UNIVERSITY OF CALIFORNIA SAN DIEGO

Trusted Systems for Uncertain Times

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

in

Computer Science

by

David William Kohlbrenner

Committee in charge:

Professor Hovav Shacham, Chair
Professor Ranjit Jhala
Professor Farinaz Koushanfar
Professor Dean Tullsen
Professor Geoff Voelker

2018

The Dissertation of David William Kohlbrenner is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

DEDICATION

For Nina

EPIGRAPH

Tlön may be a labyrinth,
but it is a labyrinth plotted by men,
a labyrinth destined to be deciphered by men.

Jorge Luis Borges

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, thank you to my incredible spouse, Nina, who supported me in every way throughout my PhD. Perhaps most directly in explaining data visualization concepts and why beige and navy is not an ideal color scheme.

A much deserved thank you to my advisor, Hovav Shacham, who has guided my research interests and supported them as they meandered.

A special thanks to Keaton Mowery, who was not just a co-author and fellow student, but taught me how to be a successful graduate student.

To Joe DeBlasio, it is perhaps unfortunate that our studies were not in international law and politics, or we might both have been done earlier.

Office 3140 residents, current and former, deserve all of the hype the office gets. Grad school would not have been the same without you all.

To all my co-authors: Keaton Mowery, Michael Wei, Hovav Shacham, Steve Swanson, Mark Andrysco, Ranjit Jhala, Sorin Lerner, Craig Disselkoen, Leo Porter, and Dean Tullsen, thank you.

An additional thanks to all those who gave input on all my projects, successful and not, there are too many of you to name.

Chapters 1 and 3, in part, are a reprint of the material as it appears in IEEE Security and Privacy (Oakland) 2015. Andrysco, Mark; Kohlbrenner, David; Mowery, Keaton; Jhala, Ranjit; Lerner, Sorin; Shacham, Hovav, 2015. The dissertation author was a primary investigator and a primary author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in USENIX Security 2017. Kohlbrenner, David; Shacham, Hovav, 2017. The dissertation author was the primary investigator and the primary author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in USENIX Security 2016. Kohlbrenner, David; Shacham, Hovav, 2016. The dissertation author was the primary investigator and the primary author of this paper.

VITA

2011	Bachelor of Science, Computer Science Carnegie Mellon University
2011-2015	Research Assistant University of California San Diego
2015	Master of Science, Computer Science University of California San Diego
2015-2018	Research Assistant University of California San Diego
2018	Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

D. Kohlbrenner and H. Shacham "On the effectiveness of mitigations against floating-point timing channels." USENIX Security. August 2017

C. Disselkoen, D. Kohlbrenner, L. Porter, D. Tullsen "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX" USENIX Security. August 2017

D. Kohlbrenner and H. Shacham "Trusted Browsers for Uncertain Times." USENIX Security. August 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

K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson "Welcome to the Entropics: Boot-Time Entropy in Embedded Devices." IEEE Security and Privacy (Oakland), May 2013

ABSTRACT OF THE DISSERTATION

Trusted Systems for Uncertain Times

by

David William Kohlbrenner

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Hovav Shacham, Chair

When software is designed, even with security in mind, assumptions are made about the details of hardware behavior. Unfortunately, the correctness of such assumptions can be critical to the desired security properties. In this dissertation we first demonstrate how incorrect assumptions about the hardware abstraction lead to side-channels that threaten modern software security, and second we propose a principled method of timing channel defense for modern web browsers.

We show how performance variations in floating-point math instructions enable the first demonstrated instruction-data timing side-channel on commodity hardware. We use this side-channel in two case studies to prove it's viability. First, we redesign a previous attack on an

older version of the Firefox web browser to violate the Same Origin Policy. Second, we break the guarantees of a differentially private database designed to resist timing attacks. We show how the timing side-channel arises from hardware optimization decisions that have been well understood in the architecture, numerical analysis, and game-engine communities, but largely ignored in security.

Using a detailed measurement and analysis of floating-point performance, we examine the progress and potential of defenses against floating-point timing side-channels. We find that all deployed defensive schemes for desktop web browsers were insufficient, and most are still vulnerable. Using the same analysis methods, we show how a proposed defensive scheme makes incorrect assumptions about the hardware features it leverages, negating its guarantees.

As a possible remediation to the problem of floating-point timing side-channels, we present `libfixedtimefixedpoint` as an alternative to floating-point. It provides a fixed-point implementation of most available floating-point operations and is designed to run in constant time regardless of the input values.

Finally, we discuss structural problems in modern web browser design that make them amenable to all timing attacks. Adapting solutions from parallel problems solved by early trusted operating systems projects, we propose a modified browser architecture providing a provable defensive guarantee against all timing attacks. We then demonstrate the viability of this scheme by prototyping aspects of the architecture in a modified web browser.

Introduction

Computer science is built on layered abstractions. Each step up, from transistor to scripting language, requires abstracting away the lower layers to some degree. Layered abstraction has been, and is, one of the great successes of computer science and engineering as disciplines. As David Wheeler once said: "We can solve any problem by introducing an extra level of indirection."

But in security research we focus on where assumptions are made and there is no greater trove of assumptions in computing than the hardware abstractions we rely on. It is an unfortunate reality that our software executes on the same silicon gates that we abstract away to make it possible to write. As the security community has deepened its understanding of how to measure hardware's behaviors the recent years have seen a rise in attacks on systems that were, by reasonable previous evaluations, well designed and security aware. New techniques for measurement, reverse-engineering of hardware, and increased community understanding of the components of our modern computers have all made this acceleration possible.

This dissertation is a slice of that progress, with a specific focus on understanding the true behavior of the FPU (Floating-Point Unit) and on adapting the lessons of the trusted operating system efforts of the late 1980s to mitigate timing side-channels. Our attack structures, defensive designs, and measurement techniques have all contributed to the growth of this area, with other researchers responding and improving each.

We accomplish two goals: to demonstrate that instruction-data timing side-channels are a real threat to modern software, and to show how principled degradation of clocks can mitigate timing side-channels in web browsers. Specifically, we demonstrate a series of vulnerabilities

spread over years in most major desktop web browsers and other software caused by timing variability in floating-point operations. As well as propose and demonstrate a prototype of a new browser architecture that relies on timing defense concepts from the VAX secure kernel project.

Outline

Chapter 1 presents a new type of timing side-channel attack that uses the execution time of individual floating-point math operations to attack software, specifically the Firefox web browser and the Fuzz differentially private database scheme.

Chapter 2 introduces a new strategy for evaluating floating-point performance with a focus on identifying value classes that can be used in timing side-channels. This evaluation is also applied to existing and proposed defenses, and finds small timing channels in them. As part of this evaluation of defenses we present new floating-point timing side-channel attacks against Firefox, Chrome, and Safari, despite their respective mitigations.

Chapter 3 presents a potential solution to the floating-point timing side-channel, `libfixedtimefixedpoint` (`libftfp`), a constant time fixed-point math library. We evaluate the library's performance, and discuss difficulties in constructing constant-time math.

Finally, Chapter 4 describes a defensive model for web browsers adapted from the VAX secure kernel project. We motivate this by demonstrating a series of time-measurement techniques for web browsers to bypass deployed defenses by both improving the accuracy of degraded clocks, as well as by using unorthodox browser interfaces as clocks. The VAX fuzzy time defensive scheme is shown to be effective at defeating our new techniques, and is shown to be applicable to the web browser context through a verifiable design proposal (Fermata) and an engineering prototype (Fuzzyfox).

Chapter 1

On Subnormal Floating Point and Abnormal Timing

1.1 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 [52], but the intervening years have yielded only one exploitable example: integer multiplication on some small-die embedded processors [38].

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 `request-`

`AnimationFrame` API was added), the Firefox browser has 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 [81]. We show that floating point data timing channels in the computation of filters (without any data-dependent branches) enable similar attacks. Our attack applies to Firefox versions 23 through 27, including 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. [40] 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 [63], 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 [38] relied on SPA power traces to amplify the timing signal, hence requiring invasive access to the system.

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 1.2),
- 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 1.3) and the Fuzz differentially private database (Section 1.4).

1.2 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 [47]. In 1985, the Institute of Electrical and Electronics Engineers published a technical standard for floating point formats: IEEE-754 [20]. 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 [1], and led to efforts to verify hardware implementations [65, 75, 74, 5]. 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 [71].

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.

Table 1.1. IEEE-754 Formats

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	$6.5e{-}4996$	$3.36e{-}4932$	$1.19e4932$

1.2.1 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. Table 1.1 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. Table 1.2 summarizes the special values and their encoding.

1.2.2 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

Table 1.2. IEEE-754 Special Value Encoding

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

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.

1.2.3 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× [26]. 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 [2]. NVIDIA has stated that, consequently, certain operations can suffer from performance problems when operating on subnormal values [42], 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.

1.2.4 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

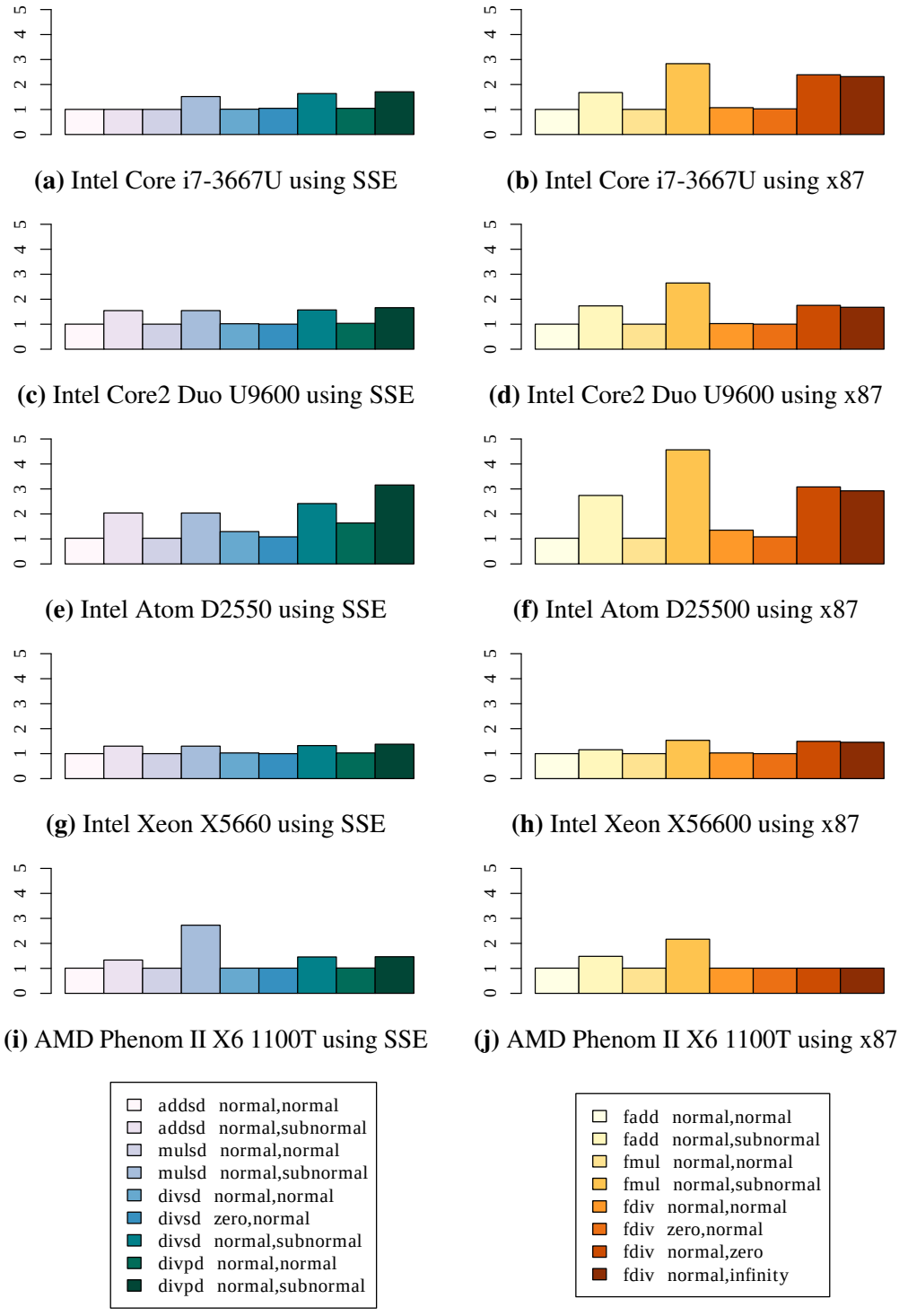
operation on the two registers, and storing the result back into memory. This load-operation-store cycle corresponds closely with code likely to be found in the wild. We averaged 1000 runs for each combination of instructions and operands, and all results are consistent and reproducible.

Figure 1.1 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.

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,



(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 1.1. 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.

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.

1.2.5 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 [33, 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.

1.3 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.

1.3.1 A History of Stolen Pixels

In 2013, Paul Stone [81] (and, independently, Kotcher et al. [55]) 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 `iframes`. 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 `iframes`, 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 1.3.2.

