range $[1, 2^{w-1}-2]$ (i.e., any bit pattern other than all zeros or all ones). The standard specifies a set of special values: positive and negative zero (all significand and exponent bits of are zero), infinity (all exponent bits are ones and the significand is zero), and NaN (all exponent bits are ones and the significand is non-zero). The standard also includes a special class of values called *subnormal* when the exponent is all zeros with a non-zero significand. In this form, the exponent is set to the fixed value 2^{w-u-1}, and the signficand loses the implicit bit and therefore falls in the

range $[1, 2^u - 1]$.

3.2.2 Timing Variation

Timing differences have been observed when performing operations on different input classes. In particular, mutliplication on subnormal numbers is orders of magnitude slower on x86 processor. When digging deeper, previous work has shown that other operations are vulnerable to small speedups on special values. These facts make floating-point operations a source of timing channels because observers can learn information about values based on the execution time.

Floating-Point timing channels have demonstrated to be effective sources of exploits. Andrysco et al. gave a proof of concept where cross-origin pixel data could be exfiltrated using floating-point timing variability in Firefox [3]. After initial attempts to close these timing channels, Kohlbrenner and Hovav demonstrated that the fixes applied to Firefox were insufficient, and they further provided proof of concept exploits for Safari and Chrome [20].

Every attack is based off the execution time of a private value multiplied by a subnormal number. If the private value is zero, the execution time is very fast at about 4 clock cycles. If the private value is non-zero (e.g. 1), the instruction is very slow at about 140 clock cycles.

3.2.3 Constant Time via Software Emulation

The defense called CTFP (constant-time floating-point) by Andrysco et. al. avoids computation on dangerous values by replacing them with safe values [3]. For example, the square root of NaN is *faster* than the square root of a normal number (e.g. `sqrt(NaN)` vs `sqrt(1.5)`). CTFP uses bitmasks to detect and replace these types of dangerous values. Because the substituted values will produce an incorrect output, the result itself must be replaced with the correct output. For example, figure 3.2 demonstrates a version of square root that protects against NaN

```
safeSqrt(v):
  t := if isNaN(v) then 1.5 else v
  r := sqrt(t)
  f := if isNaN(v) then NaN else r
  return f
```

Figure 3.1: Pseudocode for `safeSqrt` that protects against NaN. The input of NaN is detected and replaced with 1.5 on line 2. After performing the square root (line 3), the result is then fixed on line 4 by substituting in NaN if the input was originally NaN.

values.

Although process of detecting and precomputing the result is reasonable for special values, the number of subnormals makes the process impractical. Previous work has used underflow to prevent subnormals in one of two ways.

- The `RESTRICT` method that underflows enough values to guarantee that the output will never be subnormal. For example, the value `1e-155` is not subnormal, but multiplying `1e-155` by itself produces `1e-310`, a subnormal number. To solve this problem, `RESTRICT` simply underflows any values below `sqrt(DBL_MIN)` (about `1.5e-154`) which guarantees any multiplication will not produce a subnormal. In order to maximize the usable range of values, each operation (addition, multiplication, divisione, and square root) uses a unique set of underflow boundaries.

- The `FULL` method uses a prediction stage to safely determine if the output will be subnormal without performing the dangerous computation. If the output will be subnormal, the inputs are replaced with zero to return the underflow result of zero. Subnormal inputs are directly dealt with by immediately replacing them with zero.

3.2.4 Faster Constant-Time Operations

Our work uses two insights that improve performance of constant-time implementations. First, the FTZ+DAZ flags may be used to prevent slowdowns from subnormals in a princi-

pled manner that does not risk them being unset. Second, when restricting the range of usable floating-point values, ranges may be selected that allows some protections to be safely omitted without compromising the constant-time properties. The new techniques are summarized below:

- `BASIC` is a pared down constant-time implementation that ignores the timing variations of subnormal numbers. Because `BASIC` does not handle subnormals, it serves as the foundation for building *complete* constant-time implementations by handling the subnormal case.

- `FAST` aims to improve on the speed of `RESTRICT` by leveraging static analysis to remove unneeded underflows. The previous protection scheme, `RESTRICT`, naïvely applies underflows to every input of every operation to prevent timing variation. `FAST` reduces the number of protections by static analyzing the code to determine which may be omitted. Safety is preserved through an enhanced formed of interval arithmetic that statically proves the absense of subnormal values.

