On Subnormal Floating Point and Abnormal Timing

Marc Andrysco,[1] David Kohlbrenner,[1] Keaton Mowery,[1] Ranjit Jhala, Sorin Lerner, and Hovav Shacham
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

***Abstract*—We identify a timing channel in the floating point instructions of modern x86 processors: the running time of floating point addition and multiplication instructions can vary by two orders of magnitude depending on their operands. We develop a benchmark measuring the timing variability of floating point operations and report on its results. We use floating point data timing variability to demonstrate practical attacks on the security of the Firefox browser (versions 23 through 27) and the Fuzz differentially private database. Finally, we initiate the study of mitigations to floating point data timing channels with `libfixedtimefixedpoint`, a new fixed-point, constant-time math library.**

Modern floating point standards and implementations are sophisticated, complex, and subtle, a fact that has not been sufficiently recognized by the security community. More work is needed to assess the implications of the use of floating point instructions in security-relevant software.

I. Introduction

The running time of floating point addition and multiplication instructions can vary by two orders of magnitude depending on their operands. This fact, known for decades by numerical analysts, has not been sufficiently recognized by the security community.

Floating point operations, if performed on secret data, expose software to *data timing channels*: timing side channels that arise not because the trace of instructions executed or the trace of memory locations accessed vary according to secret inputs, but because the same instructions, acting on the same memory locations, vary in their running time.

Data timing channels were hypothesized by Kocher in his 1996 paper introducing timing side-channel analysis to cryptography [33], but the intervening years have yielded only one exploitable example: integer multiplication on some small-die embedded processors [24].

In this paper, we show that data timing channels are not a hypothetical threat but a real and pervasive danger to software security. We use the timing variability of floating point operations, specifically surrounding special-case "subnormal" numbers very close to zero, to break the security of two real-world systems.

First, we demonstrate that subnormal floating point data timing channels can be used to break the isolation guarantees of Web browsers. From release 23 (when the `requestAnimationFrame` API was added) and until release 28 (when SVG filters were moved to the GPU), the Firefox browser allowed JavaScript to measure the running time of SVG filters applied to Web content through CSS. Paul Stone showed that timing variations arising from a data-dependent branch in one filter, `feMorphology`, could be exploited to perform history sniffing or reveal the content of cross-origin iframes [49]. We show that floating point data timing channels in the computation of filters (without any data-dependent branches) enable similar attacks. Our attack also applies to the "Extended Support Release" of Firefox 24, which formed the basis of the Tor Browser in the 1.0 and 1.1 releases of the Tails operating system.

Second, perhaps more startlingly, we show how subnormals can be used to break the differential privacy guarantees of an extremely carefully engineered data analytics system that was specifically crafted to prevent such leaks. Haeberlen et al. [26] identified a timing covert channel by which malicious queries could break the differential privacy guarantees of the PINQ and Airavat databases. They designed and implemented Fuzz, a differentially private database that "effectively closes all known remotely exploitable channels," including timing channels. We show that carefully chosen values returned by Fuzz microqueries can affect the running time of floating point computation performed by the Fuzz kernel, introducing an exploitable timing side channel. Fuzz has had trouble with floating point before: As Mironov showed [38], Fuzz and several other differentially private databases sample from the Laplacian distribution using an algorithm that interacts badly with fixed-precision floating point arithmetic, allowing sensitive information to leak in the least significant bits of computed results.

A key technical challenge our attacks overcome is how to amplify a timing signal of just a few processor cycles. Ours are the first attacks to exploit data timing channels through timing alone; Großschädl et al.'s attack on integer multipliers with early termination [24] relied on SPA power traces to amplify the timing signal, hence requiring invasive access to the system.

Having established floating point data timing channels as a real and pervasive danger to software security, we turn to defenses. We design and evaluate a new library, `libfixedtimefixedpoint`, for non-integer math for which all operations run in constant time. We have manually

[1]The three first authors contributed equally to the paper.

verified that an AMD64 binary of our library uses only integer instructions that we believe are constant-time. Emulating non-integer operations in constant time imposes overheads, but the overheads may be acceptable for security-critical applications: addition and multiplication in our library take just 15 and 43 cycles, respectively, on a Core i7 2635QM. Our library is available under an open source license.

To sum up, in this paper we demonstrate that data timing channels are a real danger to software security and identify potential mitigation strategies by making the following contributions:

- We show that operations over potentially subnormal values are a data timing channel on modern x86 processors, by measuring the timing variability of floating point operations (Section II),
- We demonstrate how floating point timing variability can be used to mount practical attacks on the security of the Firefox browser (versions 23 through 27) (Section III) and the Fuzz differentially private database (Section IV).
- We initiate the study of mitigations to floating point data timing channels by developing a new fixed-point, constant-time math library (Section V).

II. IEEE-754 Floating Point, As Implemented

Floating point computation is found throughout modern software development, enabling applications to represent a much larger range of values than integers alone. Although floating point formats have been in use for many decades, they have recently gained particular prominence as the exclusive numerical format in JavaScript. There has historically been a variety of competing floating point formats, each defining unique, incompatible encodings with differing properties [30]. In 1985, the Institute of Electrical and Electronics Engineers published a technical standard for floating point formats: IEEE-754 [14]. This specification has seen wide adoption and is implemented by nearly all computers in use today.

Although successful, the IEEE-754 standard poses a difficult challenge for hardware implementors and software developers alike. The complexity of the implementation has led to real-world bugs, such as the Intel Pentium FDIV bug [29], and led to efforts to verify hardware implementations [3, 40, 44, 45]. Software has equally struggled to handle floating point numbers correctly; for example, PHP has had an infinite loop bug when attempting to interpret a specific number [42].

In this section, we will cover the intricacies of IEEE-754 floating point numbers, looking in particular at corner cases defined by the standard, how they are handled by a processor, and how timing information can be extracted.

Format Name	Size Bits	Subnormal Min	Normal Min	Normal Max
Half	16	$6.0e{-}8$	$6.10e{-}5$	$6.55e4$
Single	32	$1.4e{-}45$	$1.18e{-}38$	$3.40e38$
Double	64	$4.9e{-}324$	$2.23e{-}308$	$1.79e308$
Quad	128	$6.5e{-}4996$	$3.36e{-}4932$	$1.19e4932$

Figure 1: IEEE-754 Formats

Value	Exponent	Significand
Zero	All Zeros	Zero
Infinity	All Ones	Zero
Not-a-Number	All Ones	Non-zero
Subnormal	All Zeros	Non-zero

Figure 2: IEEE-754 Special Value Encoding

A. IEEE-754 Floating Point Format

In contrast to the relatively simple two's complement format used for signed integers, IEEE-754 floating point numbers have a more complicated, multi-part format with numerous special cases. Each number is composed of a sign bit, an exponent, and a significand, together representing the real number $(-1)^{sign} \times significand \times 2^{exponent}$. The raw exponent is stored as an unsigned integer, but its effective value is calculated by adding a negative bias value, allowing representation of negative exponents. In normal operation, the significand is stored with an implicit "leading 1": the bits making up the significand actually represent the binary number $1.b_0b_1 \ldots b_N$. To support different precision requirements, the standard defines formats varying from 16 bits to 64 bits. Figure 1 summarizes the formats defined by the IEEE-754 standard.

To accommodate values that cannot be represented in the above format, the standard reserves special encodings for zero, infinity, and not-a-number. Additionally, the standard specifies an encoding for an alternate class of numbers, referred to as *subnormal* (also called denormal). Unlike normal numbers, subnormals are restricted to using the smallest possible exponent, and their significand uses a fixed leading 0 bit, with the form $0.b_0b_1 \ldots b_N$. By removing the leading 1 bit, subnormals allow the representation of values very close to zero. Figure 2 summarizes the special values and their encoding.

B. Processor Implementations

PC processors have supported IEEE-754 floating point values since the introduction of the Intel 8087 floating point coprocessor in 1980. The x87 instruction set was created to communicate with this coprocessor and was later integrated directly into 80486 and later processors. In x87, all computations are internally performed using the 80-

bit "double-extended" format, only converting to the 32-bit or 64-bit formats when performing a load or store. x87 instructions support typical arithmetic (addition, subtraction, multiplication, division) as well as transcendental functions (trigonometry, exponentiation, and logarithms).

Beginning with the Pentium III in 1999, Intel introduced the Streaming SIMD Extensions (SSE) instructions for operating on floating point values, with the ability to perform multiple operations simultaneously. Unlike x87, SSE instructions operate directly on 32-bit and 64-bit operands without using a high-precision internal format. SSE supports simple operations, but does not implement transcendental functions. Although nearly all current Intel-based hardware supports SSE, compilers targeting 32-bit systems do not typically assume SSE support. As result, most 32-bit software uses the x87 instruction set.

IEEE-754 floating point is widely implemented, including in graphics processing units and many mobile processors. Hardware support for subnormal numbers is less common, with some processors rounding subnormals to zero and others falling back on software emulation.

C. Subnormal Performance Variability

Due to the complex nature of the floating point numbers, processors struggle to handle certain inputs efficiently. In particular, it is well understood that operating on subnormal values can cause extreme performance issues, including slowdowns of up to 100× [19]. As an example, on a Core i7 processor using SSE instructions, performing standard multiply between two normal numbers takes 4 clock cycles, whereas the same multiply given a subnormal input takes over 200 clock cycles. Although the timing signal from a single subnormal computation can be difficult to measure, a timing signal can be amplified when computation occurs in a tight loop — a situation that is common with floating point numbers.