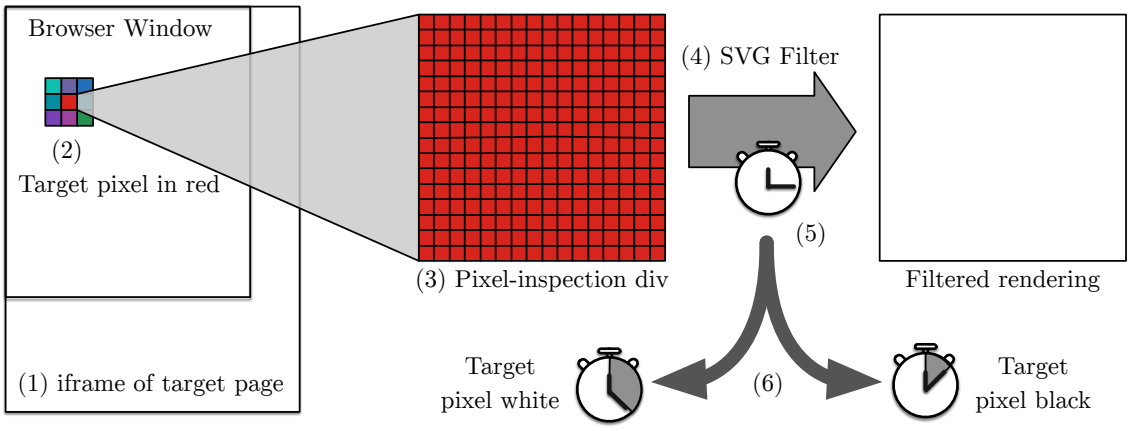


Figure 1.2. Cross-Origin SVG Filter Pixel Stealing Attack in Firefox

1.3.2 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 1.2.4. 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 1.2.

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.

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 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 1.2.4) 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.

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.

1.3.3 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 [81], 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 [12]. 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 [76].

Firefox 30 and onwards[1] 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 [81]. Once in possession of CSRF tokens, the attacking page can mount standard CSRF attacks [13].

1.3.4 Attack Implementation and Measurement

We developed a test page version of the attack described in Section 1.3.2, 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

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

Table 1.3. Firefox Checkerboard Recovery 32-bit

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

Table 1.4. Firefox Checkerboard Recovery 64-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

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. Tables 1.3 and 1.4 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 table 1.3 has several entries with very high black errors. These are entirely due

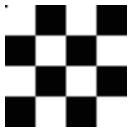


Figure 1.3. Stealing a 48×48 pixel checkerboard

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 table 1.3, 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 1.3 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 1.4 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 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 [81], with intelligent selection of pixels, OCR can be run reading only $log_2(N)$ pixels per character for a target font with N

Figure 1.4. Stealing a 48×48 pixel region from www.bbc.com, at 100×100, 200×200, and 300×300 pixel-inspection `div` size.

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.

1.3.5 Vulnerable Browsers

While the attack described in Section 1.3.2 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. Other major browsers run filters on some combination of GPU and/or CPU. We do not investigate the vulnerability of other browsers in this paper. While some GPUs [42] exhibit similar timing differences, our test design was unable to detect them.

to this attack from version 23 (released August 6, 2013) through 27. From Firefox 28 (released March 18, 2014) onward, the SVG filter implementation changed, and the convolution filter switched to fixed-point arithmatic. 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 1.2.4 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.

1.3.6 Firefox Response

The original Mozilla SVG filter timing attack bug thread [80] 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[2] that there was no particular reason to believe that GPUs would be constant time where CPUs were not. Jacob has noted[3] 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.

1.3.7 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 1.2.4

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

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

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 [80]) are likely to make such filters unusable even if developed.

of the CSS filters specification[4] 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[5] 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 with a new partially fixed-point implementation. The eventual move to the GPU should not be considered a fix. As Mark Harris, NVIDIA's Chief Technologist for GPU Computing [42] notes, some GPUs do exhibit measurable performance impact with subnormal values; see Section 1.2.3 for more. We believe that as page-visible timing precision improves, even GPU floating point calculations (as used in other browsers) will become vulnerable.

1.4 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

[4] https://dvcs.w3.org/hg/FXTF/raw-file/705f723192d2/filters/Overview.html
[5] https://dvcs.w3.org/hg/FXTF/raw-file/4b53107dd95d/filters/index.html

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 [27, 28] 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.

1.4.1 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 [27, 28].

1.4.2 Differential Privacy Databases

Several groups have used the theory of differential privacy to construct *differentially private databases*, like PINQ [62] and Airavat [73], 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.

1.4.3 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. [40] 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.

1.4.4 Restoring Privacy by Eliminating Timing Channels

Haeberlen et al. [40] 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 [40].

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 "time-out" 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[6] runtime to compile it into a 32-bit x86 executable, suitable for executing on a database.

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.

[6] 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 1.5. C implementation of `bagsum`, Fuzz's function to aggregate the results of per-row query computation. Attacker-controlled values are highlighted.

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 1.5. 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 [40].

1.4.5 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.

When we look closely at the implementation of `cbagsum` (Figure 1.5), 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`.

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 1.2.4, due to timing differences in floating point

Table 1.5. Fuzz query wall-clock duration. Each query was run 4 times on a database of 1M rows. The probing query was run twice: on a database which contained the row of interest and a database that did not. The non-probing queries produce a constant value for each 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

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.

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 Table 1.5. 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.

1.5 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 [52]. 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 [18, 16].

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 [3], 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 [91].

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 [39]; Großschädl et al. [38] 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. [11], 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 [30], the number of private photos in a Web gallery [17], what signature database a user's antivirus program runs [6], and how many items are in a user's shopping cart [92].

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 [85, 77, 79]. 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 [4, 15, 43, 14], by using "conditional execution" implemented via bit-masking and ternary choice [64] or by using if-conversion [21]. 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 [3] 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 [3]. One approach to thwarting such attacks is to modify the hardware [54], OS, or use a virtualization layer [51] 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 [25]. By then controlling the

scheduler, one can ensure that a deterministic number of steps are taken at each security level [50].

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 [58]. 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 [9]. 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 [90].

1.6 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., [22, 23]); (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.

Chapter 1, in part, is a reprint of the material as it appears in IEEE Security and Privacy (Oakland) 2015. Andrysco, Mark; Kohlbrenner, David; Mowery, Keaton; Jhala, Ranjit; Lerner, Sorin; Shacham, Hovav, 2015. The dissertation author was a primary investigator and a primary author of this paper.

Chapter 2

On the effectiveness of mitigations against floating-point timing channels.

2.1 Introduction

The time a modern processor takes to execute a floating-point instruction can vary with the instruction's operands. For example, subnormal floating-point values consumed or produced by an instruction can induce an order-of-magnitude execution slowdown. In 2015, Andrysco et al. [8] exploited the slowdown in subnormal processing to break the privacy guarantees of a differentially private database system and to mount pixel-stealing attacks against Firefox releases 23–27. In a pixel-stealing attack, a malicious web page learns the contents of a web page presented to a user's browser by a different site, in violation of the browser's origin-isolation guarantees.

Andrysco et al. proposed mitigations against floating-point timing attacks:

- Replace floating-point computations with fixed-point computations relying on the processor's integer ALU.
- Use processor flags to cause subnormal values to be treated as zero, avoiding slowdowns associated with subnormal values.
- Shift sensitive floating-point computations to the GPU or other hardware not known to be vulnerable.

At USENIX Security 2016, Rane, Lin, and Tiwari [70] proposed additional mitigations:

- Use program analysis to identify floating-point operations whose inputs cannot be subnormal; these operations will not experience subnormal slowdowns.
- Run floating-point operations whose inputs might be subnormal on the the processor's SIMD unit, loading the a SIMD lane with a dummy operation chosen to induce consistent worst-case execution time.

Rane, Lin, and Tiwari implemented their proposed mitigations in a research prototype Firefox browser. Variants of the Andrysco et al. mitigations have been adopted in the latest versions of Firefox, Safari, and Chrome.

We evaluate how effective the proposed mitigations are at preventing pixel stealing. We find that, other than avoiding the floating point unit altogether, the proposed mitigations are *not effective* at preventing pixel stealing — at best, they reduce the rate at which pixels can be read. Our attacks make use of details of floating point performance beyond the subnormal slowdowns observed by Andrysco et al.

Our contributions are as follows:

1. We give a more refined account of how floating-point instruction timing varies with operand values than did Andrysco et al. In particular, we show that operands with a zero exponent or significand induce small but exploitable speedups in many operations.
2. We evaluate the SIMD defense proposed by Rane, Lin, and Tiwari, giving strong evidence that processors execute the two operations sequentially, not in parallel.
3. We revisit browser implementations of SVG filters two years after the Andrysco et al. attacks. Despite attempts at remediation, we find that the latest versions of Chrome, Firefox, and Safari are all vulnerable to pixel-stealing attacks.
4. We show that subnormal values induce slowdowns in CUDA calculations on modern Nvidia GPUs.

Taken together, our findings demonstrate that the floating point units of modern processors are more complex than previously realized, and that defenses that seek to take advantage of that unit without creating timing side channels require careful evaluation.

Ethics and disclosure.

We have disclosed the pixel-stealing attacks we found to Apple, Google, and Mozilla. Mozilla has already committed to deploying a patch. We will give Apple and Google adequate time to patch before publishing our findings.

2.2 Background

Many floating point instructions are known to exhibit performance differences based on the operands. Andrysco et al. [8] leveraged these timing differences to defeat the claimed privacy guarantees of two systems: Mozilla Firefox (versions 23–27) and the Fuzz differentially private database. Andrysco et al.'s attack on Firefox, and the attacks on browsers we present, use SVG filter timing to break the Same-Origin Policy, an idea introduced by Stone [81] and Kotcher et al. [55].

2.2.1 IEEE-754 floating point

For the purposes of this paper we will refer to floating point, floats, and doubles to mean the IEEE-754 floating point standard (see Table 2.1) unless otherwise specified. The floating point unit (FPU) accessed via Intel's single scalar Streaming SIMD (Single Instruction, Multiple Data) Extensions (SSE) instructions adheres to this standard on all processors we discuss. We omit discussion of the x87 legacy FPU that is still accessible on a modern x86_64 processor.

The IEEE-754 floating point standard is the most common floating point implementation available on commodity CPUs. Figure 2.1 shows the layout of the IEEE-754 single precision float and the value calculation. Note that the actual exponent used in the 2^{exp} portion is $exponent - bias$ where the bias is half the unsigned maximum value of the exponent's range. This format allows

Table 2.1. IEEE-754 Format type ranges (Reproduced with permission from [8])

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	$4.9e{-}324$	$2.23e{-}308$	$1.79e308$

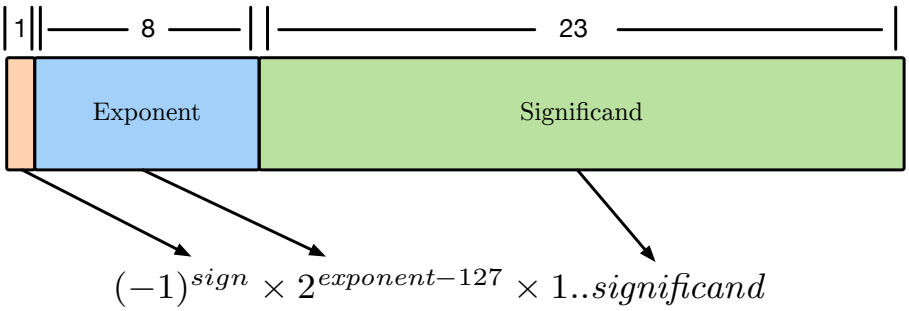


Figure 2.1. IEEE-754 single precision float

for the full range of positive and negative exponent values to be represented easily. If the exponent has any non 0 bits the value is *normal*, and the significand has an implicit leading 1 bit. If the exponent is all 0 bits (i.e., $exponent - bias = -bias$) then the value is *subnormal*, and there is no implicit leading 1 bit. As shown in Table 2.1 this means that subnormal values are fantastically small. Subnormal values are valuable because they enable gradual underflow for floating point computations. Gradual underflow guarantees that given any two floats, $a \neq b$, there exists a floating point value $c \neq 0$ that is the difference $a - b = c$. The use of this property is demonstrated by the simple pseudocode "if $a \neq b$ then $x \,/\, (a-b)$," which does not expect to generate an infinity by dividing by zero. Without subnormals the IEEE-754 standard could not guarantee gradual underflow for normals and a number of adverse scenarios such as the one above can occur. As Andrysco et al. [8] observe, subnormal values do not frequently arise, and special hardware or microcode is used to handle them on most CPUs.

Andrysco et al.'s attacks made use of the substantial timing differences between operations on subnormal (or denormal) floating point values and on normal floating point values. See Table 2.2 for a list of non-normal IEEE-754 value types. In this paper we present additional benchmarks

Table 2.2. IEEE-754 Special Value Encoding (Reproduced with permission from [8])

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

that demonstrate that (smaller) timing differences arise from more than just subnormal operands. Section 2.3 describes our benchmarking results.