- `FLAGS` takes a different approach to removing subnormals by setting the FTZ+DAZ flags on every control flow into a function. Because third-party code can easily unset FTZ+DAZ, the `FLAGS` approach assumes any incoming path to the function is dangerous and must be protected. Sprinkling code to enable FTZ+DAZ creates a unique performance profile that works best on code with fewer function calls.

Each one of these techniques is described in detail in their respective sections.

3.3 Basic

This section presents the `BASIC` defense against floating-point timing variability. `BASIC` only protects against timing variation from special values and explicitly does not protect sub-

```
def safeSqrt(v):
  t := if v == 0.0 then t = 1.5 else v
  u := sqrt(t)
  r := if v == 0.0 then r = 0.0 else u
  return r
```

Figure 3.2: The pseudocode for square-root that protects only against positive zero. The input value of zero is replaced with 1.5 to prevent timing variation in the `sqrt` call. To provide the correct output if the input was zero, the output fixed by replacing it with zero.

normal values. In this way, it serves as a foundation for building complete defenses that protect against all known timing channels.

For a simple example, we will consider protecting the square-root operation. Square-root has timing variability for all special values (zero, Inf, NaN), negative numbers, powers-of-four, and subnormals. The `BASIC` version of square-root is identical to the previous defenses `FAST` and `FULL` with the exception that the subnormal underflow is omitted. Figure 3.2 provides pseudocode for square-root that protects against positive zero. The code is expanded by iteratively protecting against all special values. The full protection for powers-of-four is slightly different and described in the original CTFP defense [4].

Ignoring subnormals is very useful for programs that can set the FTZ+DAZ flags once on startup and can guarantee they will never change. This is possible in situation where you compile the final binary and have only trusted third-party libraries (or no imported code at all). It is not safe if the protected code is itself a library since the application code or other third-party may disable the FTZ+DAZ flags. For large codebases or projects that use untrused third-party code, correctly managing the flags can be difficult and lead to vulnerabilities [20]. Section 3.5 demonstrates a principles approach to guarantee that the flags are always enabled.

3.4 Fast

`FAST` is an optimized constant-time implementation built on two assumptions. First, most

```
t := uflow(a) * uflow(b)
r := uflow(t) * uflow(c)
```

Figure 3.3: The transformed code from `(a*b)*c` using `RESTRICT`. To ensure safety, `RESTRICT` must underflow both inputs to each multiplication. All values are single-precision floating-point numbers.

floating-point arithmetic does not utilize the entire range of floating-point values; instead, most programs operate on a very limited range of values. Second, by restricting the range of floating-point values to a conservative range, we can omit costly underflows that would otherwise be required. Combining these facts allows us to implement a constant-time method for performing floating-point arithmetic with far less overhead than previous techniques.

The full range of floating-point values is extremely large: `DBL_MAX` is greater than 10^{308} and `FLT_MAX` is greater than 10^{38}. For context, the size of the observable universe is about 10^{36} nanometers, a value that still falls within the range of single-precision floating-point. Despite the enormous range, most floating-point programs only operate within small portion of the overall space. Furthermore, it is best practice to avoid extreme values since operations with large magnitude differences are likely to incur significant precision loss.

To take advantage of these facts, `FAST` restricts computation to a small subset of all values $[min, max]$. The ranges are $[2^{-24}, 2^{24}]$ for `floats` and $[2^{-53}, 2^{53}]$ for `doubles`. Any values below `min` *may* be underflowed to zero (or negative zero), and values above `max` *may* be overflowed to infinity (or negative infinity). The key concept behind `FAST` is that values will only be replaced when they are statically determined to be possibly unsafe (i.e. non-constant-time).

3.4.1 Basic Example

As a motivating example, consider the very simple program `(a*b)*c` using `RESTRICT` with single-precision numbers. The transformed code is in figure 3.3. To recap how `RESTRICT` works, each operation is prefixed with an underflow so that any dangerous values below `1.1e-19`

are replaced with zero. The output of the operation is guaranteed to generate a normal number since multiplication of the lower bound always yields a normal number. We once again underflow both inputs to the second operations, `t` and `c`. This results in four underflows for only two floating-point operations. It would be convenient if we could use a smarter technique that can reduce the need to repeatedly underflow.