The SSE instruction set includes the processor flags flush-to-zero (FTZ) and denormals-are-zero (DAZ), to prevent subnormal values from occurring as inputs to or outputs from instructions. When flags are set, the performance problems associated with subnormals disappears on all processors we tested, although there are no guarantees that these flags will always solve these performance issues. Unfortunately, the x87 instruction set does not provide any method to disable subnormal values.

Beginning with the Fermi microarchitecture, NVIDIA graphics cards support subnormal floating point values [41]. NVIDIA has stated that, consequently, certain operations can suffer from performance problems when operating on subnormal values [27], generating a measurable effect. As graphics card processors have not historically supported subnormal numbers, this provides evidence that subnormals and timing channels will likely become more prominent on future graphics cards.

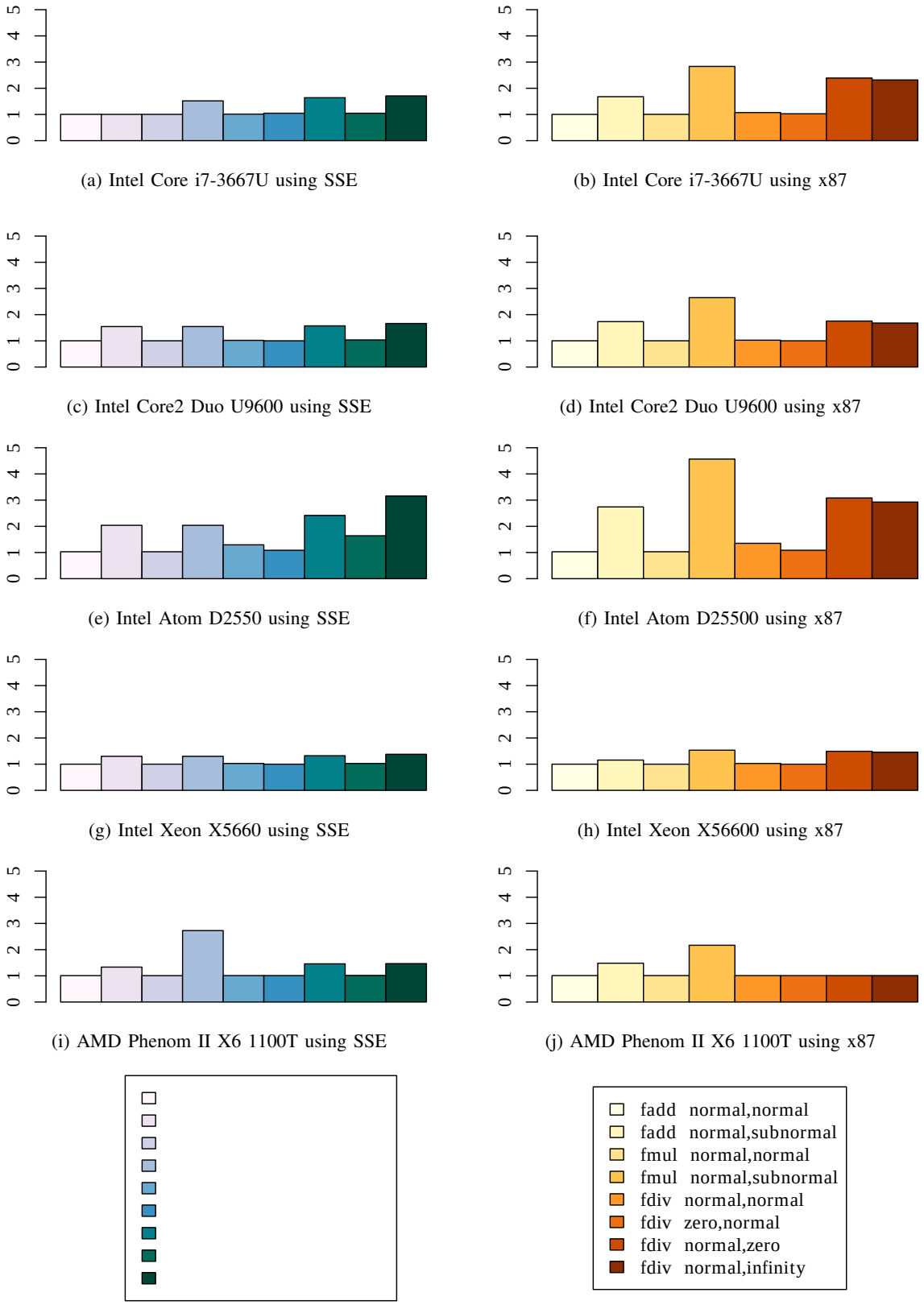
D. Floating Point Benchmarks

To better understand and characterize the slowdowns of floating point instructions, we created a benchmark to measure the execution speed of varying combinations of operations and inputs. We tested x87 and SSE instructions for addition, multiplication, and division, including both scalar and packed SIMD versions. For inputs, we tested every combination of normal values, subnormals, zero, infinity, and not-a-number. For SSE instructions, we performed each test under every combination of the DAZ and FTZ flags. Because x87 instructions slow down when loading and storing into registers, whereas SSE instructions have slowdown when the mathematical operation occurs, we normalize all tests by measuring the number of clock cycles to complete the sequence of loading two values from memory into registers, performing an operation on the two registers, and storing the result back into memory. This load-operation-store cycle corresponds closely with code likely to be found in the wild. We averaged 1000 runs for each combination of instructions and operands, and all results are consistent and reproducible.

Figure 3 summarizes the most interesting results from the benchmark. In particular, multiplying or dividing with a subnormal in either operand or as output produces slowdown on all processors, whether SSE or x87 instructions were used. On all architectures other than the Core i7 using SSE, we found similar slowdowns on add instructions with a subnormal input or output. Using SIMD instructions to operate on multiple subnormals at once amplified the measured performance hit. It is important to note that slowdowns occur when the computation result is a subnormal, even if both inputs were normal values.

The x87 instructions caused highly varying slowdowns that were not limited to subnormal values. Performing a division by zero produces the special value infinity, and dividing by infinity produces the special value zero. In both cases, these operations caused significant slowdown with the x87 `fdiv` instruction and, more surprisingly, the timing of the two operations were measurably different. Additionally, operations involving not-a-number suffered large performance degradation. These slowdowns effected all tested Intel architectures, although the selection of AMD machines we tested showed no performance penalty for operating on special values beside subnormals.

All slowdowns discussed so far have centered around exceptional inputs and outputs: infinity, subnormal, and not-a-number. However, we have measured variable timing with typical values: zero, and normal numbers. For example, the division instructions produce a minor *speedup* on SSE when dividing zero in comparison to dividing a normal number — a case that uses the extremely innocuous values of zero and two. In one very specific instance, we even measured a speedup by a Core i7 when dividing one by one.



(a) Intel Core i7-3667U using SSE

(b) Intel Core i7-3667U using x87

(c) Intel Core2 Duo U9600 using SSE

(d) Intel Core2 Duo U9600 using x87

(e) Intel Atom D2550 using SSE

(f) Intel Atom D25500 using x87

(g) Intel Xeon X5660 using SSE

(h) Intel Xeon X56600 using x87

(i) AMD Phenom II X6 1100T using SSE

(j) AMD Phenom II X6 1100T using x87

Figure 3: Timing variability of instructions based on input operands. Each test measures the time taken to complete a sequence of loading two values from memory into registers, performing the specified operation using the registers as input, and storing the result back in memory. The y-axis gives the ratio of time taken to perform the specified operation versus the time taken to perform an addition between two normal numbers.

These results show that the timings of floating point operations vary wildly based on data input. The amount of slowdown and on which values is highly dependent on the processor, varying significantly between different architectures by the same manufacturer. As a takeaway, developers have absolutely no guarantees about the timing of floating point operations unless they are able to know *exactly* which processor is used, what instructions are executed, and what inputs are fed into those instruction. Even accounting for all these factors, we cannot say with confidence whether or not these timing differences will persist in future processors, or whether new data-dependent timing channels will be discovered later.

E. Subnormal Rationale

Subnormal support incurs a significant overhead, so why should processors support subnormals? And if they are supported, why should they be enabled by default? The most compelling reason for subnormal support involves reasoning about code like this [23, Section 2.2.4]:

```
if(a != b)
  y = 1 / (a - b);
```

Checking that the variables `a` and `b` are not equal would appear to guarantee that the result $a-b$ could never be zero and the division would be safe. The result $a-b$ could be a subnormal value, causing a division by zero if subnormals are rounded to zero. Subnormals make possible "gradual underflow," preserving the property that two unequal values can be subtracted yielding a non-zero result.

III. Firefox Pixel Stealing

In this section, we demonstrate the use of subnormal floating point numbers to subvert Firefox's single-origin policy, and show how a malicious website can use modern browser features to extract page content from unaffiliated victim sites in an `iframe`, or to sniff user browsing history.

A. A History of Stolen Pixels

In 2013, Paul Stone [49] (and, independently, Kotcher et al. [35]) demonstrated a new technique for cross-origin pixel stealing in the browser: a timing side-channel present in CSS Scalable Vector Graphics (SVG) transforms. These transforms can be applied (via CSS) to any element of a webpage, including `iframe`s. Notably, when cross-origin content is contained in an `iframe`, the containing page can apply SVG transformation filters at will to that `iframe` (whose content the page does not control). By choosing specific SVG filters and measuring page render times, Stone was able to repeatably extract any pixel value from a website he did not control.