2.2.2 SVG floating point timing attacks

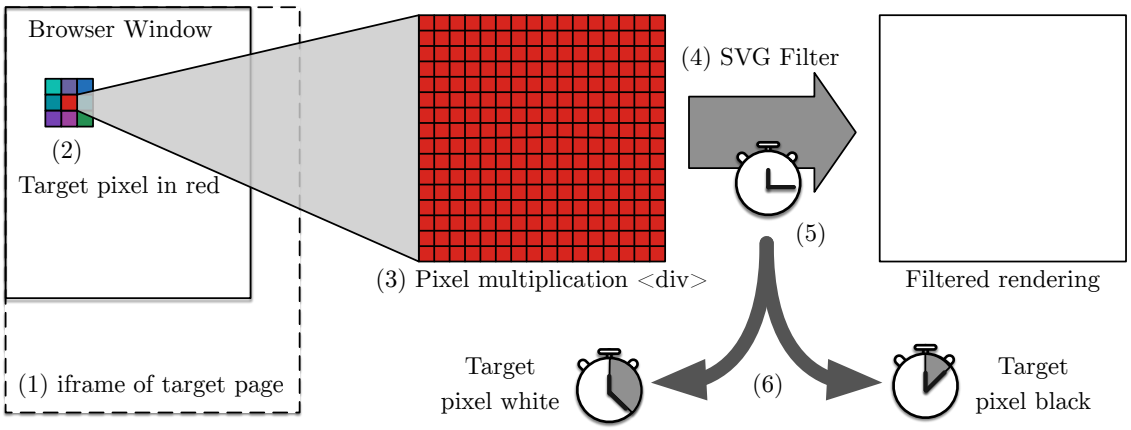


Figure 2.2. Cross-Origin SVG Filter Pixel Stealing Attack in Firefox, reproduced from [8] with permission

Andrysco et al. [8] presented an attack on Firefox SVG filters that is very similar to the attacks detailed later in this paper. Thus, we provide an overview of how that attack works for reference.

Figure 2.2 shows the workflow of the SVG timing attack.

1. The attacking page creates a large `iframe` of the victim page inside of a container `<div>`
2. The container `<div>` is sized to 1x1 pixel and can be scrolled to the current target pixel on the `iframe` using the `scrollTop` and `scrollLeft` properties.

3. The target pixel is duplicated into a larger container `<div>` using the `-mozelement` CSS property. This creates a `<div>` that is arbitrarily sized and consists only of copies of the target pixel.

4. The SVG filter that runs in variable time (`feConvolveMatrix`) is applied to the the pixel duplication `<div>`

5. The rendering time of the filter is measured using `requestAnimationFrame` to get a callback when the next frame is completed and `performance.now` for high resolution timing.

6. The rendering time is compared to the threshold determined during the learning phase and categorized as white or black.

Since the targeted `iframe` and the attacker page are on different origins, the attacking page should not be able to learn any information about the `iframe`'s content. However, since the rendering time of the SVG filter is visible to the attacker page, and the rendering time is dependent on the `iframe` content, the attacking page is able to violate this policy and learn pixel information.

2.3 New floating point timing observations

Andryso et al. [8] presented a number of timing variations in floating point computation based on subnormal and special value arguments. We expand this category to note that *any* value with a zero significand or exponent exhibits different timing behavior on most Intel CPUs.

Table 2.3 shows a summary of our findings for our primary test platform running an Intel i5-4460 CPU. Unsurprisingly, double precision floating point numbers show more types of, and larger amounts of, variation than single precision floats.

Figures 2.3, 2.4, 2.5, and 2.6 are crosstables showing average cycle counts for division and multiplication on double and single precision floats on the Intel i5-4460. We refer to the

Table 2.3. Observed sources of timing differences under different settings on an Intel i5-4460. – : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

Operation	Default	FTZ & DAZ	-ffast-math
Single Precision			
Add/Sub	–	–	–
Mul	S	–	–
Div	S	–	–
Sqrt	M	Z	–
Double Precision			
Add/Sub	–	–	–
Mul	S	–	–
Div	M	Z	Z
Sqrt	M	Z	Z

type of operation (add, subtract, divide, etc) as the operation, and the specific combination of operands and operation as the computation. Cells highlighted in blue indicate computations that averaged 1 cycle higher than the mode across all computations for that operation. Cells in orange indicate the same for 1 cycle less than the mode. Bold face indicates a computation that had a standard deviation of > 1 cycle (none of the tests on the Intel i5-4460 had standard deviations above 1 cycle). All other crosstables in this paper follow this format unless otherwise noted.

We run each computation (operation and argument pair) in a tight loop for 40,000,000 iterations, take the total number of CPU cycles during the execution, remove loop overheads, and find the average cycles per computation. This process is repeated for each operation and argument pair and stored. Finally, we run the entire testing apparatus 10 times and store all the results. Thus, we execute each computation 400,000,000 times split into 10 distinct samples. This apparatus measures the steady-state execution time of each computation.

The entirety of our data across multiple generations of Intel and AMD CPUs, as well as tools and instructions for generating this data, are available at https://cseweb.ucsd.edu/~dkohlbre/floats.

It is important to note that the Andrysco et al. [8] focused on the performance difference

between subnormal and normal operands, while we observe that there are additional classes of values worth examining. The specific differences on powers-of-two are more difficult to detect with a naive analysis as they cause a slight speedup when compared to the massive slowdown of subnormals.

	0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	6.57	6.57	6.60	6.58	6.59	6.57	6.59	6.58	6.59
1.0	6.59	6.59	6.59	6.57	6.56	130.90	130.85	6.58	6.57
1e10	6.57	6.59	6.58	6.59	6.56	130.90	130.91	6.58	6.58
1e+30	6.59	6.56	6.58	6.59	6.57	130.90	130.91	6.59	6.58
1e-30	6.57	6.59	6.59	6.57	6.59	6.59	6.58	6.58	6.57
1e-41	6.56	130.90	130.89	130.87	6.56	6.57	6.57	130.96	130.90
1e-42	6.59	130.89	130.88	130.90	6.57	6.58	6.57	130.85	130.89
256	6.58	6.58	6.55	6.57	6.58	130.92	130.88	6.57	6.56
257	6.56	6.55	6.59	6.58	6.57	130.89	130.88	6.57	6.58

Figure 2.3. Multiplication timing for single precision floats on Intel i5-4460

	Divisor								
Dividend	0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	6.55	6.50	6.58	6.57	6.54	6.57	6.56	6.58	6.59
1.0	6.58	6.58	6.58	6.57	6.57	152.59	152.57	6.59	6.60
1e10	6.58	6.58	6.58	6.59	6.58	152.57	152.56	6.56	6.58
1e+30	6.57	6.57	6.59	6.57	6.56	152.59	152.51	6.58	6.60
1e-30	6.57	6.57	155.37	6.57	6.58	152.54	152.59	6.57	6.54
1e-41	6.58	149.75	6.57	6.56	152.56	152.57	152.59	149.72	152.55
1e-42	6.59	149.72	6.56	6.56	152.60	152.56	152.49	149.74	152.54
256	6.58	6.60	6.56	6.60	6.55	152.53	152.70	6.58	6.58
257	6.58	6.58	6.57	6.57	6.54	152.59	152.51	6.57	6.55

Figure 2.4. Division timing for single precision floats on Intel i5-4460

2.4 Fixed point defenses in Firefox

In version 28 Firefox switched to a new set of SVG filter implementations that caused the attack presented by Andrysco et al. [8] to stop functioning. Many of these implementations no longer used floating point math, instead using their own fixed point arithmetic.

	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.59	6.56	6.59	6.58	6.58	6.57	6.58	6.59	6.57
1.0	6.57	6.59	6.55	6.57	6.57	6.56	6.56	6.56	130.89
1e10	6.55	6.55	6.56	6.58	6.56	6.56	6.56	6.57	130.95
1e+200	6.55	6.57	6.56	6.58	6.59	6.53	6.55	6.58	130.92
1e-300	6.51	6.57	6.56	6.59	6.57	6.57	6.55	6.58	6.54
1e-42	6.55	6.57	6.55	6.57	6.55	6.58	6.58	6.58	6.55
256	6.58	6.53	6.56	6.54	6.56	6.56	6.58	6.57	130.94
257	6.59	6.57	6.60	6.56	6.58	6.56	6.57	6.59	130.90
1e-320	6.59	130.90	130.92	130.94	6.59	6.58	130.95	130.91	6.56

Figure 2.5. Multiplication timing for double precision floats on Intel i5-4460

	Divisor								
Dividend	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.56	6.59	6.58	6.55	6.57	6.58	6.57	6.57	6.59
1.0	6.58	6.58	12.19	12.17	12.22	12.24	6.57	12.24	165.76
1e10	6.58	6.55	12.25	12.20	12.23	12.25	6.57	12.22	165.81
1e+200	6.60	6.60	12.25	12.20	12.22	12.22	6.58	12.24	165.79
1e-300	6.59	6.57	175.22	12.24	12.17	12.22	6.52	12.23	165.83
1e-42	6.60	6.53	12.23	12.22	12.21	12.24	6.58	12.21	165.79
256	6.57	6.55	12.24	12.20	12.20	12.20	6.53	12.22	165.79
257	6.55	6.58	12.24	12.22	12.24	12.23	6.56	12.21	165.80
1e-320	6.56	150.73	165.79	6.59	165.78	165.76	150.66	165.80	165.78

Figure 2.6. Division timing for double precision floats on Intel i5-4460

As the `feConvolveMatrix` implementation now consists entirely of integer operations, we cannot use floating point timing side channels to exploit it. We instead examined a number of the other SVG filter implementations and found that several had not yet been ported to the new fixed point implementation, such as the lighting filters.

2.4.1 Fixed point implementation

The fixed point implementation used in Firefox SVG filters is a simple 32-bit format with no Not-a-Number, Infinity, or other special case handling. Since they make use of the standard add/subtract/multiply operations for 32-bit integers, we know of no timing side channels based on operands for this implementation. Integer division is known to be timing variable based on

the upper 32-bits of 64-bit operands, but none of the filters can generate intermediate values requiring the upper 32-bits. Thus, none of the filters we examined using fixed point had any instruction data timing based side channels. Handling the full range of floating point functionality in a fixed point and constant time way is expensive and complex, as seen in [8].

A side effect of a simple implementation is that it cannot handle more complex operations that could induce NaNs or infinities and must process them.

2.4.2 Lighting filter attack

Our Firefox SVG timing attack makes use of the `feSpecularLighting` lighting model with an `fePointLight`. This particular filter in this configuration is not ported to fixed point, and performs a scaling operation over the input alpha channel. The `surfaceScale` property in `feSpecularLighting` controls this scaling operation and can be set to an arbitrary floating point value when creating the filter. With this tool, we perform the following attack similar to the one in section 2.2.2. We need only to modify step 4 as seen below to enable the use of the new lighting filter attack.

1. Steps 1-3 are the same as section 2.2.2.

   4.1. Apply an `feColorMatrix` to the pixel multiplier `<div>` that sets the alpha channel based entirely on the input color values. This sets the alpha channel to 1 for a black pixel input, and 0 for a white pixel input.

   4.2. Apply the timing variable `feSpecularLighting` filter with subnormal `surfaceScale` and an attached `fePointLight` as the timing vulnerable filter.

5. Steps 5 and 6 are the same as section 2.2.2.

In this case, we differentiate between n^2 multiplications of *subnormal* $\times$ 0 (black) vs *subnormal* $\times$ 1 (white) where n is the width/height of the copied pixel `<div>`. Since our measurements show a difference of 7 cycles vs 130 cycles for each multiplication (see Figure

2.3), we can easily detect this difference once we scale n enough that the faster white pixel case takes longer than 16ms (circa $n = 200$) in our tests. We need to cross this 16ms threshold as frames take a minimum of 16ms to render (60fps) on our test systems.

In our tests on an Intel i5-4460 with Firefox 49+ we were able to consistently obtain $>$ 99% accuracy (on black and white images) at an average of 17ms per pixel. This is approximately as fast as an attack using this method can operate, since Firefox animates at a capped 60fps on all our test systems.

We notified Mozilla of this attack and they are working on a comprehensive solution. Firefox has patched the `surfaceScale` based attack on the `feSpecularLighting` filter in Firefox 52 and assigned the attack CVE-2017-5407.

2.5 Safari

At the time of writing this paper, Safari has not implemented any defensive mechanisms that hamper the SVG timing attack presented in [8]. Thus, with a rework of the attack framework, we are able to modify the attack presented in Andrysco et al against the `feConvolveMatrix` filter for Firefox 25 to work against current Safari.

Webkit (Safari) uses its own SVG filter implementations not used in other browsers. None of the SVG filters had GPU support at the time of this paper, but some CSS transforms could be GPU accelerated.

The Webkit `feConvolveMatrix` filter is implemented in the obvious way; multiply each kernel sized pixel region against the kernel element-by-element, sum, and divide the result by the divisor. We can therefore cause operations with $0\times$subnormal or normal$\times$subnormal depending on the target pixel. Since as we have seen these can a $0\times$subnormal can be $21\times$ faster than a subnormal times a normal, we can easily detect the difference between executing over a black pixel or a white pixel.

We disclosed the vulnerability to Apple, and they have released a patch for Safari that

```
<div id="pixel" style="width:500px;height:500px;overflow:
   hidden">
  <div id="scroll" style="width:1px; height:1px; overflow
     :hidden; transform:scale(600.0);margin:249px auto">
  <iframe id="frame" position="absolute" frameborder="0"
     scrolling="no" src="TARGET_URL"/>
  </div>
</div>
```

Figure 2.7. HTML and style design for the pixel multiplying structure used in our attacks on Safari and Chrome

completely removes cross-origin SVG filter support. We believe this to be the most comprehensive solution. The vulnerability was assigned CVE-2017-7006.