What would happen if we chose a more restrictive limit of `1e-10` and applied it to all the inputs? Assuming the worst case where every input is exactly `1e-10`, the smallest value of `t` is `1e-20` and the final result is at least `1e-30`, which is clearly a normal number. By performing an over-restriction of the inputs, we have eliminated the need for the underflow on the intermediate value `t`. This leaves a number of questions. What is a reasonable restriction? How many operations does this eliminate? How do we automate the elimination of underflows? We will tackle the automation problem first before touching on how we select a safe range.

3.4.2 Tracking Subnormal-ness

Our goal is to statically determine which floating-point values do not need protection. Although it may first appear that an interval analysis pass is adequate, the simple example of `r=a-1` with $a \in [1,2]$ demonstrates that it lacks the necessary precision. The result interval $r \in [0,1]$ clearly includes every positive subnormal value, but if we consider when `a` is either 1 or its successor 1.00000012, the result is either exactly zero or `1.2e-10`. Neither value is subnormal, and the values only get larger as we move up the interval. Despite the interval covering subnormal numbers, the values are too spaced out to take on a subnormal value.

The spacing is tracked by determining the least significant bit (LSB) that a value can occupy. For the binary number 1.01$\underline{1}$, the LSB is underlined and takes the value -3 because it is three places to the right of the decimal. When applied to intervals, the LSB represents the last bit any value in the range can occupy. Because all bits to the right of the LSB must be zero, every value in the range is an integer multiple of 2^{LSB}. For single-precision numbers, all subnormals

fall below 2^{-126}. Thus, for a number to be subnormal, it is necessary (but not sufficient) to have an LSB below -126. The corollary is that any value with an LSB of at least -126 must not be subnormal.

`FAST` uses an analysis that combines standard intervals with an LSB into an *augmented interval*. The argumented interval is represented by the triple $V = (\underline{v}, \bar{v}, LSB_v)$. A value v in the range V satisfies the equation $\underline{v} \leq v \leq \bar{v}$ and is an integer multiple of 2^{LSB_v}. The endpoints of the range $\underline{v}$ and $\bar{v}$ are floating-point numbers, and they may include the values $-\infty$ or ∞. The LSB is always an integer.

The below rules provide sound approximations of the LSB for floating-point operations. The proofs that each operation is correct rely on the fact that all numbers can be written explicitly in terms of the significand (v_{sig}) and LSB (LSB_v) using the formula $v = v_{sig} 2^{LSB_v}$.

Unknown Values. The LSB of any unknown value is given as least-significant bit of the floating-point type. For example, the smallest LSB of a `float` is -149 because every `float` (including subnormals) is an integer multiple of 2^{-149}.

Constants. The LSB is directly found by directly computing the LSB of the constant. The LSB of `0.0` (and `-0.0`) is infinite – this bizarre definition stems from the fact that no matter how large an LSB we choose, the value zero can always be written as $0 * 2^{LSB}$. Otherwise, the LSB is computed by simply counting the position of the right-most 1-bit.

Addition and Subtraction ($r = a \pm b$). Due to the fact that addition (and subtraction) can only propagate digits to the left, the LSB of the output can be no worse than either input $LSB_r = \min(LSB_a, LSB_b)$. More formally, the result of addition is $r = a_{sig} 2^{LSB_a} + b_{sig} 2^{LSB_b}$. Because both a_{sig} and b_{sig} are integers, the result must be an integer multiple either 2^{LSB_a} or 2^{LSB_b}, whichever is smaller.

Multiplication ($r = a * b$). By writing out the expansion for both floating-point numbers, the result of a multiplication is $r = a_{sig} * b_{sig} * 2^{LSB_a + LSB_b}$ where the the LSBs added. It may be the case that the term $a_{sig} * b_{sig}$ would have zero bits on the right-hand side, but $LSB_r \geq LSB_a + LSB_b$

```
type AugIval = (lo:FP, hi:FP, lsb:Int)
type Fact = (ivals:List AugIval, nan:Bool)
```

Figure 3.4: Type declarations for augmented intervals (`AugIval`) and facts (`Fact`).

is a sound overapproximation.