The SVG filters available in browsers include blurs, clipping, color transforms, and generalized convolutions. When applied to a DOM element via CSS, the SVG filter must be computed over the rendered pixels of the filtered element every time the content of that element changes. Stone discovered that the `feMorphology` (erosion and dilation) SVG filter was written with a particular optimization, allowing for a fast path on nearly homogeneous input. For each output pixel, this filter considers a sliding window of input pixels, taking the darkest individual pixel in the window as the output. As long as the previous darkest pixel remains in the window, the filter is designed to consider only new pixels in the window, rather than all pixels in the window. Obviously, this minor optimization will trigger much more often on an single-color image rather than a highly noisy one. This presents a *timing side-channel*, where the amount of time rendering the transformed image takes leaks information about the content. By layering `iframe`s, Stone's attack is able to isolate individual pixels of interest, multiply them against a noisy image, and repeatedly time the rendering of the `feMorphology` filter on the result to extract pixel values. The exact methods used to isolate and extract the value are very similar to the methods we used, as described in Section III-B.

B. Pixel Extraction via SVG Filters & Floating Point

We have implemented a new SVG filter timing attack, using floating point instruction timing rather than the source code fast path described above. Our attack takes advantage of longer wall-clock execution times of floating point instructions with subnormal arguments versus normal arguments, as described in Section II-D. This attack can read arbitrary pixels from any victim webpage, as long as the victim page can be rendered in an `iframe`. A full description of our attack follows, and is illustrated in Figure 4.

1) Pixel Isolation and Expansion: To amplify the timing side channel enough to be measurable, we first must isolate and expand the targeted pixel. First, the victim `iframe` (1) is set to a very large size (to avoid scrolling) and its source is set to the page of interest. Next, to select the target pixel, we place the `iframe` in a 1×1 pixel `div` (2). We scroll this `iframe` relative to the `div` via JavaScript such that the 1×1 pixel `div` displays only the currently selected target pixel. We additionally apply a thresholding `feColorMatrix` and `feComponentTransfer` to the 1×1 pixel `div`, to binarize the color to black or white. The targeted pixel is now ready to be attacked. Next, we introduce a second `div` with the `background:-moz-element` attribute set to the isolating 1×1 `div`. With this, we generate an arbitrarily sized *pixel-inspection* `div` (3) whose fill color matches the thresholded target pixel.

2) SVG Filter and Timing: To read the pixel value, we need to time a computation on the targeted pixel. We attach a `feConvolveMatrix` SVG filter (4) to the pixel-inspection `div`, which introduces the timing side channel. `feConvolveMatrix` is a generalized filter that allows for the definition of an arbitrary kernel matrix that is then run

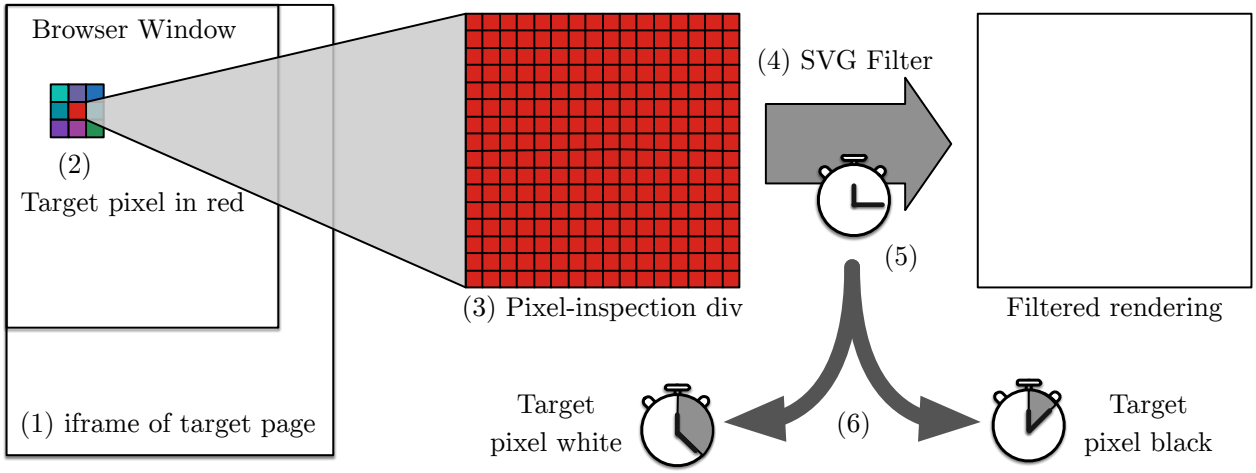


Figure 4: Cross-Origin SVG Filter Pixel Stealing Attack in Firefox

over the input pixels. In our case, we use a 2×2 matrix, all of whose entries are set to the subnormal value $1e-42$. When this filter computes an output pixel, if the source pixels are non-zero (white), the floating point operation performed is $norm \times subnormal = subnormal$. When the source pixels are zero (black), the operation is $zero \times subnormal = zero$. These multiplications are then summed, non-black images result in several summations of $subnormal + subnormal = subnormal$ while a black image results in several $zero + zero = zero$ floating point operations. Depending on the processor, this will result in some amount of computation time difference (see Section II-D) based on the source image's color. Our test page timed the following SVG filter to extract pixels.

```
<feConvolveMatrix in="SourceGraphic"
order="2 2" edgeMode="duplicate"
kernelMatrix=
  "1e-42 1e-42 1e-42 1e-42"
preserveAlpha="false" />
```

We time the rendering of the filtered `div` (5) using `requestAnimationFrame`, which allows registration of a function to be called on completion of the next frame. We time the render by adding the `feConvolveMatrix` filter to the pixel-inspection `div`, taking a high resolution time reading, and registering a function that will take another time stamp after the frame is completed. We use `performance.now()` as our high-resolution timer. For each pixel, we repeat this process once, and make a guess (6) as to its original color using the calibrated threshold described below. Note that timing the filter over only the original 1×1 `div` would not have worked, since the render timings must be greater than the minimum frame render time for there to be a difference between black and white pixels. By applying this filter to the pixel-inspection `div` we obtain a timing for an individual pixel that is perceptible by the timer.

3) Calibration: Since every machine, browser install, and even page render can be slightly different, we run a calibration phase before attempting to steal pixels. The goal of the calibration phase is to obtain average render times for black and white pixels, and then calculate a threshold for classifying target pixels. The calibration phase sets the color of the isolating `div` to black and white alternating, while timing the rendering of the filtered output each time using the above timing scheme. By averaging several white render times and black render times, and taking the midpoint between the averages, we calculate a threshold T. During the pixel steal attack, we time the filtered rendering of each pixel, and compare to T. We categorize the pixel as black or white based on if the time is above or below T.

We found proper calibration to be one of the trickiest parts of making the attack reliable. Render times are generally relatively stable, but will unexpectedly be very slow or fast. We found that different systems needed a different sized pixel-inspection `div` before render times showed a difference between black and white. If the `div` is too small, the rendering time always lies within a single frame (16ms) and we can see no difference from JavaScript between black and white. If the `div` is too large, Firefox will often give obviously incorrect times for the render, far smaller than is possible. This occurs, for example when the `div` is larger than the browser window, and our registered function is mistakenly called when the non-displayed portions of the page finish rendering (that is, instantly). One version of the attack attempted to automatically find an optimal size for each target machine, but consistently ran into problems with undependable render times, causing this calibration to choose much larger pixel-inspection `div` sizes than needed. We settled on expanding the target pixel to a 200×200 region by default, as this was reliable on all tested vulnerable configurations.

Firefox	Duration (min)	B&W delta (ms)	Black errors	White errors
23	7.24	39.68	41.7	3.9
24	5.50	40.04	146.5	1.0
25	5.54	47.08	103.3	1.3
26	6.27	43.17	0.0	1.2
27	6.41	42.88	0.2	2.4

Figure 5: Firefox Checkerboard Recovery 32-bit

Firefox	Duration (min)	B&W delta (ms)	Black errors	White errors
23	2.33	27.76	0.0	4.4
24	2.19	26.06	0.0	3.7
25	2.24	26.06	0.2	10.0
26	2.15	24.66	0.1	3.0
27	2.21	25.86	0.0	2.0

Figure 6: Firefox Checkerboard Recovery 64-bit

C. Building an Attack

The loss of a single pixel value may not seem important; however, by reading multiple arbitrary pixel values, an attacker can perform several attacks. These are the same attacks proposed by Stone [49], since under our attack model, an attacker has similar capabilities.

First, the attacker can sniff browser history by applying a custom style to links on the sniffing page — black background for visited and white for unvisited, for example — and reading a single pixel of the background of the link. Web pages normally cannot determine what color the browser has applied to links they include, precisely because this would allow an attacker to learn what URLs a user has visited [7]. For robustness in the face of noisy rendering times, the attack would likely need to read several background pixels. Given 3 pixel reads per link, an attacker can check 10 or more links per second on a machine similar to our test setup.

The attacker can also read cross-origin pixels for pages that allow themselves to be `iframe`d. This would allow an attacker to read any sensitive data on the target site, such as usernames, account information, or login status. Many sites disallow embedding in `iframe`s for sensitive pages, and these pages would be protected from this attack [46].

Firefox 30 and onwards[2] disallowed the `view-source:` scheme in `iframe`s, but prior to that change the attacker could steal CSRF tokens from even protected pages. Since a victim page's frame-busting JavaScript did not run under the `view-source:` scheme, and CSRF tokens are exposed in the source, the attacker could simply read these using a primitive OCR as suggested by Stone [49]. Once in possession of CSRF tokens, the attacking page can mount standard CSRF attacks [8].