2.5.1 Tweaks for Safari

Two challenges arise from this switch; the `-moz-element` feature is not present in Safari, and there is a 1 frame delay in processing the SVG filter application.

Rather than use `-moz-element` to duplicate pixels as in [8], we instead use the `-transform : scale(x)` CSS transform. This corresponds to modifying step 3 in section 2.2.2. Due to the way the scale operation works, the scaled DOM element must first be centered in a parent element, and then the parent element can have SVG filters applied. The ordering of transforms, elements, and filters to cause the desired effect is brittle, and we detail our exact setup in figure 2.7. The "pixel" element is the element that has the vulnerable filter applied to it during the attack. The "scroll" element selects a pixel to extract (by setting the `scrollTop` and `scrollLeft` properties) as well as multiplying the target pixel. Finally the "frame" element is the iframe containing the victim page.

We address the 1-frame delay by simply measuring the total time it takes to render 2 frames after the SVG filter is applied to the element. This is accomplished by chaining 2 `requestAnimationFrame` callbacks. This consistently allowed us to measure the render time of the target SVG filter on our test machine. However, this does limit the maximum rate of

pixel extraction since we only get at best a pixel every 33ms.

2.6 DAZ/FTZ FPU flag defenses in Chrome

Google Chrome implements CSS and SVG filter support through the Skia [1] graphics library. As of July of 2016, when executing Skia filters on the CPU, Chrome enables an FPU control flag based countermeasure to timing attacks. Specifically, Chrome enables the Flush-to-Zero (FTZ) and Denormals-are-Zero (DAZ) flags.

These flags are two of the many FPU control flags that can be set. Flags determine options such as when to set a floating point exception, what rounding options to use, and how to handle subnormals. The FTZ flag indicates to the FPU that whenever it would produce a subnormal as the result of a calculation, it instead produces a zero. The DAZ flag indicates to the FPU that any subnormal operand should be treated as if it were zero in the computation. Generally these flags are enabled together as a performance optimization to avoid any use or generation of subnormal values. However, these flags break strict IEEE-754 compatibility and so some compilers do not enable them without specific optimization flags. In the case of Chrome, FTZ and DAZ are enabled and disabled manually in the Skia rendering path.

	Divisor								
Dividend	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.58	6.59	6.58	6.55	6.59	6.54	6.54	6.56	6.56
1.0	6.55	6.55	12.23	12.19	12.22	12.22	6.56	12.25	6.56
1e10	6.58	6.59	12.22	12.22	12.21	12.21	6.59	12.23	6.59
1e+200	6.57	6.59	12.22	12.20	12.17	12.21	6.58	12.17	6.57
1e-300	6.59	6.57	12.18	12.23	12.24	12.22	6.59	12.24	6.57
1e-42	6.58	6.56	12.21	12.25	12.23	12.18	6.56	12.21	6.58
256	6.57	6.60	12.20	12.22	12.24	12.24	6.57	12.23	6.54
257	6.57	6.58	12.22	12.23	12.25	12.20	6.57	12.23	6.58
1e-320	6.57	6.58	6.60	6.51	6.59	6.57	6.58	6.55	6.58

Figure 2.8. Division timing for double precision floats on Intel i5-4460+FTZ/DAZ

[1] https://skia.org/

2.6.1 Attacking Chrome

We present a cross-origin pixel stealing attack for Google Chrome using the `feConvolveMatrix` filter. As in our previous attacks, we observe the timing differences between white and black pixels rendered with a specific convolution matrix. This attack works without any changes on all major platforms for Chrome that support GPU acceleration. We have tested it on Windows 10 (Intel i7-6700k), Ubuntu Linux 16.10 (Intel i5-4460), OSX 10.11.6 (Intel i7-3667U Macbook Air), and a Chromebook Pixel LS ChromeOS 55.0.2883.105 (i7-5500U) on versions of Chrome from 54-56. The attack is very similar to the one detailed in section 2.2.2 and Figure 2.2.

Unlike Firefox, we cannot trivially supply subnormal value like "1e-41", as the Skia SVG float parsing code treats them as 0s. The float parsing in Skia attempts to avoid introducing subnormal values by disallowing exponents ≤ -37. Thus we use the value $0.0000001e-35$ or simply the fully written out form, which is correctly parsed into a subnormal value. Since the FTZ and DAZ flags are set only on entering the Skia rendering code, the parsing is not subject to these flags and we can always successfully generate subnormals at parse time.

The largest obstacle we bypass is the use of the FTZ and DAZ control flags. These flags reduce the precision and representable space of floats, but prevent any performance impact caused by subnormals for these filters in our experiments. As shown in section 2.3 even with these flags enabled the `div` and `sqrt` operations still have timing variation. Unfortunately none of the current SVG filter implementations we examined have tight division loops over doubles, or tight square root operations over floats. Thus, our attack must circumvent the use of the FTZ and DAZ flags altogether.

Chrome enables the FTZ and DAZ control flags whenever a filter is set to run on the CPU, which disallows our Firefox or Safari attacks from applying directly to Chrome. However, we found that the FTZ and DAZ flags are not set when a filter is going to execute on the GPU. This would normally only be useful for a GPU-based attack but we can force the `feConvolve-`

`Matrix` filter to abort from GPU acceleration at the last possible moment and fall back to the CPU implementation by having a kernel matrix over the maximum supported GPU size of 36 elements. Chrome does not enable the FTZ and DAZ flags when it executes this fallback, allowing our timing attack to use subnormal values.

We force the target `<div>` to start on the GPU rendering path by applying a CSS `transform:rotateY()` to it. This is a well known trick for causing future animations and filters to be performed on the GPU, and it works consistently. Without this, the `feConvolveMatrix` GPU implementation would never fire, as it will not choose the GPU over the CPU on its own. It is only because of our ability to force CPU fallback with the FTZ and DAZ flags disabled that allows our CPU Chrome attack to function.

Note that even if FTZ/DAZ are enabled in all cases there are still scenerios that show timing variation as seen in Figure 2.8 and Table 2.3. Chrome's Skia configuration currently uses single precision floats, and thus only need avoid sqrt operations as far as we know. However, any use of double precision floats will additionally require avoidance of division. We did not observe any currently vulnerable uses of single precision sqrt, or of double precision floating point operations in the Skia codebase.

We notified Google of this attack and a patch enabling FTZ/DAZ even on GPU bail is now available on Chrome. Google assigned this issue CVE-2017-5107.

2.6.2 Frame timing on Chrome

An additional obstacle to our Chrome attack was obtaining accurate frame render times. Unlike on Firefox or Safari, adding a filter to a `<div>`'s style and then calling `getAnimationFrame` is insufficient to be sure that the time until the callback occurs will accurately represent the rendering time of the filter. In fact, the frame that the filter is actually rendered on differs by platform and is not consistent on Linux. We instead run algorithm 1 to get the approximate rendering time of a given frame. Since we only care about the relative rendering time between white and black pixels, the possibly extra time included doesn't matter as long as it is moder-

ately consistent. This technique allowed our attack to operate on all tested platforms without modification.

```
Result: Duration of SVG filter rendering
total_duration = 0ms;
long_frame_seen = False;
while true do
    /* Wait for next frame                                    */
    requestAnimationFrame;
    if duration > 40ms then
        /* Long frame probably containing the SVG
           rendering occurred                                 */
        long_frame_seen = True;
        total_duration += duration;
    else
        if long_frame_seen then
            /* A short frame after a long frame              */
            return total_duration;
        end
    end
    total_duration += duration;
end
```

Algorithm 1: How to measure SVG filter rendering times in Chrome

2.7 Revisiting the effectiveness of Escort

Escort [70] proposes defenses against multiple types of timing side channels, notably a defense using SIMD vector operations to protect against the floating point attack presented by Andrysco et al in [8].

Single Instruction, Multiple Data (SIMD) instructions are an extension to the x86_64 ISA designed to improve the performance of vector operations. These instructions allow 1-4 independent computations of the same operation (divide, add, subtract, etc) to be performed at once using large registers. By placing the first set of operands in the top half of the register, and the second set of operands in the bottom half, multiple computations can be easily performed with a single opcode. Intel does not provide significant detail about the execution of these

instructions and does not provide guarantees about their performance behavior.

2.7.1 Escort overview

Escort performs several transforms during compilation designed to remove timing side channels. First, they modify 'elementary operations' (floating point math operations for the purpose of this paper). Second, they perform a number of basic block linearizations, array access changes, and branch removals to transform the control flow of the program to constant time and minimize side effects.

We do not evaluate the efficacy of the higher level control flow transforms and instead evaluate only the elementary operations.

Escort's tool is to construct a set of dummy operands (the *escort*) that are computed at the same time as the secret operands to obscure the running time of the secret operands. Escort places the dummy arguments in one lane of the SIMD instruction, and the sensitive arguments in another lane. Since the instruction only retires when the full set of computations are complete, the running time of the entire operation is hypothesized to be dependent only on the slowest operation. This is true if and only if the different lanes are computed in parallel. To obscure the running time of the sensitive operands, Escort places two subnormal arguments in the dummy lane of all modified operations under the assumption that this will exercise the slowest path through the hardware.

Escort will replace most floating point operations it encounters. However, if it can prove (using the Z3 SMT solver [24]) that the operation will never have subnormal values as operands it declines to replace the operation. This means that if a function filters out subnormals before performing computation, the computation will be done with standard scalar floating point operations and not vector operations. This results in significant performance gains when applicable, as the scalar operations can be two orders of magnitude faster than the subnormal vector operations. The replacement operations consist of hand-coded assembly contained in a library; `libdrag`.

Table 2.4. Timing differences observed for `libdrag` vs default operations on an Intel i5-4460. – : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

Operation	Default	`libdrag`
Single Precision		
Add/Sub	–	–
Mul	S	–
Div	S	Z
Sqrt	M	Z
Double Precision		
Add/Sub	–	–
Mul	S	–
Div	M	Z
Sqrt	M	Z

	Divisor								
Dividend	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	186.46	186.48	186.50	186.44	186.42	186.49	186.50	186.48	186.51
1.0	186.45	186.48	195.93	195.94	195.93	195.86	186.48	195.87	186.48
1e10	186.51	186.49	195.92	195.90	195.92	195.87	186.47	195.86	186.46
1e+200	186.50	186.50	195.90	195.94	195.89	195.91	186.46	195.90	186.50
1e-300	186.48	186.44	195.91	195.88	195.93	195.92	186.53	195.95	186.44
1e-42	186.44	186.51	195.92	195.94	195.87	195.89	186.51	195.93	186.47
256	186.49	186.49	195.91	195.91	195.87	195.89	186.45	195.91	186.44
257	186.46	186.47	195.96	195.92	195.92	195.96	186.49	195.98	186.45
1e-320	186.49	186.49	186.43	186.48	186.49	186.49	186.50	186.52	186.46

Figure 2.9. Division timing for double precision floats on Intel i5-4460+Escort

However, operations that do not receive subnormals can still exhibit timing differences. As seen in Figure 2.6 and summarized in Table 2.3 timing differences arise on value types that can commonly occur (0, powers of 2, etc). While significantly less obvious than the impact of subnormals, these still constitute a potential timing side channel. `libdrag` can easily fix this, at serious performance cost, by enabling the floating point replacements for all floating point operations with no exceptions.

To determine if Escort closes floating point timing side channel when enabled, we measured the timing behavior of Escort's `libdrag` floating point operations, as well as the

Table 2.5. Timing differences observed for `libdrag` vs default operations on an AMD Phenom II X2 550.
– : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

Operation	Default	`libdrag`
Single Precision		
Add/Sub	S	S
Mul	S	–
Div	S	–
Sqrt	S	–
Double Precision		
Add/Sub	S	S
Mul	S	–
Div	S	–
Sqrt	S	–

end-to-end runtime of toy programs compiled under Escort.

2.7.2 `libdrag` micro-benchmarks

For the micro-benchmarking of the `libdrag` functions we use a simple tool we developed for running timing tests of library functions based on Intel's recommendations for instruction timing. This is the same tool we used to produce measurements for section 2.3.

We benchmarked each of `libdrag`'s functions against a range of valid numbers on several different CPUs. We do not present results for Not-a-Number (NaN) or infinities.

Results on Intel i5-4460

Our results for the Intel i5-4460 CPU roughly correspond to the variations presented in [70] (which tested on an Intel i7-2600) for `libdrag`. We do not observe any measurable timing variation in any add, multiply, or subtract operations for single or double precision floating point. We do observe notable timing differences based on argument values for single and double precision division and square-root operations. The cross table results for double precision division are shown in Figure 2.9. Table 2.4 summarizes the timing variations we observed.

Dividend	Divisor							
	0.0	1.0	1.0e-10	1.0e-323	1.0e-43	1.0e100	256	257
	Runtime (Seconds)							
0.0	10.09	10.08	10.08	10.08	10.08	10.08	10.08	10.10
1.0	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e-10	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e-323	10.08	10.08	10.08	10.08	10.08	10.08	10.08	10.08
1.0e-43	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e100	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
256	10.08	10.08	10.55	10.08	10.57	10.55	10.08	10.57
257	10.09	10.08	10.55	10.08	10.57	10.55	10.08	10.55

Figure 2.10. Division timing for double precision floats on Intel i5-4460 macro-test

For division, it appears that the numerator has no impact on the running time of the computation. The denominator shows variation based on if the significand or exponent is all zero bits. When either portion is zero in the denominator computations run consistently faster in both single and double precision floating point. Differences observed range from 2% to 5% in contrast to the 2500% differences observed in section 2.3.