Square Root ($r = sqrt(a)$). Using the explicit formula for the input gives $r = a_{sig}^{1/2} * 2^{LSB_a/2}$. If LSB_a is even, the LSB_r is exactly $r/2$. When LSB_a is odd, we pessemistically select $LSB_r = \lfloor r/2 \rfloor$. Like multiplication, the square root operation may create a result significand r_{sig} with zero bits on the right, but these may be ignored to provide a correct overapproximation.

Division ($r = a/b$). Rewriting division in terms of multiplication yields $LSB_r = LSB_a * LSB_{b^{-1}}$. Unfortunately, knowing the LSB of b does not help us find the LSB of its inverse. On the other hand, if we knew the largest value b can occupy, then we know the smallest value of b^{-1}. Tracking the maximum magnitude of a value is soundly approximated using interval arithmetic, a well establish technique for range analysis [23]. Knowing the interval that b^{-1} occupies, we select the worst case LSB over the entire interval.

Returning to the example `(a*b)*c` from figure 3.3, let us start by underflowing all inputs below `1e-6`. The *LSB* for each input is directly computed as -36. Following the rules for multiplication, the *LSB* of `a*b` is at worst -72. The final multiplication gives an *LSB* of $-72-36 = -108$. This *LSB* is slightly above the subnormal range (values below -126). Thus, the final operation is statically shown to be safe and the underflow of the intermediate product `a*b` may be safely omitted.

3.4.3 Full Analysis of LLVM Bitcode

`FAST` uses an analysis pass on the LLVM bitcode to determine which underflows are not required. The analysis processes every instruction once, in the order they are found in the bitcode. This means that the analysis does not handle loops or any backward control flow (e.g.

goto a previous label) – such cases are assumed to take on any value.

Each value is represented by a *fact* consisting of a list of augmented intervals and a flag that indicates the value may be NaN (see figure 3.4 for the type of fact). Support for multiple intervals is necessary because the inverse of an interval crossing zero is disjoint (e.g. the inverse of $[-1,1]$ is $[-\infty,-1]\cup[1,\infty]$. The values covered by a fact is the union of all values contains by the augmented intervals plus the single value of NaN if the `nan` flag is set. The fact can cover all special values: both zeros and Inf may be found in the augmented interval.

The following rules dictate how to compute the fact for a given operation. Each rule also contains a sketch of correctness.

Unknown Values. Any unknown can take on any value given by $([(-\infty,\infty,b)],true)$ where b is the lowest bit possible given the format. The fact covers every floating-point number using the argumented interval, and NaN is covered by the flag. Thus every possible floating-point value is represented in the fact.

Constants. Any non-NaN constant is $([(c,c,\text{lsb}(c))],false)$ representing an interval covering the single value c. The LSB is directly found by computing the right-most non-zero digit of c. A NaN constant is $([],true)$, only setting the NaN flag and provding no augmented intervals.

Square Root. Square root, being a unary operation, simply requires the computing the square root of the end points in each interval, assuming they are strictly non-negative. Because the the square root of negative values produces NaN, any strictly negative intervals are discarded. For intervals that contain both negative and positive values, the output only uses the positive portion of the interval. Lastly, the NaN flag of the output is set if the input contains NaN *or* any input interval contains a negative value. These rules correspond to the square root definition by IEEE-754.

Binary Operations. All binary operations require operating on the cross-product of the input intervals. In this manner, each interval from the left-hand side is operated on by each interval from the right-hand side, thereby ensuring that every value is accounted for in the output

other than NaN. Binary operations follow the the standard rules of interval arithmetic to soundly approximate the output. Combined with the LSB computation from section 3.4.2, the output set of augmented intervals account for all non-NaN values. The NaN flag is set based on the rules established by the IEEE-754 standard (e.g. $\infty - \infty = NaN$, $0/0 = NaN$, etc).

Interval Union. The number of augmented intervals grows very quickly due to cross-product in binary operations. It is therefore useful to compute the union of intervals to make the analysis tractible. The union of two intervals simply extends the lower and upper bounds to be their minimum and maximum and the LSB is set to the minimum of the two input LSBs. The resulting augmented interval covers every value in the input interval by both completely overlapping (in terms of endpoints) and reducing the spacing the smaller of the two. For the NaN flag, it is set if either input fact has the NaN flag set.