D. Attack Implementation and Measurement

We developed a test page version of the attack described in Section III-B, that attempted to steal a 48×48 region of pixels containing a black and white checkerboard pattern. As the pattern was static, the page was able to calculate the number of errors. We ran this page in official Firefox major releases on a Debian Linux machine with an Intel Core i7-2600 CPU. The machine was under a normal desktop load, with another browser running an email client. We tested each affected major version of Firefox. We ran the experiment ten times, with a forced page reload between runs; only the attack page was open. Figures 5 and 6 show the averaged results for each vulnerable major Firefox release. Duration measures the total time to steal the 48×48 region took in minutes. B&W delta is the difference found during calibration for black pixel vs white pixel render time with filter in milliseconds. Errors measure the respective number of pixels that were not labeled with the correct color. We included an option on the test page to change how many copies of the target pixel were created, defaulting to a 200×200 region; all data was collected with this default. We found that at larger areas, the filter took predictably longer. Since timing fluctuations were not amplified the same amount, there were fewer timings near the threshold, resulting in fewer pixel errors.

Note that Figure 5 has several entries with very high black errors. These are entirely due to individual runs with poor calibration. It is unclear what caused some renders of the SVG filter to take two orders of magnitude longer than average, but it occurred much more frequently on the 32-bit version of Firefox than the 64-bit.

When we went to investigate the high rate of black errors in Figure 5, we discovered that the test machine had undergone an OS package update. This has caused the same 32-bit binary versions of Firefox as before to exhibit similar error rates to the 64-bit versions. Average timings and deltas of 32-bit Firefox versions have not been affected, but the occasional large timing differences are no longer present. The likely culprit is some aspect of the GTK and glibc software stack that has changed in such a way that older Firefox 32-bit releases are more stable. We were unable to determine exactly what aspect of the update caused this change.

Figure 7 shows a common run from 64-bit Firefox 27 on a Debian Linux machine. This instance has a single white pixel error, which was present in almost every test run. In our testing, the first recorded animation frame render time is unexpectedly fast, which causes a single error.

Figure 8 shows the stolen pixels from the front page of http://www.bbc.com using different pixel-inspection `div` sizes. These tests were run on Firefox 27 64-bit on the

[2]https://bugzilla.mozilla.org/show_bug.cgi?id=624883

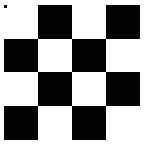


Figure 7: Stealing a 48×48 pixel checkerboard

Figure 8: Stealing a 48×48 pixel region from www.bbc.com, at 100×100, 200×200, and 300×300 pixel-inspection `div` size.

same Debian Linux machine as the other tests. As the size of the filtered region (pixel-inspection `div`) increases, the render time and the delta between black and white pixels increases. Thus, the minor fluctuations in timing have less impact on the total render time, and the output has less errors. This effect is more pronounced on larger websites running JavaScript and loading other resources than on our test checkerboard image.

While stealing a 48×48 checkerboard takes several minutes, an attack does not have to steal all the pixels on a page to be useful. As demonstrated in [49], with intelligent selection of pixels, OCR can be run reading only $log_2(N)$ pixels per character for a target font with N characters. Since our attack reads around 16.4 pixels-per-second in the best case, we can read alphanumeric text at ≈ 3.23 characters per second. Alternatively, history sniffing requires one pixel per URL, so we can scan 16.4 potential URLs per second in the best case.

E. Vulnerable Browsers

While the attack described in Section III-B works on any SVG filter that will accept subnormal floating point values, it relies on the FPU to exhibit timing differences based on arguments. We found that the only major browser (as of mid-2014) that ran SVG filters on the CPU was Firefox. All other major browsers ran filter computations on the GPU, regardless of configuration. While some GPUs [27] exhibit similar timing differences, our test design was unable to detect them.

Firefox was vulnerable to this attack from version 23 (released August 6, 2013) through 27. From Firefox 28 (released March 18, 2014) onward, all SVG filters are run on the GPU. Prior to Firefox 23, the browser did not support `requestAnimationFrame`, and thus timing the rendering of the filtered pixels was impossible. We have demonstrated our test page extracting pixels from Firefox 23–27 686 (32-bit) and AMD64 (64-bit) builds on Debian Linux. We have also demonstrated the attack on Windows 7, Mac OS X, and TAILS prior to 1.2. While there are no substantive differences between versions within an architecture, there were notable performance differences between 32-bit and 64-bit builds.

These differences arose because the 32-bit builds use the x87 FPU, while the 64-bit builds use SSE instructions for floating point computations. As described in Section II-D the timing of various floating point operations differs wildly between x87 and SSE instructions. Interestingly, Windows builds of Firefox were only available in 32-bit during this period, so all floating point math was done on the x87 FPU.

F. Firefox Response

The original Mozilla SVG filter timing attack bug thread [50] included a long discussion of how to avoid exploitable timing side-channel vulnerabilities. Paul Stone suggested (as the working draft of the spec did at the time) that filters not be allowed to run over cross-origin pixels. However, the general sentiment was that moving filters to the GPU would eliminate these channels, and that, until then, constant time implementations of the filters could be written in C++. While it appears that, after significant engineering effort, they were able to close the specific `feMorphology` filter timing side-channel used by Stone, our attack demonstrates that not all timing side-channels were removed. Benoit Jacob expressed concern[3] that there was no particular reason to believe that GPUs would be constant time where CPUs were not. Jacob has noted[4] several likely timing side-channels, arising from different floating point inputs to various browser components. We have disclosed the pixel-stealing attack and our concerns to Mozilla.

G. Recommendations

Engineering truly timing side-channel resistant SVG filters is a complex task with two competing goals. Browsers are evaluated heavily on speed, and their developers often focus on improving performance by fractions of a percent. Thus, SVG filters must be fast, and serious performance degradations as a result of hardening filters is unacceptable. Simultaneously, for a filter to be resistant, it must be constant time. Any predictable variability in render times will result in a side channel. Building a very fast and yet completely constant time SVG filter implementation is not only very difficult, it is platform specific! As our data in Section II-D shows, operations that are safe on one platform are unsafe on another, requiring many more complex filters to have hand-crafted assembly per-CPU model for genuinely constant time operation. This amount of work is likely infeasible for browser developers, and the performance impacts (as seen in [50]) are likely to make such filters unusable even if developed.

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=711043#c52

[4] See https://www.khronos.org/webgl/public-mailing-list/archives/1310/msg00030.html and http://permalink.gmane.org/gmane.comp.mozilla.devel.platform/5293

The current working draft of the CSS filters specification[5] mandates that all filters must be made completely constant time, but notes that there are often hardware or platform specific timing side-channels in various computations. A previous version (2012) of the working draft[6] suggested fetching the cross-origin resource with CORS, and stated, "...a filter effect that is applying to a cross-origin '`iframe`' element would receive a completely blank input image." We believe that due to the challenges in creating fast constant-time SVG filters, the latter approach is advisable. Allowing any attacker-observable and attacker-controlled computation over sensitive cross-origin pixels is dangerous. It is important to note that even if this recommendation is followed, history sniffing will still be possible with non-constant time filters. Since history sniffing does not require any cross-origin pixels to be involved, an attacker can continue to implement our attack using any timing variability found in SVG filters. Current versions of Firefox (33 at the time of writing) will still perform attacker-controlled SVG filter transforms over cross-origin content, albeit using the GPU rather than CPU. As Mark Harris, NVIDIA's Chief Technologist for GPU Computing [27] notes, some GPUs do exhibit measurable performance impact with subnormal values; see Section II-C for more. We believe that as page-visible timing precision improves, even GPU floating point calculations will become vulnerable.

IV. Differentially Private Databases

While "big data" has the potential of offering valuable insights from aggregating information about large populations (for example, genetic markers that are predictive of serious diseases), it carries with it the danger of violating the privacy of individuals in those populations (for example, that a given person is afflicted by a particular condition).

Differential Privacy (DP) is a relatively recent approach [20, 21] which aims to reconcile the ability to make precise statistical estimates about the properties of large data sets *without* violating the privacy of any individual sample in the data set.

At a high level DP works by adding noise — random values from a carefully chosen distribution — to the results, in a way that masks the exact value of the individual samples while approximately preserving the overall aggregate result over all the samples.

A. Mathematics of Differential Privacy

More concretely, imagine a data set D, and a query program Q which the querier would like to run. For example, D could be the admission data for a hospital, and Q might compute the number of heart patients and the average length of their stays. Person A, who visited the hospital after a heart attack, has a single entry in D: a. We can create a new database D' by removing a from D: $D' = D - \{a\}$. Differential privacy means that a querier cannot tell which database Q runs on — $Q(D)$ is indistinguishable from $Q(D')$. In this way, a malicious attacker cannot learn whether A has heart problems, but an honest querier can roughly learn the average duration of the hospital's heart patient visits.

A basic parameter of differential privacy schemes is ε, which scales the privacy of the scheme. Smaller ε gives a more secure scheme, but introduces more uncertainty into the query results.

There are several approaches to achieving differential privacy, but the most common is the addition of noise from a Laplacian distribution. Addition of properly scaled noise (which can be positive or negative), will completely mask the existence of any single entry a. For details on the Laplacian distribution, see Dwork [20, 21].

B. Differential Privacy Databases

Several groups have used the theory of differential privacy to construct *differentially private databases*, like PINQ [37] and Airavat [43], which allow the user to ask queries of datasets, and which transparently add noise to preserve privacy.

At a high-level, these databases work by carefully restricting the queries into a *map-reduce* format. That is, the user supplies a "microquery" that *maps* each row of the database to some numeric result, and a "macroquery" that *reduces* the (mapped) results from each row into the overall aggregate result.

By structuring queries in this manner, the DP database can add noise at the appropriate points *after* the aggregation (reducing), in order to provide rigorous differential privacy guarantees.