Square root shows a similar behavior, where if either the significand or exponent is all 0 bits the computation runs consistently faster. This matches the behavior seen for many operations in scalar computations. (See Table 2.3)

An interesting outcome of this behavior is that subnormal values cause a *speedup* under `libdrag` rather than the slowdown observed under scalar operations.

We speculate that this is the result of fast paths in the microcode handling for vector operations. Using performance counters we determined that all vector operations containing a subnormal value execute microcode rather than hardwired logic on the FPU hardware. As all values with a zero significand or exponent experienced a speedup, we believe that the division and square root microcode handles these portions separately with a shortcut in the case of zero. Intel did not release any details on the cause of these timing effects when asked.

AMD Phenom II X2 550

Table 2.5 summarizes our results on the AMD Phenom II X2 550. As with the Intel i5-4460 we observe timing variation in the AMD Phenom II X2 550. However, the variation is now confined to addition and subtraction with subnormal values. By examining the cycle times for each operation in the default and `libdrag` case we found that the total cycle time for an escorted add or subtract is approximately equal to the sum of the cycle counts for a subnormal,subnormal operation and the test case. Thus, we believe that the AMD Phenom II X2 550 is performing each operation sequentially and with the same hardware or microcode as scalar operations for addition and subtraction.

2.7.3 Escort compiled toy programs

For end-to-end tests we wrote toy programs that perform a specified floating point operation an arbitrary number of times, and compiled them under Escort and `gcc`. We then use the Linux `time` utility to measure runtimes of the entire program. We designed the test setup such that each run of the test program performed the same value parsing and setup steps regardless of the test values, with only the values entering the computation differing between runs. We ran the target computation 160,000,000 times per execution, and ran each test 10 times. We see the same effects as in our microbenchmarks. Figure 2.10 shows the crosstable for these results. Note that cells are colorized if they differ by 2% rather than 1 cycle.

2.7.4 `libdrag` modified Firefox

We modified a build of Firefox 25 in consultation with Rane et al [70] to match the version they tested. Since multiply no longer shows any timing variation in `libdrag` we are restricted to observing a potential $\leq 2\%$ difference in only the divide, which occurs once per pixel regardless of the kernel. Additionally, since the denominator is the portion controlled by the attacker and the secret value is the numerator, we are not able to update the pixel stealing attack for the modified Firefox 25.

The modifications to Firefox 25 were confined to hand made changes to the `feConvolveMatrix` implementation targeted in [8]. We did not test other SVG filters for vulnerability under the Escort/`libdrag` modifications.

Given the observed timing variations in the AMD Phenom II X2 550 in section 2.7.2 we believe that multiple SVG filters would be timing side channel vulnerable under Escort on that CPU.

2.7.5 Escort summary

Unfortunately our benchmarks consistently demonstrated a small but detectable timing difference for `libdrag`'s vector operations based on operand values. For our test Intel CPUs it appears that `div` and `mul` exhibit timing differences under Escort. For our AMD CPUs we observed variation only for `add`/`sub`. Additionally, these differences are no more than 5% as compared to the 500% or more differences observed in scalar operations. We have made Rane, Lin and Tiwari aware of these findings.

The 'escort' mechanism can only serve as an effective defense if vector operations are computed in parallel. In all CPUs we tested the most likely explanation for the observed timing difference is that vector operations are executed serially when in microcode. As mentioned in section 2.7.2 we know that any vector operation including a subnormal argument is executed in microcode, and all evidence supports the microcode executing vector operations serially. Thus, absent substantial architectural changes, we do not believe that the 'escort' vector mechanism can close all floating point data timing channels.

2.8 GPU floating point performanace

In this section we discuss the results of GPU floating point benchmarks, and the use of GPU acceleration in SVG filters for Google Chrome.

Dividend	Divisor 0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	5.17	5.85	5.85	5.85	5.85	5.89	5.89	5.85	5.85
1.0	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
1e10	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e+30	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e-30	6.19	2.59	7.82	6.51	2.59	8.40	8.40	2.59	2.59
1e-41	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
1e-42	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
256	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
257	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59

Figure 2.11. Division timing for single precision floats on Nvidia GeForce GT 430

2.8.1 Browser GPU support

All major browsers make use of GPU hardware acceleration to improve performance for various applications. However, only two currently make use of GPUs for SVG and CSS transforms; Safari and Chrome. Currently, Safari only supports a subset of CSS transformations on the GPU, and none of the SVG transforms. Chrome supports a subset of the CSS and SVG filters on the GPU. Firefox intends to port filters to the GPU, but there is currently no support.

2.8.2 Performance

We performed a series of CUDA benchmarks on an Nvidia GeForce GT 430 to determine the impact of subnormal values on computation time. The results for division are shown in Figure 2.11. All other results (add, sub, mul) were constant time regardless of the inputs..

As Figure 2.11 shows, subnormals induce significant slowdowns on divsion operations for single precision floats. Unfortunately, no SVG filters implemented in Chrome on the GPU perform tight division loops. Thus, extracting timing differences from the occational division they do perform is extremely difficult.

If a filter were found to perform tight division loops, or a GPU that has timing variation on non-division operations were found, the same attacks as in previous sections could be ported

to the GPU accelerated filters.

We believe that even without a specific attack, the demonstration of timing variation based on operand values in GPUs should invalidate "move to the GPU" as a defensive strategy.

2.9 Related work

Felten and Schneider were the first to mount timing side-channel attacks against browsers. They observed that resources already present in the browser's cache are loaded faster than ones that must be requested from a server, and that this can be used by malicious JavaScript to learn what pages a user has visited [30]. Felten and Schneider's history sniffing attack was later refined by Zalewski [89]. Because many sites load resources specific to a user's approximate geographic location, cache timing can reveal the user's location, as shown by Jia et al. [46].

JavaScript can also ask the browser to make a cross-origin request and then learn (via callback) how long the response took to arrive and be processed. Timing channels can be introduced by the code that runs on the server to generate the response; by the time it takes the response to be transmitted over the network, which will depend on how many bytes it contains; or by the browser code that attempts to parse the response. These cross-site timing attacks were introduced by Bortz, Boneh, and Nandy [17], who showed they could be used to learn the number of items in a user's shopping cart. Evans [29] and, later, Gelernter and Herzberg [32], showed they could be used to confirm the presence of a specific string in a user's search history or webmail mailbox. Van Goethem, Joosen, and Nikiforakis [83] observed that callbacks introduced to support HTML5 features allow attackers to time individual stages in the browser's response-processing pipeline, thereby learning response size more reliably than with previous approaches.

The interaction of new browser features — TypedArrays, which translate JavaScript variable references to memory accesses more predictably than general arrays, and nanosecond-resolution clocks — allow attackers to learn whether specific lines have been evicted from the

processor's last-level cache. Yossi Oren first showed that such microarchitectural timing channels can be mounted from JavaScript [67], and used them to learn gross system activity. Recently, Gras et al. [34] extended Oren's techniques to learn where pages are mapped in the browser's virtual memory, defeating address-space layout randomization. In response, browsers rounded down the clocks provided to JavaScript to 5 μs granularity. Kohlbrenner and Shacham [53] proposed a browser architecture that degrades the clocks available to JavaScript in a more principled way, drawing on ideas from the "fuzzy time" mitigation [45] in the VAX VMM Security Kernel [49].

Browsers allow Web pages to apply SVG filters to elements including cross-origin iframes. If filter processing time varies with the underlying pixel values, those pixel values will leak. Paul Stone [81] and, independently, Kotcher et al. [55], showed that such pixel-stealing attacks are feasible; the filters they exploited had pixel-dependent branches. Andrysco et al. [8] showed that pixel-stealing was feasible even when the filter executed the same instruction trace regardless of pixel values, provided those instructions exhibit data-dependent timing behavior, as floating-point instructions do. Rane, Lin, and Tiwari [70] proposed program transformation that allow the processor floating-point unit to be used while eliminating data-dependent instruction timing, in the hope of defeating Andrysco et al.'s attacks.

2.10 Conclusions and future work

We have extensively benchmarked floating point performance on a range of CPUs under scalar operations, FTZ/DAZ FPU flags, `-ffast-math` compiler options, and Rane, Lin, and Tiwari's Escort. We identified operand-dependent timing differences on all tested platforms and in all configurations; many of the timing differences we identified were overlooked in previous work.

In the case of Escort, our data strongly suggests that processors execute SIMD operations on subnormal values sequentially, not in parallel. If this is true, a redesign of the vector processing

unit would be required to make Escort effective at closing all floating-point timing channels.

We have revisited browser implementations of SVG filters, and found (and responsibly disclosed) exploitable timing variations in the latest versions of Chrome, Firefox, and Safari.

Finally, we have shown that modern GPUs exhibit slowdowns in processing subnormal values, meaning that the problem extends beyond x86 processors. We are currently evaluating whether these slowdowns allow pixel stealing using SVG filters implemented on the GPU.

We have uncovered enough variation in timing across Intel and AMD microarchitectural revisions that we believe that comprehensive measurement on many different processor families — in particular, ARM — will be valuable. For the specific processors we studied, we believe we are in a position to identify specific flags, specific operations, and specific operand sizes that run in constant time. Perhaps the best one can hope for is an architecture-aware library that could ensure no timing variable floating point operations occur while preserving as much of the IEEE-754 standard as possible.

Tools, proof-of-concept attacks, and additional benchmark data are available at https://cseweb.ucsd.edu/~dkohlbre/floats.

We close with broader lessons from our work.

For software developers:

We believe that floating point operations as implemented by CPUs today are simply too unpredictable to be used in a timing-security sensitive context. Only defensive measures that completely remove either SSE floating point operations (fixed-point implementations) or remove the sensitive nature of the computation are completely effective. Software that operates on sensitive, non-integer values should use fixed-point math, for example by including Andrysco et al.'s `libfixedtimefixedpoint`, which Almeida et al. recently proved runs in constant time [7].

For browser vendors:

Some browser vendors have expended substantial effort in redesigning their SVG filter code in the wake of the Andrysco et al. attacks. Even so, we were able to find (different) exploitable floating-point timing differences in Chrome, Firefox, and Safari. We believe that the attack surface is simply too large; as new filters and features are added additional timing channels will inevitably open. We recommend that browser vendors follow Safari's lead in disallowing cross-origin SVG filters and other computation over cross-origin pixel data in the absence of Cross-Origin Resource Sharing (CORS) authorization.

It is important that browser vendors also consider patching individual timing side channels in SVG filters as they are found. Even with a origin policy that blocks the cross-origin pixel stealing, any timing side channel allows an attacking page to run a history sniffing attack. Thus, a comprehensive approach to SVG filters as a threat to user privacy combines disallowing cross-origin SVG filters and removes timing channels with constant time coding techniques.

For processor vendors:

Processor vendors have resisted calls to document which of their instructions run in constant time regardless of operands, even for operations as basic as integer multiplication. It is possible that floating point instructions are unusual not because they exhibit timing variation but because their operands have meaningful algebraic structure, allowing intelligent exploration of the search space for timing variations; even so, we identified timing variations that Andrysco et al. overlooked. How much code that is conjectured to be constant-time is in fact unsafe? Processor vendors should document possible timing variations in at least those instructions commonly used in crypto software.

Acknowledgements

We thank Eric Rescorla and Jet Villegas for sharing their insights about Firefox internals, and Philip Rogers, Joel Weinberger, and Stephen White for sharing their insights about Chrome

internals.

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

We thank Ashay Rane for his assistance in obtaining and testing the Escort compiler and `libdrag` library.

This material is based upon work supported by the National Science Foundation under Grants No. 1228967 and 1514435, and by a gift from Mozilla.

Chapter 2, in part, is a reprint of the material as it appears in USENIX Security 2017. Kohlbrenner, David; Shacham, Hovav, 2017. The dissertation author was the primary investigator and the primary author of this paper.

Chapter 3

Constant time fixed-point math

3.1 Introduction

In this chapter, we examine one approach to solving the problem of floating-point timing side-channels: substituting constant time fixed-point math in place of floating-point.

We design and evaluate a new library, `libfixedtimefixedpoint`, for non-integer math for which all operations run in constant time. We have manually 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.

In independent work, `libfixedtimefixedpoint` was shown to be constant time when compiled to LLVM bytecode[7]. While this does not imply that when compiled to x86_64 assembly `libftfp` remains constant time, we believe this result validates our structural approach.

3.2 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.