3.4.4 `FAST` Algorithm

The `FAST` implementation walks through LLVM bitcode to determine which floating-point operations should be prefixed with an underflow (or overflow for division). `FAST` iterates first over each basic block in the order given by LLVM, and then it iterates over each instruction to apply protection as needed. All unknown inputs are assumed to take on any possible value. Once a fact is computed for a given instruction, underflow protection is applied only if any augmented interval overlaps the subnormal range *and* the LSB is small enough to generate subnormal numbers. If the instruction requires protection, all values below a threshold are rounded to zero (and negative zero). For division operations, the numerator overflows if falls above a threshold.

There is an inherent trade-off determined by the underflow (and overflow) threshold. Smaller underflow thresholds (and larger overflow thresholds) increase the usable range of values but reduce the number of opportunities to bypass explicit protection. Likewise, restricting oneself to a small range of floating-point numbers improves performance by more effectively

pruning underflows and overflows. By setting the thresholds to the largest possible range, `FAST` would exactly mimic the `RESTRICT` scheme from previous research with one exception: the static analyis may find code sequences that propagate safe values without over-restricting. In this way, the augmented intervals can serve as an optimization pass while still preserving the constant-time safety of the code.

$$\exists z, a \leq z2^{LSB} \leq b \tag{3.1}$$

3.5 Flags

The basic principle behind `FLAGS` is to forcefully set the FTZ+DAZ flags in order to prevent slowdowns due to subnormal numbers. The FTZ+DAZ flags are set using `ldmxcsr` instruction, and the flags state is preserved until changed by another call to `ldmxcsr` or by the kernel when context switching. Because the FTZ+DAZ flags are stateful, it may be difficult to guarantee that the flags are never modified. The developer must reason about not just their code but also every called library and all third-party code executed in the process's context. For multi-process programs, this adds the requirement that every process must ensure that the FTZ+DAZ flags are set. To aid the developers, `FLAGS` transforms the source code to ensure that the FTZ+DAZ flags are always enabled on the target code, even in the presence of third-party code.

At its most basic, `FLAGS` enables FTZ+DAZ with `ldmxcsr`[1] at every untrusted entry point to the target code. To cover these cases, FTZ+DAZ are set at the beginning of every function and after every function call. We assume that the target code does not intentionally unset the flags. The code in figure 3.5 demonstrates how `FLAGS` protects a function containing floating-point operations. Despite only containing a pair of floating-point operations, the call to

[1] a call to `rdmxcsr` is required to prevent modifying other CPU flags.

```
foo(a, b):
        setFlags()     # added code
        r := a+b*b
        s := bar(r)
        setFlags()     # added code
        return s
```

Figure 3.5: Pseudocode for a function `foo` that was protected using `FLAGS`. The two calls to `setFlags` are used to enable FTZ+DAZ. Because `bar` may be an untrusted function, the flags must be set after returning from it.

an external function requires setting FTZ+DAZ a second time. The code for changing the CPU flags is a slow operation, taking approximately 26 clock cycles. Because function calls occur so frequently in a program, it is necessary to try and avoid setting the flags whenever possible.

3.5.1 Optimization using Static Analysis

Due to performance issues, the primary concern for `FLAGS` focuses on recovering performance by omitting flags when unneeded. `FAST` uses three techniques for eliminating calls to set FTZ+DAZ.

Functions without floating-point math. Any function that does not perform any floating-point operations does not require protection and is therefore omitted. No protection is used when either entering the function or returning from a function call.

Calls to local functions (that are protected). Calls to *protected* local functions do not require asserting the FTZ+DAZ flags after returning. This is safe due to the fact that any protected function is guaranteed to keep the flags set before returning to the caller. Unprotected functions are not safe to be called because they themselves may contain further unsafe calls (e.g. calls to third-party code) without resetting the flags.

Obviously unsafe floating-point values. `FLAGS` does not protect operations where either input or the output is used in an unsafe manner. For example, consider the code **`if`** `(a+b) > 0: while(true)`. Even though the operation `a+b` would normally be protected, its very next

use is in a conditional that may loop forever. The code is clearly unsafe because information about `a` and `b` are leaked by simply observing if the program hangs. We can therefore conclude that `a+b` does not need protection, and thus omit it. Operations are found to be obviously unsafe by walking the use-def chain on the LLVM intermediate representation. Chains that lead to conditionals are assumed to be unsafe, allowing the source operation to be left unprotected.

3.6 Evaluation

We evaluate `BASIC`, `FAST`, and `FLAGS` to answer three questions: does the resulting code execute in constant-time? do our restrictions break existing programs? and how much overhead is added?