C. Timing Channels Break Privacy

Unfortunately, the DP guarantees crucially rely on the fact that the user is privy *only* to the primary *numerical results* of the query, and not other unintended results or attributes, such as query running times.

Indeed, Haeberlen et al. [26] demonstrate that if the user can also determine the running time of queries she posed to the system, then the resulting covert channel can be used to compromise the DP guarantees.

In particular, Haeberlen et al. show how to mount classical timing attacks on PINQ and Airavat by carefully crafting queries that follow the same basic pattern: if a highly sensitive record is seen, the microquery performs an unexpected action (such as spinning in a loop for several seconds, or using extra memory). By then observing the running time (or memory consumption), the querier can infer that the sensitive record is present in the database.

[5] https://dvcs.w3.org/hg/FXTF/raw-file/705f723192d2/filters/Overview.html

[6] https://dvcs.w3.org/hg/FXTF/raw-file/4b53107dd95d/filters/index.html

D. Restoring Privacy by Eliminating Timing Channels

Haeberlen et al. [26] also present a new database called Fuzz, which aims to restore privacy by carefully designing the query language and run-time to ensure that all queries execute in exactly the same amount of time, *independent* of the database contents. This property is achieved by a series of measures. A rough sketch of Fuzz is presented next in this work; for a full treatment, please refer to the original paper [26].

1) Fuzz Queries: In the differential database model, queries are written and supplied by an attacker, while the database is operated by a trusted party. With this in mind, Fuzz's designers spent most of their effort protecting and sanitizing queries. Each query is submitted to Fuzz as source code, written in a subset of Caml, and is heavily restricted in the actions it can take.

Queries are written using the *map-reduce* programming model: a microquery maps over each individual row to produce a result, and the macroquery combines the row results into aggregate statistics. To produce a differentially private result, Fuzz modifies the macroquery's results slightly, by adding a random value drawn from a Laplacian distribution.

The differential privacy guarantee concerns a single row—a malicious attacker should be unable to determine the existence of, or indeed anything about, a single row. Fuzz therefore requires each query program to declare the possible output range of its microqueries, and this parameter is used to generate the distribution of Laplacian noise. Once the noise is added, the contribution of each individual row to the final result is masked.

Further, to achieve a global constant execution time, Fuzz requires each microquery to execute in a constant amount of time. Therefore, query authors must also specify a "timeout" and a "default value" for each microquery. To enforce these limits, Fuzz requires a somewhat involved operating system and hardware configuration, including running on its own dedicated machine. While each microquery is executing, a tight loop, calling `rdtsc` to read the clock cycle counter, waits for the microquery deadline to arrive. When it does, the watcher issues a `longjmp` call, resetting the Caml interpreter to a previously-established `setjmp` location, ready to record the microquery result. If the microquery has finished and produced a value, that value will be used; otherwise, the default value will be substituted for this row.

This interpreter reset also guarantees another essential property of Fuzz: microquery non-communication. If microqueries could communicate, and base their result on the result of a previous microquery, they could, in aggregate, overwhelm the Laplacian noise addition step and break the differential privacy guarantee. The Fuzz query language has no communication primitives, and the interpreter reset eliminates any side-channels.

Once the query is written and ready to run, Fuzz uses a modified version of the Caml Light[7] runtime to compile it into a 32-bit x86 executable, suitable for executing on a database.

2) Query Aggregation and Environment: Macroqueries aggregate the results of microqueries, which are computations performed in isolation on each row of the database. Fuzz-provided library functions bridge the gap between macro- and micro-queries.

Fuzz provides queries with four Caml functions for this purpose: `bagmap`, `bagsplit`, `bagsize`, and `bagsum` (in Fuzz parlance, collections of data are known as "bags"). These correspond roughly to `map` (`bagmap`), `filter` (`bagsplit`), and `reduce` (`bagsize` and `bagsum`) in functional programming, but have been specifically designed and implemented to support constant-time operation.

Internally, these functions are implemented in two parts: a small Caml shim and a backend function written in C. They are written to ensure constant-time execution; for example, `bagsplit` creates a new copy of the database, identical in size to the original, with non-existent rows marked via metadata.

Fortunately, `bagsum` and `bagsize` are fairly simple to write in a constant-time way: they need to perform a very simple operation once for each active row in a bag. Since the database size is considered public information, they simply run a `for` loop over the bag. Fuzz's C implementation of `bagsum` can be seen in Figure 9. Note that, as aggregating functions, they will only run once per macroquery, and are assumed to be constant-time in the size of the database, which is public information. Fuzz, therefore, does not try to restrict them via technical means (like `longjmp`) to run in constant time. Also, Fuzz's strategy for timeout-based limitation will not work on these aggregating functions—there is no default value that will not immediately indicate to the querier that a timeout has occurred, and that fact alone could be enough to break differential privacy.

In contrast, `bagmap` and `bagsplit` allow a query to run *arbitrary* code on each item in a bag. To execute such queries in constant-time, Fuzz makes various modifications to the Caml runtime and operating system configuration, as described in the preceding section and in the original Fuzz paper [26].

E. Subnormal-based Timing Attack on Fuzz

As part of its software distribution, Fuzz includes several sample queries—including several example "evil" queries, which demonstrate the constant-time nature of Fuzz. These queries are modified versions of Haeberlen et al.'s timing attacks against PINQ and Airavat, mentioned earlier. Fuzz's protections close these timing attack vectors, and the malicious queries that ship with Fuzz are unable to expose sensitive records.

[7] http://caml.inria.fr/caml-light/

```
value cbagsum(value dbhandleV) {
  dbHandle db =
    database[Int_val(dbhandleV)];
  double d = 0;
  int i;
  for (i=0; i<__numRows; i++) {
    char *theRow = db +
      (__numBytesPerRow*i);

    assert((theRow[0] == 'N') ||
           (theRow[0] == 'X'));
    /* don't forget the 0x01 */
    if (theRow[0] == 'N')
      d += atof(&theRow[2]);
  }
  return copy_double(d);
}
```

Figure 9: C implementation of `bagsum`, Fuzz's function to aggregate the results of per-row query computation. Attacker-controlled values are highlighted.

When we look closely at the implementation of `cbagsum` (Figure 9), other potential issues reveal themselves. First, untrusted metadata (`theRow[0]`) is used to decide control flow. While the time spent on a single `atof` and an add is quite small, a meticulous attacker could learn details about approximately how many rows were summed.

However, if the attacker is interested in the existence or non-existence of a single row, this is a very weak signal — to reliably extract information, the attacker needs a way to amplify the transmission, letting the result somehow impact the processing of other rows. To do this, we leverage the data type Fuzz uses for the accumulator: `double`.

1) Amplification by Accumulation: Simply, the attacker writes three nearly-identical queries, and submits each for execution. The first query uses `bagmap` to process each row, and produces 0 for each element. The second query is much the same, but produces a subnormal for each row — this represents the worst case scenario, where every row is of interest. The third query almost always produces 0 as well, but includes a probe: if a row of interest is seen, it produces a subnormal floating point number (in our case, 10^{-310}); otherwise, zero.

If the sensitive row is the first row of a 1,000,000 row database, the first query will add 0 to itself 1,000,000 times. The probe query, if it finds an interesting row, will add a subnormal to zero 1,000,000 times. As described in Section II-D, due to timing differences in floating point hardware, the probe query will take very slightly longer than the baseline, and from this, the attacker can deduce the presence of the sensitive row.

Probing?	Mean (s)	Min (s)	Max (s)
No (all zero)	50.300	50.295	50.304
Yes (row not present)	50.309	50.299	50.336
Yes (row present)	50.489	50.488	50.493
No (all subnorm)	51.515	51.493	51.552

Figure 10: Fuzz query wall-clock duration. Each query was run 4 times on a database of 1M rows. The probing query was run twice: on one database which contained the row of interest and one database that did not. The non-probing queries simply produce a constant value for each row.

2) Experimental Setup: Our dedicated Fuzz test machine was an Intel Core 2 Duo E8400 at 3.00 GHz, equipped with 4 GiB of memory. We installed Ubuntu 12.04.4 with a 64-bit 3.11.0 Linux kernel. Following Fuzz's suggestions, we disabled all non-kernel daemons, restricted all processes and threads to run on a single CPU core, disabled CPU frequency scaling, disabled disk flushing, ran Fuzz from a ramdisk, mounted all disk-based filesystems as read-only, and ran Fuzz as root so that it could assign its timing loop exclusively to the free processor core.

We ran our malicious probing query and the non-probing baseline benchmarks on this test machine over a sample census database of 1 million rows. The 31st row indicated a 59-year-old woman of indeterminate race making over $200,000, exactly what our malicious query is trying to find. We also ran the malicious query against a "clean" version of the database, which lacked that particular row.

The running time of these queries is presented in Figure 10. Note the large difference (1.2 s) between the two baseline queries: this is due to both the subnormal addition delay and variable time `atof` execution ("0" is easier to parse than "1e-308").

By running the all-zeroes baseline query along with the all-subnormal baseline query, the attacker generates a range of possible timings, and can then place the probe query somewhere on this range. In our case, we see a clear separation of about 0.18 s between the successful probe query, which finds the row of interest, and the all-zeroes baseline. When the database does not have the row of interest, the probe query fails, and the timings are indistinguishable from the baseline. After all the work Fuzz puts in to achieve constant-time query execution, it achieves a total variance of 0.009 s on the all-zeros baseline query. An increase in running time of even 0.18 s is clearly distinguishable, even over a network connection.

By comparing the total execution times of the three queries, the attacker can deduce the presence or absence of any row she is interested in, breaking the differential privacy guarantee that Fuzz is built to provide.