3.2.1 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 3.1), 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

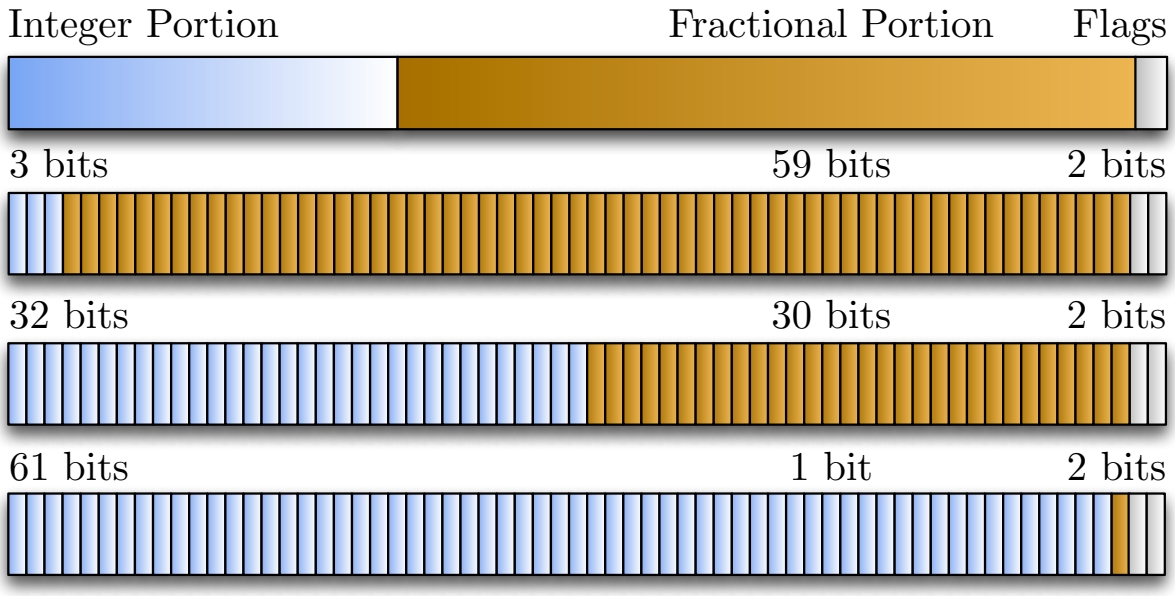


Figure 3.1. 3 possible internal layouts of a LibFTFP `fixed`. LibFTFP supports anywhere between 1 and 61 fractional bits, chosen at library compilation time.

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).

3.2.2 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.

3.2.3 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. [38] 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,

```
int64_t fix_to_int64(fixed op1) {
  return ({ uint8_t isinfpos = (((op1)&((fixed) 0x3)) == ((
     fixed) 0x2)); uint8_t isinfneg = (((op1)&((fixed) 0x3))
      == ((fixed) 0x3)); uint8_t isnan = (((op1)&((fixed) 0
     x3)) == ((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) & (1LL << (((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__
     ;}) & (92233720368547758O7LL)) | (({uint64_t SE_m__ =
     (1ull << ((1)-1)); (((uint64_t) (!!(isinfneg))) ^
     SE_m__) - SE_m__;}) & ((-92233720368547758O7LL -1))) |
     (({uint64_t SE_m__ = (1ull << ((1)-1)); (((uint64_t)
     (!!(!ex))) ^ SE_m__) - SE_m__;}) & (result_nosign)));
     });
}
```

Figure 3.2. Conversion of a LibFTFP value to an `int64`, after the C pre-processor has been run.

rather simple example of the style of coding necessary to generate code that can even be argued to run in constant time. See Figure 3.2 for an example of our C code with macros fully expanded.

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.

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 3.3. Every x86 instruction used by LibFTFP.

The full list can be found in Figure 3.3.

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 3.1. 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.

3.2.4 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

Table 3.1. 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.

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

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.

Our custom version of Fuzz computes all of our database queries from Chapter 1.4.5,

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 `doubles`, 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.

Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in IEEE Security and Privacy (Oakland) 2015. Andrysco, Mark; Kohlbrenner, David; Mowery, Keaton; Jhala, Ranjit; Lerner, Sorin; Shacham, Hovav, 2015. The dissertation author was a primary investigator and a primary author of this paper.

Chapter 4

Trusted browsers for uncertain times

4.1 Introduction

Web browsers download and run JavaScript code from sites a user visits as well as third-party sites like ad networks, granting that code access to system resources through the DOM. Keeping that untrusted code from taking control of the user's system is the *confinement* problem. In addition, browsers must ensure that code running in one origin does not learn sensitive information about the user's interaction with another origin. This is the *compartmentalization* problem.

A failure of confinement can lead to a failure of compartmentalization. But JavaScript can also learn sensitive information without escaping from its sandbox, in particular by exploiting *timing side channels*. A timing channel is made possible when an attacker can compare a *modulated clock* — one in which ticks arrive faster or slower depending on a secret — to a *reference clock* — one in which ticks arrive at a consistent rate. For example, browsers allow web pages to apply SVG transformations to page elements, including cross-origin frames, via CSS. Paul Stone showed that a fast-path optimization in the `feMorphology` filter created a timing attack that allowed attackers to steal pixels or sniff a user's browsing history, using `Window.requestAnimationFrame()` as a modulated clock [81]. More recently, Oren et al. showed that, in the presence of a high-resolution reference clock like `performance.now`, attackers could use JavaScript TypedArrays to measure instantaneous load on the last-level

processor cache [67].

Browser vendors are aware of the danger that timing channels pose compartmentalization and have made efforts to address it.

First, they have attempted to eliminate modulated clocks by making any code that manipulates secret values run in constant time. In a hundred-message Bugzilla thread, for example, Mozilla engineers decided to address Stone's pixel-stealing work by rewriting the `feMorphology` filter implementation using constant-time comparisons.[1]

Second, they have attempted to reduce the resolution of reference clocks available to JavaScript code. In May, 2015, the Tor Browser developers reduced the resolution of the `performance.now` high-resolution timer to 100 ms as an anti-fingerprinting measure.[2] In late 2015, some major browsers (Chrome, Firefox) applied similar patches (see Figure 4.1), reducing timer resolution to 5 µs to defeat Oren et al.'s cache timing attack [67].

These efforts are unlikely to succeed, because they seriously underestimate the complexity of the problem.

First, eliminating every potential modulated clock would require an audit of the entire code base, an ambitious undertaking even for a much smaller, simpler system such as a microkernel [19]. Indeed, the Mozilla fix for `feMorphology` did not consider the possibility that floating-point instructions execute faster or slower depending on their inputs, allowing pixel-stealing attacks even in supposedly "constant-time" code [8].

Second, there are many ways by which JavaScript code might synthesize a reference clock besides naively querying `performance.now`. In this paper, we show that *clock-edge detection* allows JavaScript to increase the effective resolution of a degraded `performance.now` clock by two orders of magnitude. We also present and evaluate multiple, new *implicit clocks*: techniques by which JavaScript can time events without consulting an explicit clock like `performance.now` at all. For example, videos in an HTML5 `<video>` tag are decoded in a

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=711043
[2] https://trac.torproject.org/projects/tor/ticket/1517

separate thread. JavaScript can play a simple video that changes color with each frame and examine the current frame by rendering it to a canvas. This immediately gives an implicit clock with resolution 60 Hz, and the resolution can be improved using our techniques.

In short, timing channels pose a serious danger to compartmentalization in browsers; browser vendors are aware of the problem and are attempting to address it by eliminating or degrading clocks attackers would rely on, but their ad-hoc efforts are unlikely to succeed. Our thesis in this paper is that the problem of timing channels in modern browsers is analogous to the problem of timing channels in trusted operating systems and that ideas from the trusted systems literature can inform effective browser defenses. Indeed, our description of timing channels as the comparison of a reference clock and a modulated clock is due to Wray [87], and our fuzzy mitigation strategy technique is directly inspired by Hu [44] — both papers resulting from the VAX VMM Security Kernel project, which targeted an A1 rating [49].

In this paper, we show that "fuzzy time" ideas due to Hu [44] can be adapted to building trusted browsers. Fuzzy time degrades *all* clocks, whether implicit or explicit, and it reduces the bandwidth of all timing channels. We describe the properties needed in a trusted browser where all timing sources are completely mediated. Today's browsers tightly couple the JavaScript engine and the DOM and would need extensive redesign to completely mediate all timing sources. As a proof of feasibility, we present Fuzzyfox, a fork of the Firefox browser that works within the constraints of today's browser architecture to degrade timing sources using fuzzy time. Fuzzyfox demonstrates a principled clock fuzzing scheme that can be applied to both mainstream browsers and Tor Browser using the same mechanics. We evaluate the performance overhead and compatibility of Fuzzyfox, showing that all of its ideas are suitable for deployment in products like Tor Browser and a milder version are suitable for Firefox.

```
double PerformanceBase::clampTimeResolution(double
   timeSeconds)
{
    const double resolutionSeconds = 0.000005;
    return floor(timeSeconds / resolutionSeconds) *
       resolutionSeconds;
}
```

Figure 4.1. Google Chrome `performance.now` rounding code

4.2 Clock-edge attack

Web browser vendors have attempted to mitigate timing side channel attacks like [67] by rounding down the explicit clocks available to JavaScript to some grain g. For example, Google Chrome and Firefox have implemented a 5μs grain. Figure 4.1 shows the C++ code used for rounding a `performance.now` call in Google Chrome. Tor Browser makes a different privacy and performance tradeoff and has implemented an aggressive 100ms grain.

Unfortunately, rounding down does not the guarantee that an attacker cannot accurately measure timing differences smaller than g. We present the *clock-edge* technique for improving the granularity of time measurements in the context of JavaScript clocks. Experimentally, this technique results in an increase in resolution of at least two orders of magnitude to large grained clocks. This technique can be generalized to any pair of clocks: a *major* clock, which has a known large period, and a a *minor* clock, which has a short unknown period. The major clock is used to establish the period of the minor clock, and together they can time events with more accuracy than alone.

Consider the case of a page wishing to time some JavaScript function `attack()` with a granularity smaller than some known `performance.now` grain g. The major clock in this case is the degraded `performance.now`, and we use a tight incrementing `for` loop as the minor clock. Figures 4.2 and 4.3 show how a page might execute this technique and a visual representation of the process.

```
// Find minor ticks until major edge
function nextedge(){
  start = performance.now();
  stop = start;
  count = 0;

  while(start == stop){
    stop = performance.now();
    count++;
  }

  return [count,start,stop];
}

// run learning
nextedge();
[exp,pre,start] = nextedge();

// Run target function
attack();

// Find the next major edge
[remain,stop,post] = nextedge();

// Calculate the duration
duration = (stop-start)+((exp-remain)/exp)*grain;
```

Figure 4.2. Clock-edge fine-grained timing attack in JavaScript

The page first learns the average number of loop iterations (L_{exp}) between the major clock ticks C_{l1} and C_{l2}. After learning, the page then runs until a major clock edge is detected (C_{start}) and then executes `attack()`. When `attack()` returns at major clock time C_{stop}, the page runs the minor clock (for L_{remain} ticks) until the next major clock edge (C_{post}) is detected. The page then calculates the duration of `attack()` as $(C_{stop} - C_{start}) + g * (L_{exp} - L_{remain})/(L_{exp})$. In the case of g not remaining constant, we scale the L_{exp} by $(C_{post} - C_{stop})/(C_{l2} - C_{l1})$ and set $g = C_{post} - C_{stop}$.

Since $(L_{exp} - L_{remain})/(L_{exp})$ represents a fractional portion of g, the duration measure-

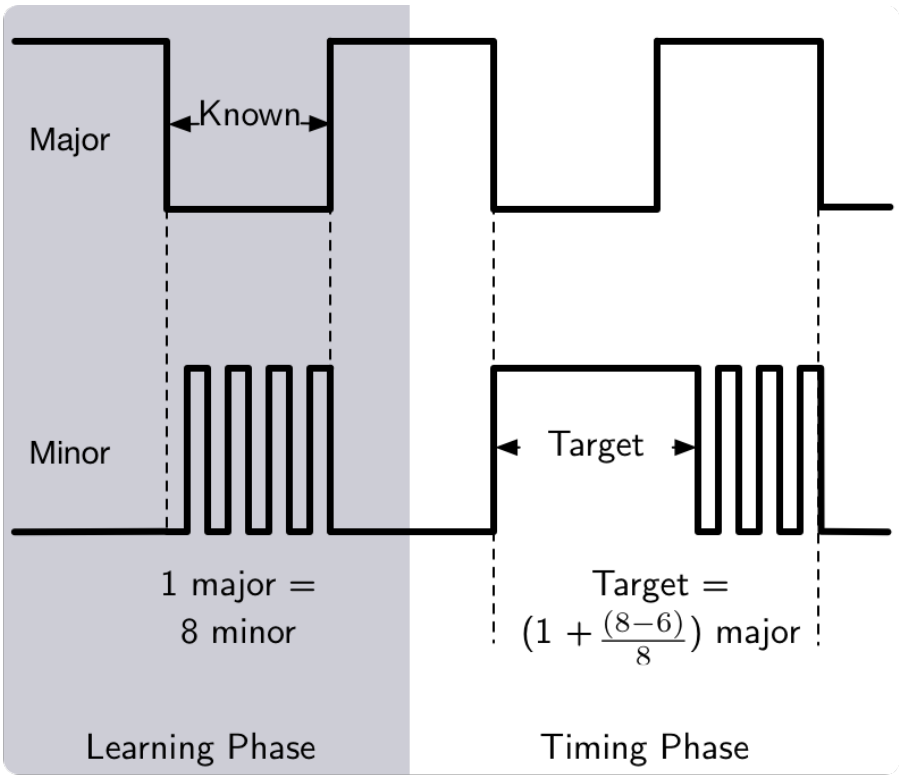


Figure 4.3. Clock-edge learning and timing

ment can plausibly obtain measurements as fine grained as g/L_{exp}. Thus, as long as the attacker has access to a suitable minor clock, the degradation of a major clock to g by rounding does not ensure an attacker cannot measure at a grain less than g.