3.6.1 Constant-Time

We verify that our implementations are in fact constant-time by precisely measuring the execution time given different inputs. This technique mirrors previous floating-point timing-channel defenses [3]. The timing difference between each operation on normals and special values, and subnormals vary by less than a single instruction. This indicates that `BASIC`, `FAST`, and `FLAGS` successfully remove the timing variations present in floating-point operations.[2]

3.6.2 Preserve Correctness

We evaluate `BASIC`, `FAST`, and `FLAGS` on benchmarks and test suites to determine if they break existing programs. Because `FAST` changes the semantics of floating-point programs by underflowing small values (and overflowing large values), we determine whether or not these changes are significant enough to cause tests to fail. `FLAGS`, on the otherhand, should only eliminate subnormals, a technique that is already applied by compilers when given the

[2] `BASIC` does not fix timing variation in subnormal as intended. See section 3.3 for details.

`-ffast-math` option. Last, `BASIC` should not change the semantics of floating-point operations at all since it exactly preserves the behavior of special values and leaves other values types unmodified.

SPEC Benchmarks. SPEC CPU2006 provides a set of benchmarks to testing the performance and correctness of compilers. Each test executes a floating-point program and verifies that the result is accurate, taking into account small deviations inherent in floating-point computation (e.g. rounding error). All three protection schemes successfully pass each benchmark.

Skia Tests. Skia is a graphics library used by the major browser vendors. The project supplies 913 unit tests and 654 rendering tests to verify the correctness of the library. The unit tests are particularly tempermental due to the fact that they verify very precise behavior in uncommon scenarios (e.g. operations on subnormals). The rendering tests mimic real-world situations by performing specific rendering operations and verifying that the resulting image is pixel perfect. `BASIC` failed 2 unit tests, `FLAGS` failed 3 unit tests, and `FAST` failed 5 unit tests. In all cases, the failures came from testing edge cases using very small floating-point numbers. More importantly, all protection schemes pass the real-world rendering tests.

3.6.3 Performance

We measure performance overhead using the benchmarks in SPEC CPU2006 and Skia.

SPEC Benchmarks. Figure 3.6 plots the overhead of all our three new protection schemes (`BASIC`, `FAST`, and `FLAGS`) compared to previous methods (`RESTRICT` and `FULL`). All three new variants are faster than `RESTRICT` and `FULL` on every benchmark. `BASIC` is always the fastest due to the fact that there is no protection against subnormals. `FAST` improves on `RESTRICT` by omitting unneeded underflows and is therefore strictly no worse performance-wise. `FLAGS` is highly variable in its overhead compared to other protection methods. For most of the benchmarks, there is no discernable difference between `FLAGS` and `BASIC` due to the fact that nearly the

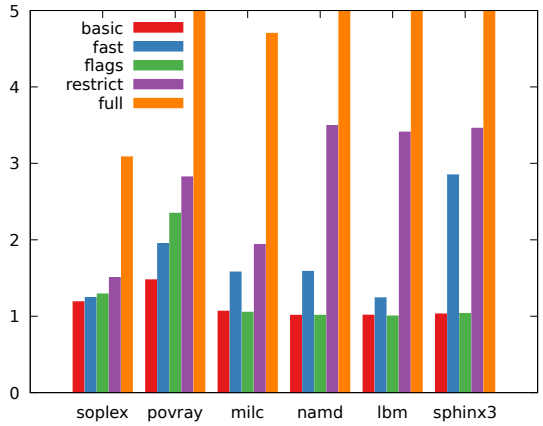

Figure 3.6: The performance overhead when applied constant-time techniques to the floating-point benchmarks in Spec CPU2006. The `BASIC`, `FAST`, and `FLAGS` show substantial speedup over the previous methods `RESTRICT` and `FULL`. The bars for `FULL` are not entirely shown due to its high overhead. For the abbreviated benchmarks, `FULL` has a slowdown between 7x and 12x.

entire benchmark is spend inside a single function with no external calls. Thus, the cost of setting the flags is only incurred a handful of times over the course of the many minutes the benchmark takes to complete. On the other hand, `FLAGS` is slower in `450.soplex` and `453.povray` due to the fact that these benchmarks make heavy use of function calls in the critical path. This leads to significant overhead when forcefully asserting the FTZ+DAZ flags for every entry and return from a function.