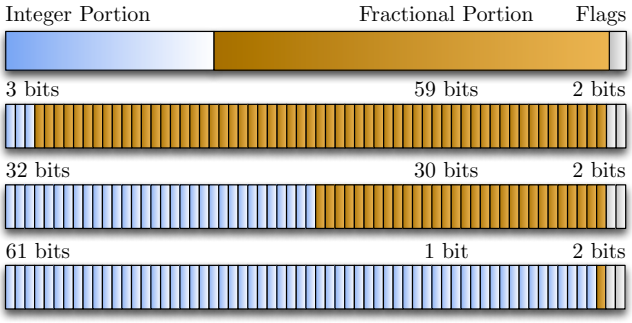


Figure 11: 3 possible internal layouts of a LibFTFP `fixed`. LibFTFP supports anywhere between 1 and 61 fractional bits, chosen at library compilation time.

V. Designing Constant-Time Operations

Floating point numbers have long been a source of frustration for programmers and nondeterminism in programs. Further, their use (even for basic arithmetic) can lead to security and timing issues in the host program, as we have seen in this paper. However, it is entirely infeasible to limit programmers to using only constant-time integer data types — applications involving trigonometry or logarithms require representing numbers between integers.

To bridge the gap between the input-dependent, hardware-contingent, variable-time world of the floating point and the world of constant-time arithmetic operation on pure integer types, we built and are releasing `libfixedtimefixedpoint` (LibFTFP). a C library supplying a fixed-point data type, with all library operations running in constant time. LibFTFP is available online at https://github.com/kmowery/libfixedtimefixedpoint.

LibFTFP provides the `fixed` data type. As with IEEE-754 floating point, a particular `fixed` variable can hold the value of a real number, of positive Infinity, of negative Infinity, or of not-a-number (NaN). These extra numeric states supply a means of signaling and propagating exceptional behavior through LibFTFP computations — for example, dividing 1 by 0 produces NaN, while raising 10 to the 100th power will produce positive Infinity.

A. Representation

As much as programmers would like to use pure, perfectly precise real numbers in our programs, actually representing a number in a binary-based computer involves making choices about compromises. A N-bit data type can only ever represent 2^N different things.

LibFTFP `fixed`s are 64-bit values, the same size as a IEEE-754 `double`. Two of these bits are allocated for the state flags (see Figure 11), which allow us to store the status of the number: normal, +Infinity, -Infinity, or NaN. This leaves 62 bits for the storage of the number. Any particular choice of allocation here will be suboptimal for some application: one programmer might only care about numbers between 0 and 10, but want very good precision, while another is willing to trade precision to handle numbers up to 2^{50}. Therefore, LibFTFP allows the programmer to choose, at library compilation time, the use of the remaining 62 bits: anywhere between 1 integer bit (in practice, a single sign bit) and 61 fractional bits, to 61 integer bits and 1 fractional bit, in single-bit increments. The number ranges representable by LibFTFP, then, are limited by this choice, but all LibFTFP numbers have 62 bits of precision. With I integer bits and F fractional bits ($I+F=62$), the smallest possible positive value is $\varepsilon = 2^{-F}$. The largest possible positive value is $2^{I-1} - \varepsilon$, while the largest-magnitude negative number is -2^{I-1} (the representable difference is due to two's complement sign storage).

B. Operations on Numbers

A single string of bits, by itself, is useless. It only has meaning when associated with a set of operations, transforming it from a binary sequence into a number. Thus, LibFTFP implements nearly every x87 floating point operation, each with its own input-agnostic constant running time, tested on each possible configuration of representable bits:

- Arithmetic: Add, Subtract, Multiply, Divide
- Comparison: Equality, Value Comparison
- Sign adjustment: Absolute Value, Negation
- Rounding: Floor and Ceiling
- Exponentials: e^x, $\log_2(x)$, $\log_e(x)$, $\log_{10}(x)$
- Powers: x^y, Square root
- Trigonometry: Sine, Cosine, Tangent
- Conversion: Printing (Base 10), to/from `double`, to/from `int64_t`

Composing these operations should be sufficient to produce almost any needed mathematical function, in a secure and input-agnostic manner.

Several operations are implemented as approximations, and have associated error; see the LibFTFP documentation for details.

C. Performance in Constant Time

Writing performant constant-time software is a unique challenge: the fastest and slowest paths through the code must take exactly the same amount of time, and that amount should be as small as possible.

LibFTFP uses a few simple strategies to supports its claim of constant-time operation: First, compute all possible needed values. That is, each time through each function, every code path is exercised and results are produced, even if nonsensical. For example, when dividing by zero, instead of failing immediately and returning NaN, a full division is carried out (albeit with made-up numbers). Second, use no data-directed branches. Whenever possible, we use straight-line code, devoid of any flow control, and rely on bit shifting and masking to choose between values (such as the NaN and

nonsense division result mentioned above). The few loops in LibFTFP all have a constant iteration count. Third, use basic integer operations at all times, with the expectation that integer operations will be constant-time independent of input. This is widely regarded as true on modern hardware; however, this assumption does not always hold. Notably, Großschädl et al. [24] showed that, on particular embedded processors, the time to perform integer multiplication varies with the input operands. Note that if the hardware platform cannot guarantee constant-time performance on some subset of integer operations, it is nearly impossible (if not actually impossible) to do constant-time math on that CPU, regardless of programmer effort.

While building LibFTFP, we discovered that the Intel x86 instructions for integer division (`div` and `idiv`) have an input-dependent running time. Both of these instructions divide a 128-bit number by a 64-bit number to produce a 64-bit number. In the case of overflow, a hardware Divide Error exception is raised, which is certainly not constant time, but this can be avoided with careful inspection and modification of division inputs. Unfortunately, even normal, non-overflowing operation is variable time. Notably, on a Core 2 Duo E8400, we have seen `idiv` take anywhere from 31 to 71 cycles, with multiple possible timings along the way, depending on the input. With these characteristics, LibFTFP must avoid `div` or `idiv`, leaving us with no constant-time hardware-accelerated division instructions. LibFTFP contains an alternative software implementation of integer division, using only addition, subtraction, and bit shifts, but taking this path reduces performance considerably, causing a 400% slowdown in our `fixed` division operation as compared to a version using non-constant-time `idiv`.

Writing LibFTFP required the creation of a significant amount of infrastructure to support translating even simple operations into constant time variants. Basic C language control structures like `if`, logical `and` (`&&`), and the ternary operator are unavailable in constant time programming. To emulate common operations, we built a library of C macros that would perform repeated operations. For example, the `MASK_UNLESS` macro will zero a given value if and only if the expression evaluates to false, otherwise it passes through unchanged. This is used extensively, as a replacement for control-flow-mediated assignment, to combine different possible result values for a mathematical operation into a final value. Evaluating the expression cannot result in a branch. The result of the expression is forced to 1 or 0 via `!!`, and `MASK_UNLESS` then uses the `SIGN_EXTEND` macro to generate a mask that is all 1 or all 0 bits to control the final value. Finally, the mask is combined with the initial value via binary `and` (`&`). This is only a single, rather simple example of the style of coding necessary to generate code that can even be argued to run in constant time. See Figure 12 for an example of our C code with macros fully expanded.

```
int64_t fix_to_int64(fixed op1) {
  return ({ uint8_t isinfpos = (((op1
      )&((fixed) 0x3)) == ((fixed) 0
      x2)); uint8_t isinfneg = (((op1
      )&((fixed) 0x3)) == ((fixed) 0
      x3)); uint8_t isnan = (((op1)
      &((fixed) 0x3)) == ((fixed) 0x1
      )); uint8_t ex = isinfpos |
      isinfneg | isnan; fixed
      result_nosign = (({uint64_t
      SE_m__ = (1ull << ((64 - ((60 +
       2)))-1)); (((uint64_t) ((op1)
      >> ((60 + 2)))) ^ SE_m__) -
      SE_m__;}) + !!( (!!((op1) & (1
      LL << (((60 + 2))-1))) & !!((
      op1) & ((1LL << (((60 + 2))-1))
      -1))) | ((((op1) >> (((60 + 2))
      -2)) & 0x6) == 0x6) )); ((({
      uint64_t SE_m__ = (1ull << ((1)
      -1)); (((uint64_t) (!!(isinfpos
      ))) ^ SE_m__) - SE_m__;}) &
      (9223372036854775807LL)) | (({
      uint64_t SE_m__ = (1ull << ((1)
      -1)); (((uint64_t) (!!(isinfneg
      ))) ^ SE_m__) - SE_m__;}) &
      ((-9223372036854775807LL -1)))
      | (({uint64_t SE_m__ = (1ull <<
       ((1)-1)); (((uint64_t) (!!(!ex
      ))) ^ SE_m__) - SE_m__;}) & (
      result_nosign))); });
}
```

Figure 12: Conversion of a LibFTFP value to an `int64`, after the C pre-processor has been run.

This style of coding for LibFTFP causes most compilers to output assembly conforming to our above specifications. Unfortunately, we cannot guarantee that *any* compiler will output such assembly. Users should be careful to use only the build files we have provided, and run the provided correctness tests. As a best possible effort, we are distributing a binary copy of the LibFTFP shared library, built for AMD64 Linux. This binary copy has been exhaustively manually verified via disassembly to not use any known variable time instructions or control flow structures. This, of course, assumes that the target platform has a constant time integer unit, and that basic x86 instructions are constant time. Unless users are willing to verify their local builds to this degree, we suggest using only the distributed binary version of LibFTFP.

Due to our conservative coding style, LibFTFP uses only 39 distinct x86 instructions. The full list can be found in Figure 13.