Table 4.1 shows the results of applying the clock-edge technique on a degraded `performance.now` major clock on 4 different targets at different grains. The code in figure 4.2 is an abbreviated version of the testing code. Each duration column represents a different number of iterations in the `attack()` function, which is an empty `for` loop. The minor ticks column indicates the number of iterations the learning phase detected that each major tick takes. The "None" row indicates the runtime of `attack` with no rounding enabled, and other rows indicate the durations measured at different grain settings using the clock-edge technique. Measurements were performed with a modified build of Firefox that enabled setting arbitrary grains via JavaScript.

Table 4.1. Results for running the clock-edge fine-grained timing attack against various grain settings. Averages for 100 runs shown.

Grain(ms)	Minor	Measured Durations(ms)			
None	–	0.003	0.030	0.298	3.033
0.001	2	0.002	0.029	0.299	3.103
0.005	94	0.004	0.032	0.304	3.031
0.01	192	0.003	0.030	0.298	2.998
0.08	1649	0.003	0.030	0.303	3.009
0.1	1965	0.011	0.027	0.299	3.006
1	20470	0.053	0.038	0.296	3.010
10	193151	0.112	0.208	0.332	3.159
100	1928283	0.436	0.469	0.560	3.330
500	9647265	1.045	1.076	1.294	3.437

As Table 4.1 shows, the clock-edge attack recovers durations significantly smaller than the grain settings. Notably, grains in the millisecond and higher range still permit the differentiation of events lasting only tens of μs!

Simply rounding down the available explicit clocks only has a notable impact if the attacker is attempting to differentiate between events each lasting less than a microsecond, at which level the clock-edge attack often provides no additional resolution to the rounded clock.

4.3 Measuring time in browsers without explicit clocks

In this section, we demonstrate different methods an attacker can use measure the duration of events in JavaScript. An attacker wishing to mount a timing attack against a web browser is not restricted to the use of `performance.now` for timing measurements, this section will present a number of alternative methods available. Browser features that enable these measurements are *implicit clocks*. Depending on the how the target and the clock interact with the JavaScript runtime, we define them as *exiting* or *exitless*. We do not present an exhaustive list of implicit clocks. Rather, this section should be considered the tip of the iceberg for clock techniques in browsers.

4.3.1 Measurement targets

Recall that the adversary's goal in a timing attack is to measure the duration of some event and differentiate between two or more possible executions. We assume our adversary's goal is to measure the duration of some piece of JavaScript `target()` or to measure the time until some event `target` fires a callback. There are many potential targets, exemplified by two different timing attacks on web browsers. We categorize targets and attacks into exiting and exitless and describe a canonical example for each.

Exiting targets: privacy breaches with `requestAnimationFrame`

Previous work [8] [81] has shown several different ways to achieve history sniffing or cross frame pixel reading via timing the rendering of an SVG filter over secret data. Andrysco et al [8] demonstrate a timing attack on privacy that differentiates pixels based on how long rendering an SVG convolution filter takes. This timing requires that the attacking JavaScript know exactly when the SVG filter is applied to the target and when the SVG filter finishes rendering. This is accomplished by sampling a high resolution time stamp (`performance.now`) when applying the CSS style containing the filter and when a callback for `requestAnimationFrame` fires. In this case, JavaScript must exit to allow some other computation to occur and then receives a notification via a callback that the event has completed. We refer to this type of target as an *exiting* `target`, as it exits the JavaScript runtime before completion.

Exitless targets: cache timing attacks from JavaScript

Conversely, there are *exitless* `targets`, such as Oren et al's [67] cache timing attack. This attack does not need to exit JavaScript for the `target` to run, instead they need only perform some synchronous JavaScript function call, and measure the duration of it. Any exitless `target` can be scheduled in callbacks, thus making it an exiting `target`, but an exiting `target` cannot be run in an exitless manner.

4.3.2 Implicit clocks in browsers

Supposing that all explicit clocks were removed from the browser, it is still possible that a motivated attacker can measure fine-grained durations. Rather than query an explicit clock, the attacker can find some other feature of the browser that has a known or definable execution time and use that as an *implicit clock*.

We did not test any clocks that resolve durations at an external observer, such as a cooperating server. For example, a piece of JavaScript could generate a network request, run a `target`, and then generate another network request. These clocks are mitigated by the defenses discussed in section 4.4.

We observe that just as with exiting and exitless `targets`, there are exiting and exitless implicit clocks. We will refer to a clock or timing method that does not need to leave JavaScript execution for the value reported by the clock to change as *exitless*. Similarly, a timing method that requires JavaScript execution to exit before time moves forward is *exiting*.

All exitless clocks can work for both exiting and exitless targets. However, an exitless target cannot function with an exiting clock, as the execution of the target will take control of the main thread, stopping regular callbacks or events that the exiting clock needs from firing. There may be exotic exiting clocks that do not have this restriction, but all of the ones detailed below do. An exitless attack requires using both an exitless target and clock (such as in the cache timing attack.)

Depending on the implementation of a browser feature, the clock technique may be exiting or exitless. A good example is the updating of the `played` information for an `<audio>` or `<video>` tag. This information is updated asynchronously to the main browser thread in Google Chrome but will not update during JavaScript execution in Firefox. Thus, it can be used to construct a exitless clock in Chrome but only an exiting clock in Firefox.

See Table 4.2 for how the following clocks manifest in Chrome 48 (stable), Firefox[3], and

[3]commit 0ec3174fe63d8139f842ce9eb6639349759ff4e5

Table 4.2. Implicit clock type in different browsers
L Exitless , X Exiting , — Not implemented, + Buggy

Description	Clock type		
	Firefox	**Chrome**	**Safari**
Explicit clocks	L	L	L
Video frames	L	L	L
Video `played`	X	L	L
WebSpeech API	L	+	—
`setTimeout`	X	X	X
CSS Animations	X	X	X
WebVTT API	X	X	X
Rate-limited server	X	X	X

Safari 9.0.3.

Exitless clocks

Since JavaScript is single threaded and non-preemptable, exitless clocks do not have to worry about the scheduling of other JavaScript callbacks or any other events occurring between the target and timing measurements. By the semantics of JavaScript, an exitless clock is considered a run-to-completion violation[66] and is a bug. Any time JavaScript can observe changes caused externally during a single callback qualifies as such a bug; it is only when their timing is dependable that we can construct a clock. Mozilla has explicitly stated their goal to make SpiderMonkey (the Firefox JavaScript engine) free of run-to-completion violations.

We found several exitless clocks available to JavaScript in different browsers.

1. Explicit clock queries. While expected, explicit clock queries are run-to-completion violations and expose the most accurate timing data. `performance.now` is the best source of explicit timing data in JavaScript.

2. Video frame data. By rendering a `<video>` to `<canvas>`, JavaScript can recover the current video frame. Since the video updates asynchronous to the browser event loop, this can be used to get a fine grained time-since-video-start value repeatedly.

On Firefox, video frame data updates at 60 FPS, giving a granularity of 17ms. We can load a video at 120FPS, which does not allow JavaScript access to new frames faster, but the frames JavaScript gets are a more accurate clock. We demonstrate this by generating a long-running video at 120FPS that changes the color of the entire video every frame. Thus, by sampling the current color via rendering the video to `<canvas>`, the page can measure how much time has elapsed since the video started. Video can be rendered off-screen or otherwise invisible to the user and will still update at 60FPS, making it an ideal choice for an implicit clock. We have also found that using multiple videos and averaging the reported time between them provides additional accuracy.

3. WebSpeech API. This can start/stop the speaking of a phrase from JavaScript and will give a high-resolution duration measurement when stopped. The WebSpeech API allows JavaScript to define a `SpeechSynthesisUtterance`, which contains a phrase to speak. This process can be started with `speak()` and then stopped at any time with `cancel()`. The cancelation can fire a callback whose event contains a high resolution duration of how long the system was speaking for. Thus, the attacker can start a phrase, run some target JavaScript function, and then cancel the phrase to obtain a timing target. Note that while the callback must fire to get the duration value, the duration measurement stops when `window.speechSynthesis.cancel()` is called, not when the callback eventually fires. This makes the WebSpeech API a pseudo-exitless clock in Firefox, even though we must technically wait for a callback to get back the duration measurement. Time moved forward, we just couldn't observe repeatedly. Since we can only measure the clock by stopping it, the clock-edge technique cannot be used to enhance the accuracy of the clock.

   The WebSpeech API is only supported in Firefox 44+, and on many systems will need to be manually enabled in `about:config`. Additionally, unless the OS has speech synthesis support, the clock cannot be used as it will never start speaking. Ubuntu can get this support by installing the `speech-dispatcher` package.

4. SharedArrayBuffers. While we did not test these, as the implementation is still ongoing, any sort of shared memory between JavaScript instances constitutes an exitless clock. As demonstrated in [78], this can be used as a very precise clock in real attacks.

Exiting clocks

Exiting clocks are far more numerous but also significantly less useful to an attacker, as their measurements and target execution are unlikely to be continuous.

1. `setTimeout`. Set to fire every millisecond, these then set a globally visible "time" variable when they do. This is the most basic of the exiting clocks. We set timeouts every millisecond as this is lowest resolution that can be set.

2. CSS animations. Set to finish every millisecond, these then set a globally visible "time" variable in their completion callback. These behave almost identically to `setTimeouts` and are measured in the same way.

3. WebVTT. This API can set subtitles for a `<video>` with up to millisecond precision and check which subtitles are currently displayed. The WebVTT interface provides a way for `<video>` elements to have subtitles or captions with the `<track>` element. These captions are loaded from a specified VTT file, which can specify arbitrary subtitles to appear for unlimited duration with up to millisecond precision. By setting a different subtitle to appear every millisecond, the page can determine how much time has elapsed since the video started by checking the `track.activeCues` attribute of the `<track>` element. This only updates when JavaScript is not executing.

4. A rate limited download. Using a cooperating server to send a file to the page at a known rate causes regular progress updates to be queued in callbacks. Using the `onprogress` event for XMLHTTPRequests (XHRs), the page can get a consistent stream of callbacks to a clock update function. Note that the rate of these callbacks is related to the size of the file being

retrieved, as well as the upload rate of the server. In our experiments, we used a file 100mB in size, with a server rate limited to 100kB/s using the Linux utility `trickle`. The page then assumes that the server is sending data at exactly 100kB/s and has an initial learning period to determine the rate at which the `onprogress` callbacks fire. After that is complete, the page can continue running as usual, with the assumption that it now has a regular callback firing at the calculated rate. Note that the `onprogress` events can also be requested to fire during the loading of `<video>` elements.

5. Video/audio tag `played` data. These contain the intervals of the media object that have thus far been played. By checking the furthest played point repeatedly, we can measure the duration of events. In Firefox, this only updates after JavaScript exits, but in Chrome, it updates asynchronously (making it an exitless clock for Chrome).

6. Cooperating iframes/popups from same origin. By creating a popup in the same origin, or by embedding iframes from the origin, two pages can cooperate and act on the same DOM elements. In our testing there was no way to get exitless DOM element manipulations updates in this situation. Thus, this case reduces to the `setTimeout` case or another similar method. We do not present any timing results for these clocks. Critically, if a method of sharing DOM element updates exitlessly were found this would become an exitless clock.

4.3.3 Performance of implicit clocks

The granularity, precision, and accuracy of implicit clocks varies widely by technique. We observe that most implicit clocks can be improved with the clock-edge technique from section 4.2. By substituting the `performance.now` major clock with the implicit clock technique, and using a suitable minor clock, most techniques showed notable improvements in accuracy. In this case, we want to examine how easy it would be to differentiate two different duration events. Thus, tight error bounds that are consistent are ideal.

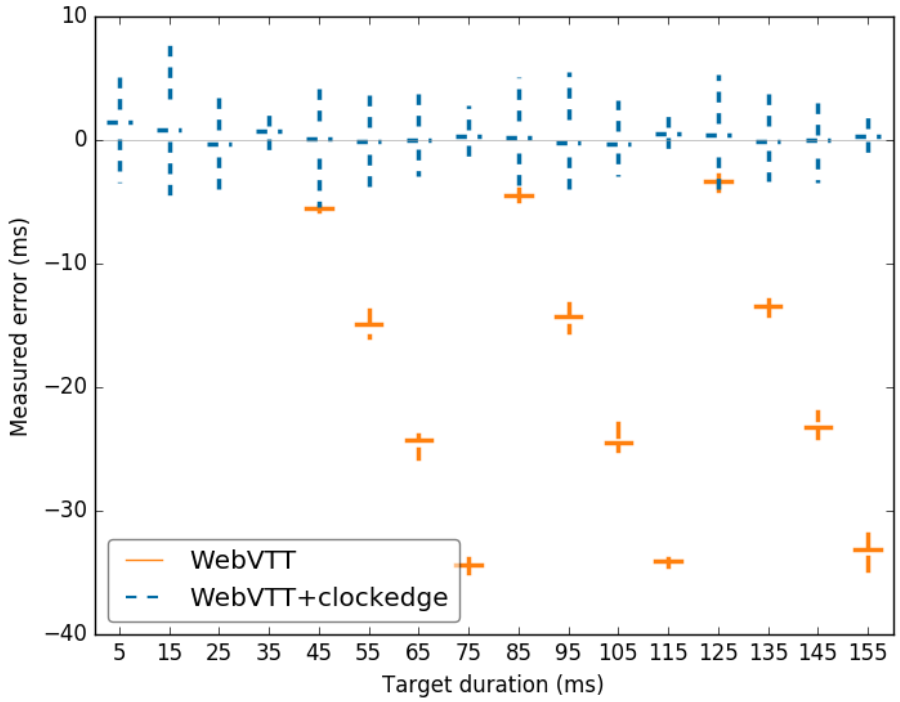


Figure 4.4. WebVTT error measurements with and without clock-edge technique

Applying the clock-edge technique to exitless clocks only requires the replacement of the explicit `performance.now` call to some other exitless clock; no change to the minor clock is needed. Exiting clocks require a new minor clock technique; instead of a tight loop, the minor clock must schedule regular timeouts that check the state of the implicit major clock. Otherwise, the exiting major clock would not change state while the minor clock is running. While repeated `setTimeout` calls would work, `setTimeout` of 0 is actually a 4ms timeout per the HTML5 spec, making it a major clock. Instead, we use repeated `postMessage` calls to the current window. These execute at a much higher rate, but the period is unknown. Thus the new implicit major clock now has a fast, unknown period minor clock, just as in the exitless case.