Skia Benchmarks. The Skia graphics library contains a tool called `nanobench` for performance testing. To ensure we only measure the slowdown on Skia, the constant-time protections are only applied to the library and not to the benchmark code itself. Figure 3.7 provides the performance overhead of `BASIC`, `FAST`, and `FLAGS` using a CDF as a percentage of tests that fall below a given threshold. `BASIC` outperforms all other constant-time techniques with a median overhead of 1.4%, and `FAST` improves on `RESTRICT` with a median overhead of 5.8% versus 10.8%. Unfortunately, `FLAGS` performs very poorly with a median overhead of almost 4%, reaching a slowdown of at least 71x for 5% of tests. The extreme overhead is inherent in Skia due to the large number of function calls that require protection.

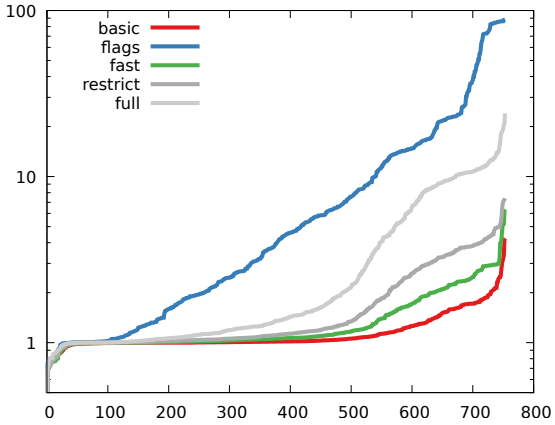

Figure 3.7: The overhead of protecting Skia's performance test as CDF of slowdown. The y-axis is the amount of slowdown relative to unprotected version of Skia, and the x-axis is the number of tests that complete with that much overhead.

3.7 Limitations and Future Work

The constant-time protections have a number of limitations, some of which may be overcome in future work.

3.7.1 Improving the Static Analysis

Both `FAST` and `FLAGS` require static analysis to reduce the overhead of each approach. The existing analysis does not take into account complex control flow, such as loops or gotos. The analysis can be expanded using a more sophisticated algorithm (e.g. work list) to improve precision. An enhanced analysis could discover more opportunities for either `FAST` or `FLAGS` to safely skip protections, ultimately leading to improved performance.

3.7.2 Faster Alternative to `BASIC`

`BASIC` emulates all special values in division and square root to account for their small timing variation. An alternative version may relax the requirement that all special values are correctly emulated. For example, if the user does not need to distinguish between Inf and NaN, the bitmasks for both division and square root can combine these cases and output a NaN. In

a different case specific to `FAST`, infinite values can be rounded down to a finite values to avoid having to handle special values at all. Such simplification would lead to more performant code.

3.7.3 Full IEEE-754 Support with Subnormals

All constant-time implementations do not support subnormals. Instead, they replace both subnormal inputs and subnormal outputs with zero. A complete reimplementation of operations using either purely integer arithmetic or carefully crafted floating-point could support subnormal values. Supporting subnormals would likely come at a significant performance cost that may be acceptable for users that require the full semantics of IEEE-754.

3.8 Acknowledgements

Chapter 3, in part is currently being prepared for submission for publication of the material. Andrysco, Marc; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this material.

Bibliography

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

[2] Cve-2017-5407. National Vulnerability Database, 2017. [online] https://nvd.nist.gov/vuln/detail/CVE-2017-5407.

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

[4] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1369–1382, New York, NY, USA, 2018. ACM.

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

[6] L. D. Baron. Preventing attacks on a user's history through css :visited selectors.

[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 2179003003. https://codereview.chromium.org/2179003003, 2016.

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

[12] I. Committee et al. 754–2008 ieee standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008, 2008.

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

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

[15] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *USENIX Security Symposium*, 2011.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

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

[18] A. Janc and L. Olejnik. Feasibility and real-world implications of web browser history detection. *Proceedings of W2SP*, 2010.

[19] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[20] D. Kohlbrenner and H. Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 69–81, Vancouver, BC, 2017. USENIX Association.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[22] 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.

[23] R. E. Moore. *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, NJ, 1966.

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

[25] A. Rane, C. Lin, and M. Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 71–86, Austin, TX, 2016. USENIX Association.

[26] 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.

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