Opcodes			
add	mov	pop	setg
and	movabs	push	setl
call	movsd	rep	setle
cdqe	movsx	ret	setne
cmp	movsxd	sar	shl
imul	movzx	sbb	shr
je	mul	seta	sub
jmp	neg	setae	test
jne	not	setbe	xor
lea	or	sete	

Figure 13: Every x86 instruction used by LibFTFP.

With regards to performance, running times (in cycles) for each of LibFTFP's operations (and their SSE counterparts, where available) can be found in Figure 14. We also include the running times for the same operations using native SSE assembly, as well as example operations from the multiple precision floating point library MPFR. While constant-time software operation does, in fact, take longer than optimized hardware, LibFTFP offers enough performance to be usable outside of the academic setting. By allowing the use of some approximations, it usually runs faster than the very precise, but extremely variable time MPFR.

To generate these numbers, we timed performance carefully, making sure to warm up both the cache and CPU frequency scaling. Each function is tested by taking a cycle count using `rdtsc` before and after running the function 2,000,000 times. Each test runs twice in succession, discarding the first set of results to warm the cache. The overhead of running the loop without the function call is then subtracted, and the remaining time is divided by the number of runs to obtain an average cycles-per-call.

D. Real-World Implementation

To determine LibFTFP's suitability for use in real-world programs, we modified the Fuzz differentially-private database and its Caml Light compiler to use `fixed`s rather than doubles as its non-integer data type. The small, streamlined nature of Caml Light made this modification fairly easy, adding or modifying around 120 lines of code in Caml Light itself.

We also had to modify Fuzz's custom additions and library functions. This mostly consisted of writing a more constant-time `cbagsum` and approach to number handling: originally, for each row, Fuzz serialized the microquery's `double` output as a string, and called `atof` on each number. `atof` is a variable-time function (intuitively, "0" is easier and faster to parse than "3.145e-60"), and so we replaced this human-readable information passing with a binary encoding of each `fixed`s bits.

Function	FTFP	SSE	MPFR
neg	6	5	12-20
abs	9	4	10-17
cmp	21	5	10-15
add	15	4	15-58
sub	15	5	14-61
mul	43	5	16-76
div	381	7-15	15-170
floor	8	5	12-48
ceil	11	5	12-56
exp	1,460	7-16	37-13,330
ln	681	11-20	18-6,900
log2	679	9-20	19-24,000
log10	674	9-21	19-18,000
sqrt	7,870	7-16	9-154
pow	2,330	11-78	40-72,000
sin	1,998	–	11-33,000
cos	1,990	–	34-29,000
tan	2,380	–	13-37,000
print	443	350-600	210-230

Figure 14: LibFTFP performance tests, as compared against the same operations via SSE and the multiprecision floating point library MPFR. Measured in cycles per function call on an Intel Core i7 2635QM at 2.00GHz. MPFR was configured with 62 bits of precision, and a few sample inputs were chosen; ranges may not be completely accurate. Note that MPFR's results are exactly correct, where LibFTFP approximates some values.

Our custom version of Fuzz computes all of our database queries from Section IV-E2, malicious or not, in 50.717 s–50.771 s. We attempted to customize our timing attack query for LibFTFP (as opposed to subnormals), but were unable to cause any appreciable timing difference. The original Fuzz, using `double`s, completes the queries in 50.300 s–51.552 s. While Fuzz's overall running times are not the most enlightening comparison (since so much work was spent making each microquery take exactly the same amount of time), we think that this shows LibFTFP is capable of handling important mathematical calculations without sacrificing too much raw performance.

VI. Related Work

In our survey of related work, we focus on side-channel attacks, in which an unwilling victim's secret information is revealed, rather than covert-channel attacks where two cooperating processes communicate despite the presence of a monitor; on timing attacks, in which secret information is revealed by how long a process takes to run, rather than through, e.g., power draw or electromagnetic emissions; and on attacks on software and general purpose computing platforms, rather than pure hardware implementations.

Code Paths: Timing side-channel attacks on cryptographic software were introduced by Kocher in a seminal 1996 paper [33]. The most straightforward mechanism for timing side-channel attacks is when software takes different code paths depending on secret values; Kocher's concrete example was the choice (based on secret key bits) of whether to multiply in a round of RSAREF's square-and-multiply exponentiation routine. In some cases such attacks are feasible even over the network [11, 13].

Memory Accesses: A second mechanism for timing side-channel attacks is when the memory access pattern of software or its use of microarchitectural functional units varies depending on secret values. Kocher's suggestion that this class of attacks might be feasible has been more than borne out; see Acıiçmez and Koç's extensive survey [1], which describes attacks that take advantage of the data cache, the instruction cache, the branch prediction unit, and functional unit contention. Unlike simple timing attacks, microarchitectural timing attacks usually require an observer process to run on the same machine as the victim; virtual-machine co-tenancy in a cloud environment can suffice [53].

Data Timing Channels: A third mechanism for timing side-channel attacks is for individual instructions to take a variable amount of time depending on secret inputs. Kocher hypothesized that, on some platforms, integer multiplication and rotation instructions might have variable running time, putting implementations of ciphers like IDEA and RC5 at risk. In 2000, Hachez and Quisquater noted in passing that the ARM7M core implements 32-bit multiplication using four applications of a 32×8 functional unit, terminating early if the most significant bits of one operand are zero [25]; Großschädl et al. [24] showed that such partial multiplier designs are common in small embedded cores, and that early termination gives rise to a side channel. Großschädl et al. exploited the early termination together with SPA power traces to break implementations of AES, RC6, RSA, and ECIES on the ARM7TDMI core. Note that while early termination induces a timing side channel, Großschädl et al.'s attack model was more invasive, requiring power traces. We are not aware of any prior work that exploits instructions with data-dependent timing through timing alone.

For programs expressed in a high-level language, timing channels may arise from interactions between layers in the software stack. For example, as shown by Barbosa et al. [6], JIT compilation may cause two branches that perform the same high-level operations to have different runtime performance.

Timing attacks are also relevant beyond crypto software. For example, timing attacks have been shown to reveal sensitive information such as a user's browsing history [22], the number of private photos in a Web gallery [12], what signature database a user's antivirus program runs [4], and how many items are in a user's shopping cart [54].

Mitigations: Due to the serious ramifications of timing channel attacks, there is a wide literature on ways to defend against them. Roughly speaking, they fall under the categories of static and dynamic mitigations.

One approach is to use a typing discipline to ensure that all control flow paths have the same number of instructions, by ensuring that conditionals have equal sized branches, and prohibiting the use of secret information in loop guards, i.e., all loop guards are constant or only depend on public, non-secret values [47, 48, 51]. If the type system rejects a program because it has "uneven" branches, the program can still be *transformed*, for example by adding suitable "padding" instructions along shorter branches [2, 9, 10, 28], by using "conditional execution" implemented via bit-masking and ternary choice [39] or by using if-conversion [15]. All of the above approaches are limited to situations where the instruction count is a proxy for actual performance, and do not protect against lower level, e.g., instruction cache attacks [1] or the data timing variation attacks we demonstrated.

Purely static or compilation methods are unlikely to be effective against attacks that exploit the timing behavior of microarchitectural entities like branch predictors or caches [1]. One approach to thwarting such attacks is to modify the hardware [34], OS, or use a virtualization layer [32] to ensure that certain cache lines containing secret data are never evicted. Another alternative, called *secure multi-execution*, uses multiple threads to simultaneously execute all the different branches of code that depend on secret data, but using different *values* that represent projections (or facets) of the values at different security levels [18]. By then controlling the scheduler, one can ensure that a deterministic number of steps are taken at each security level [31].

An orthogonal approach is to ensure the absence of hardware based timing channels by synthesizing the hardware from description languages equipped with a notion of non-interference [36]. While this approach is invasive, it could eliminate timing variations at the hardware source.

Black-box Mitigation: Another, more general, approach, which could in principle account for *any* timing channel, is to treat the machine as a black box emitting observable *events* and to interpose a *mitigation layer* that pauses the output of events to make the output timing deterministic [5]. The main drawback with this approach is the large overhead imposed by the pauses. To get around this, one can use a *gray-box* language based approach where the mitigator is exposed as a language primitive `mitigate(e) {c}` where the command `c` is executed and a pause is inserted until `e` time units have elapsed. The resulting system can guarantee the absence of timing leaks, as long as the duration `e` is independent of secret data, and regardless of the computations performed in `c`, overcoming the loop-restrictions in the original static approaches. Furthermore, the pauses are only inserted at specific places where the static methods are insufficient [52].

VII. Conclusion

In this paper, we have shown how an arcane detail about timing variations in floating point operations opens up a data timing side channel that can be used to break the security of real world systems, including a Web browser and a differentially private database carefully designed to block such attacks. While numerical analysts have known about these timing variations for decades, our results indicate that that data timing channels are a viable vector for exfiltrating sensitive information, for which, currently, there is no form of detection, let alone prevention, and which therefore warrant attention from the security community. In particular, we hope that future work will: (1) reexamine how security-relevant software relies on floating point operations, not just for timing variation but also determinism (see, e.g., [16, 17]); (2) perform a systematic and comprehensive evaluation of the variation in the way other kinds of instructions run on different inputs and on different architectures such as GPGPUs, with the goal of understanding how these variations can be used for data timing channel-based exfiltration attacks and other security concerns like fingerprinting; and (3) identify patterns for data timing vectors that can be the basis of static or dynamic mitigation tools, using language based techniques for compiling or transforming away potential channels, or run-time techniques for rewriting binaries or virtualizing problematic operations to block data timing channels.

Acknowledgements

We thank Eric Rescorla and Stefan Savage for helpful discussions about this work.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1228967, and by a gift from the Mozilla Corporation.

References

[1] O. Acıiçmez and Ç. K. Koç, "Microarchitectural attacks and countermeasures," in *Cryptographic Engineering*, Ç. K. Koç, Ed. Springer-Verlag, 2009, ch. 18, pp. 475–504.

[2] J. Agat, "Transforming out timing leaks," in *Proceedings of POPL 2000*, T. Reps, Ed. ACM Press, Jan. 2000, pp. 40–53.

[3] B. Akbarpour, A. T. Abdel-Hamid, S. Tahar, and J. Harrison, "Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL," *The Computer Journal*, vol. 53, no. 4, pp. 465–488, May 2010.

[4] M. I. Al-Saleh and J. R. Crandall, "Application-level reconnaissance: Timing channel attacks against antivirus software," in *Proceedings of LEET 2011*, C. Kruegel, Ed. USENIX, Mar. 2011.

[5] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of CCS 2010*, A. Keromytis and V. Shmatikov, Eds. ACM Press, Oct. 2010, pp. 297–307.

[6] M. Barbosa, A. Moss, and D. Page, "Constructive and destructive use of compilers in elliptic curve cryptography," *J. Cryptology*, vol. 22, no. 2, pp. 259–81, Apr. 2009.

[7] L. D. Baron, "Preventing attacks on a user's history through CSS :visited selectors," Apr. 2010, online: http://dbaron.org/mozilla/visited-privacy.

[8] A. Barth, C. Jackson, and J. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of CCS 2008*, P. Syverson and S. Jha, Eds. ACM Press, Oct. 2008, pp. 75–88.

[9] G. Barthe, T. Rezk, and M. Warnier, "Preventing timing leaks through transactional branching instructions," *Electron. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 33–55, May 2006.

[10] G. Barthe, G. Betarte, J. Diego, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of CCS 2014*, M. Yung and N. Li, Eds. ACM Press, Nov. 2014.

[11] N. T. Billy Bob Brumley, "Remote timing attacks are still practical," in *Proceedings of ESORICS 2011*, ser. LNCS, V. Atluri and C. Diaz, Eds., vol. 6879. Springer-Verlag, Sep. 2011, pp. 355–71.

[12] A. Bortz, D. Boneh, and P. Nandy, "Exposing private information by timing Web applications," in *Proceedings of WWW 2007*, P. Patel-Schneider and P. Shenoy, Eds. ACM Press, May 2007, pp. 621–28.

[13] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–16, aug 2005.

[14] J. Coonen, W. Kahan, J. Palmer, T. Pittman, and D. Stevenson, "A proposed standard for binary floating point arthmetic," *SIGNUM Newsl.*, vol. 14, no. si-2, pp. 4–12, Oct. 1979.

[15] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of IEEE Security and Privacy ("Oakland") 2009*, A. Myers and D. Evans, Eds. IEEE Computer Society, May 2009, pp. 45–60.

[16] B. Dawson, "Floating-point determinism," Online: http://randomascii.wordpress.com/2013/07/16/floating-point-determinism/, Jul. 2013, fetched: Nov 14, 2014.

[17] ——, "Intel underestimates error bounds by 1.3 quintillion," Online: http://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/, Oct. 2014, fetched: Nov 14, 2014.

[18] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proceedings of IEEE Security and Privacy ("Oakland") 2010*, D. Evans and G. Vigna, Eds. IEEE Computer Society, May 2010, pp. 109–24.

[19] I. Dooley and L. Kale, "Quantifying the interference caused by subnormal floating-point values," in *Proceedings of OSIHPA 2006*, M. Sottile, F. Petrini, and R. Mraz, Eds., Sep. 2006, online: http://osihpa.cs.utep.edu/2006/DooleySubnormal06.pdf.

[20] C. Dwork, "A firm foundation for private data analysis," *Commun. ACM*, vol. 54, no. 1, pp. 86–95, Jan. 2011.

[21] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, Aug. 2014.

[22] E. W. Felten and M. A. Schneider, "Timing attacks on Web privacy," in *Proceedings of CCS 2000*, S. Jajodia, Ed. ACM Press, Nov. 2000, pp. 25–32.

[23] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[24] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *Proceedings if ICISC 2009*,

ser. LNCS, D. Lee and S. Hong, Eds., vol. 5984. Springer-Verlag, 2010, pp. 176–92.

[25] G. Hachez and J.-J. Quisquater, "Montgomery exponentiation with no final subtractions: Improved results," in *Proceedings of CHES 2000*, ser. LNCS, Ç. K. Koç and C. Paar, Eds., vol. 1965. Springer-Verlag, Aug. 2000, pp. 293–301.

[26] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire," in *Proceedings of USENIX Security 2011*, D. Wagner, Ed. USENIX, Aug. 2011, pp. 507–21.

[27] M. Harris, "CUDA pro tip: Flush denormals with confidence," Online: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-flush-denormals-confidence/, Jan. 2013, fetched: Nov 13, 2014.

[28] D. Hedin and D. Sands, "Timing aware information flow security for a JavaCard-like bytecode," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 1, pp. 163–82, Dec. 2005.

[29] "FDIV replacement program: Description of the flaw," Whitepaper: Onnline: http://www.intel.com/support/processors/pentium/sb/CS-013007.htm, Intel, Jul. 2004, fetched: Nov 12, 2014.

[30] W. Kahan, "Why do we need a floating-point arithmetic standard?" Whitepaper: Online: http://www.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf, Feb. 1981, fetched: Nov 12, 2014.

[31] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and termination-sensitive secure information flow: Exploring a new approach," in *Proceedings of IEEE Security and Privacy ("Oakland") 2011*, G. Vigna and S. Jha, Eds. IEEE Computer Society, May 2011, pp. 413–28.

[32] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of USENIX Security 2012*, T. Kohno, Ed. USENIX, Aug. 2012, pp. 189–204.

[33] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proceedings of Crypto 1996*, ser. LNCS, N. Koblitz, Ed., vol. 1109. Springer-Verlag, Aug. 1996, pp. 104–13.

[34] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Architecting against software cache-based side-channel attacks," *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1276–88, Jul. 2013.

[35] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using CSS filters," in *Proceedings of CCS 2013*, V. Gligor and M. Yung, Eds. ACM Press, Nov. 2013, pp. 1055–62.

[36] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," in *Proceedings of PLDI 2011*, S. Blackburn, Ed. ACM Press, Jun. 2011, pp. 109–20.

[37] F. McSherry, "Privacy integrated queries," in *Proceedings of ACM SIGMOD 2009*, A. Labrinidis, Ed. ACM Press, Jun. 2009.

[38] I. Mironov, "On significance of the least significant bits for differential privacy," in *Proceedings of CCS 2012*, G. Danezis and V. Gligor, Eds. ACM Press, Oct. 2012, pp. 650–61.

[39] 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 *Proceedings of ICISC 2005*, ser. LNCS, D. Won and S. Kim, Eds., vol. 3935. Springer-Verlag, Feb. 2006, pp. 156–68.

[40] J. S. Moore, T. W. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5$_K$86 floating-point division program," *IEEE Trans. Computers*, vol. 47, no. 9, pp. 913–26, Sep. 1998.

[41] "NVIDIA's next generation CUDA compute architecture: Fermi," Whitepaper: Online: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, NVIDIA Corporation, 2009.

[42] R. Regan, "Bug #53632: PHP hangs on numeric value 2.2250738585072011e-308," Online: https://bugs.php.net/bug.php?id=53632, Dec. 2010, fetched: Nov 12, 2014.

[43] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for mapreduce," in *Proceedings of NSDI 2010*, M. Castro and A. C. Snoeren, Eds. USENIX, Mar. 2010.

[44] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS J. Comput. Math.*, vol. 1, pp. 148–200, 1998.

[45] ——, "A mechanically checked proof of correctness of the AMD K5 floating point square root microcode," *Formal Methods in System Design*, vol. 14, no. 1, pp. 75–125, Jan. 1999.

[46] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," in *Proceedings of W2SP 2010*, C. Jackson, Ed., May 2010. [Online]. Available: http://seclab.stanford.edu/websec/framebusting/framebust.pdf

[47] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of CSFW 2000*, ser. CSFW '00, P. F. Syverson, Ed. IEEE Computer Society, Jul. 2000, pp. 200–14.

[48] G. Smith, "A new type system for secure information flow," in *Proceedings of CSFW 2001*, S. Schneider, Ed. IEEE Computer Society, Jun. 2001, pp. 115–25.

[49] P. Stone, "Pixel perfect timing attacks with HTML5," Presented at Black Hat 2013, Jul. 2013, online: http://contextis.co.uk/documents/2/Browser_Timing_Attacks.pdf.

[50] ——, "Bug 711043 – (CVE-2013-1693) SVG filter timing attack," Online: https://bugzilla.mozilla.org/show_bug.cgi?id=711043, Jun. 2011, fetched: Nov 13, 2014.

[51] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proceedings of CSFW 1997*, S. Foley, Ed. IEEE Computer Society, Jun. 1997, pp. 156–69.

[52] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of PLDI 2012*, F. Tip, Ed. ACM Press, Jun. 2012, pp. 99–110.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of CCS 2012*, G. Danezis and V. Gligor, Eds. ACM Press, Oct. 2012, pp. 305–16.

[54] ——, "Cross-tenant side-channel attacks in PaaS clouds," in *Proceedings of CCS 2014*, M. Yung and N. Li, Eds. ACM Press, Nov. 2014.