Measurements were done with the same Firefox as in section 4.2. Error (y values) was calculated as the difference between the clock technique measurement and the actual duration as reported by `performance.now`. Target durations (x values) are the expected duration (*N* milliseconds) of the target event, which may differ slightly from actual duration due to system

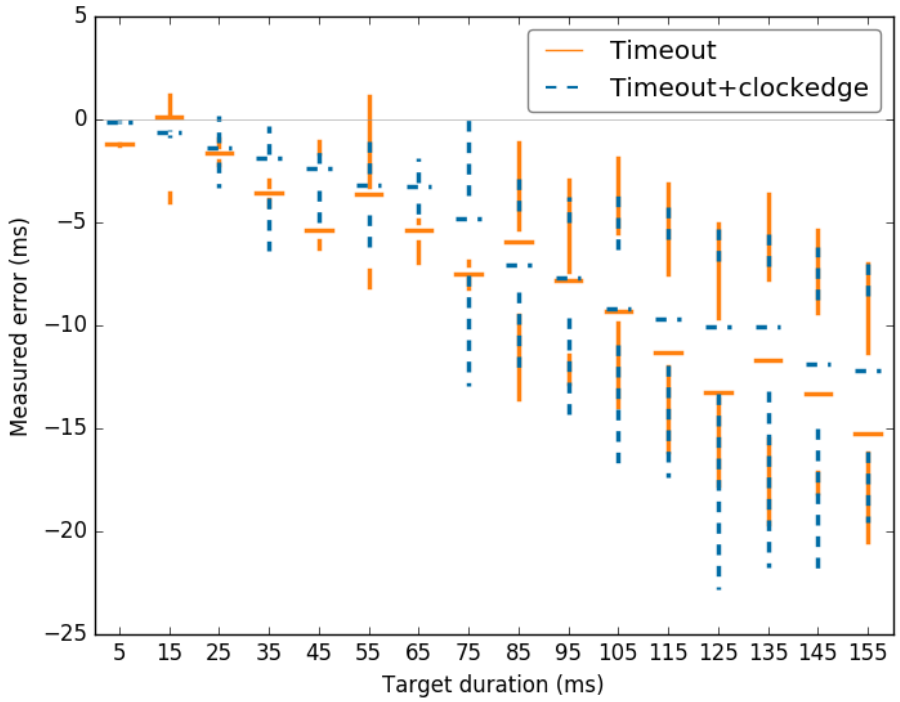


Figure 4.5. `setTimeout` error measurements with and without clock-edge technique

load or even the implicit clocks themselves interfering in the case of exiting clocks. Each target was measured 100 times, with measured durations of 0 or less removed. While actual durations varied slightly from expected, there was not considerable noise.

The exitless target we measure is a loop that runs for N milliseconds, as determined by `performance.now`. Our exiting target is a `setTimeout` for N milliseconds.

Figures 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9 show the clock technique error with and without clock-edge improvements for a variety of clock techniques described above. WebSpeech has no clockedge data for the reasons detailed in 4.3.2. Note that the y-axis differs per figure, to allow for easier comparison between clock-edge and non-clock-edge results. As can be seen in WebVTT, throttled XHRs, and video frame data, many clock techniques have a large native period that they operate at. These large periods leave plenty of space for clock-edge to improve accuracy. WebVTT shows massive improvement in the clock-edge case due to the *precision* of its major clock ticks; the more precise the original technique, the more accurate clock-edge can

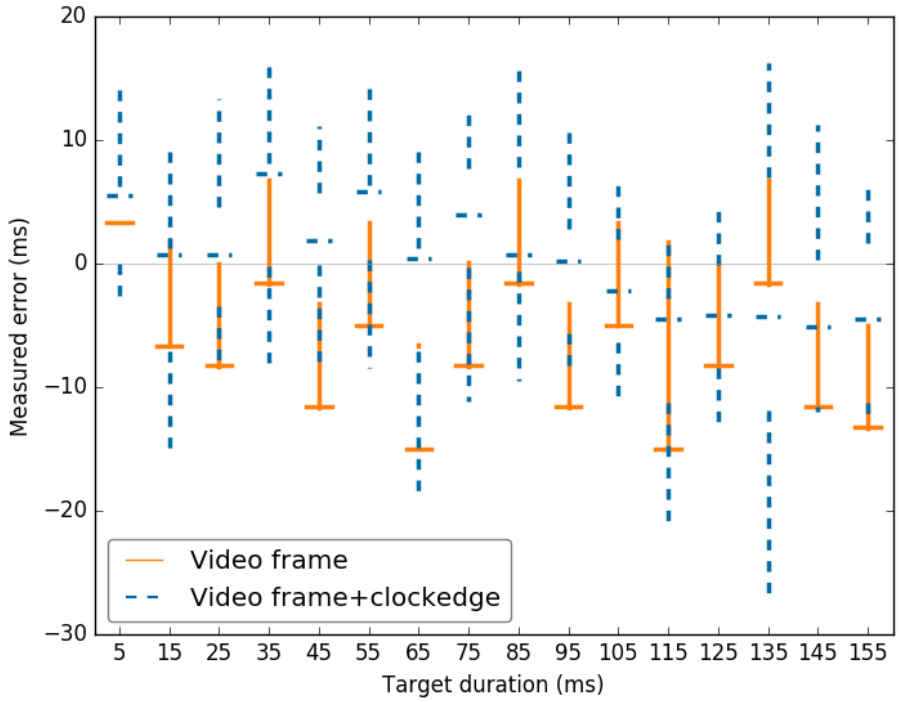


Figure 4.6. Video frame error measurements with and without clock-edge technique

be.

Figures 4.10b and 4.10a show the comparison of the averaged error for all techniques and all techniques with clock-edge respectively. The closer a line is to 0 on these graphs, the more accurate the averaged measurements will be for that technique. Again, the exceptional accuracy of WebVTT with clock-edge for long-duration events is evident.

4.4 Fermata

In this section we describe *Fermata*, a theoretical browser design that provably degrades all attacker visible clocks. Sections 4.5 and 4.6 describe our prototype implementation, Fuzzyfox, and an evaluation. Fermata is an adaptation of the *fuzzy time* operating systems concept detailed in [44] to web browsers.

Since browser vendors have expressed an interest in degrading time sources available to JavaScript, we present Fermata as a design ideal for a browser that will provably degrade all

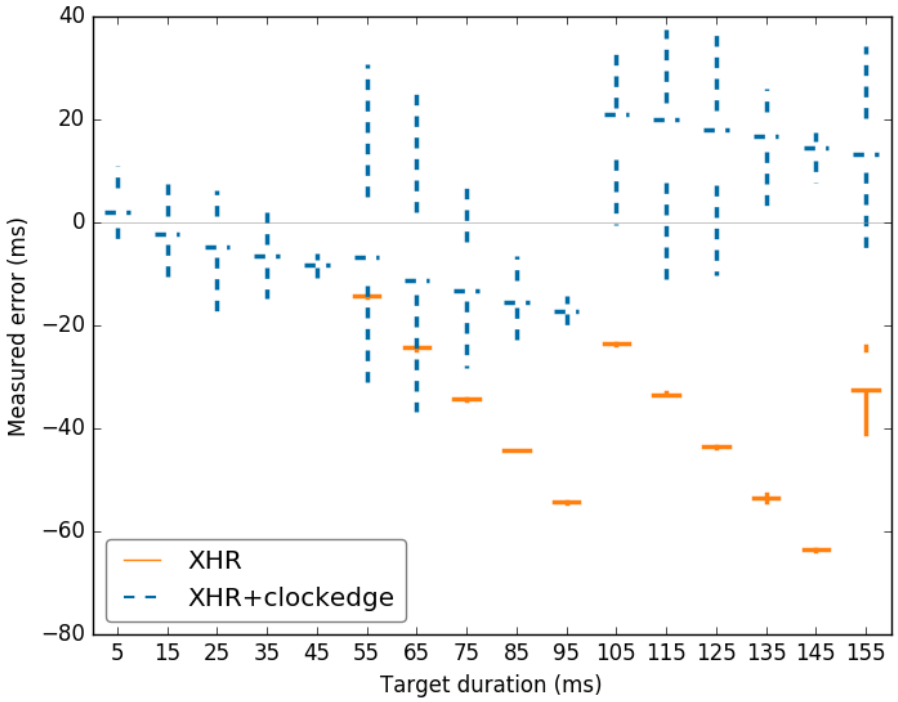


Figure 4.7. Throttled XMLHTTPRequest error measurements with and without clock-edge technique

clocks. Fermata's goal is to provide the attacker with only time sources that update at a rate such that all possible timing side channels have a bounded maximum bandwidth. This includes the use of all the implicit clocks described in section 4.3 as well as any other such clock unknown to us.

4.4.1 Why Fermata?

We propose Fermata because we believe that attempting to audit and secure all possible channels in a modern web browser is infeasible. The evaluation of a provable security focused microkernel found several tricky timing channels [19]. In that case, the microkernel was designed to be audited and already had a number of concerns accounted for; this is not true in the case of a modern web browser. Rather than allow any unknown channel to leak data arbitrarily until fixed, Fermata restricts all known and unknown channels to leak at or below a target acceptable rate.

Fermata proposes a principled alternative to the "find and mitigate all clocks" methodol-

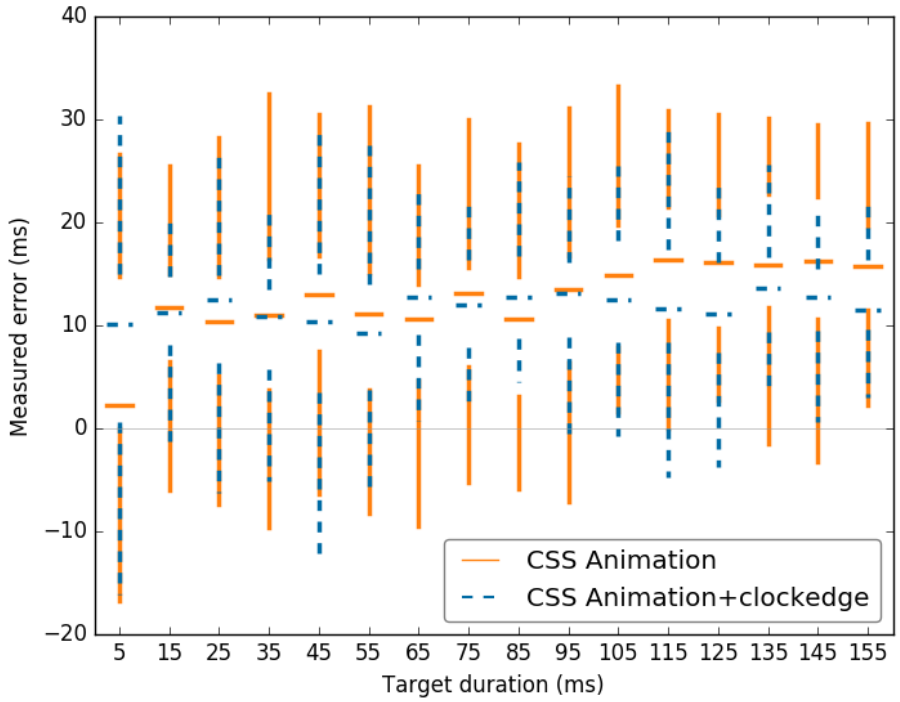


Figure 4.8. CSS animation error measurements with and without clock-edge technique

ogy that Tor Browser has already begun. Rather than manually examine every DOM manipulation, extension, or new feature, Fermata requires minimal defined interfaces between all components. By automatedly proving that all information passes through these interfaces and that all such interfaces are subject to the fuzzying process, Fermata will drastically reduce the burden of code that needs to be examined. This is analogous to other such approaches in the programming languages and formal software community.

Limiting the channel bandwidth for an attacker leaking information is not a complete solution to timing attacks on browsers, but it is a realistic one. Previous attacks on history sniffing [8] [81] have consistently cropped up. These privacy breaches are only as valuable as the amount of data they can collect. Learning that a user has visited 2-3 websites is not likely to create a unique profile of them. Learning tens of thousands of websites likely would [86]. History sniffing attacks are therefore classified based on how fast they can extract the visited status of a URL. By limiting the rate at which this information can leak, Fermata can make

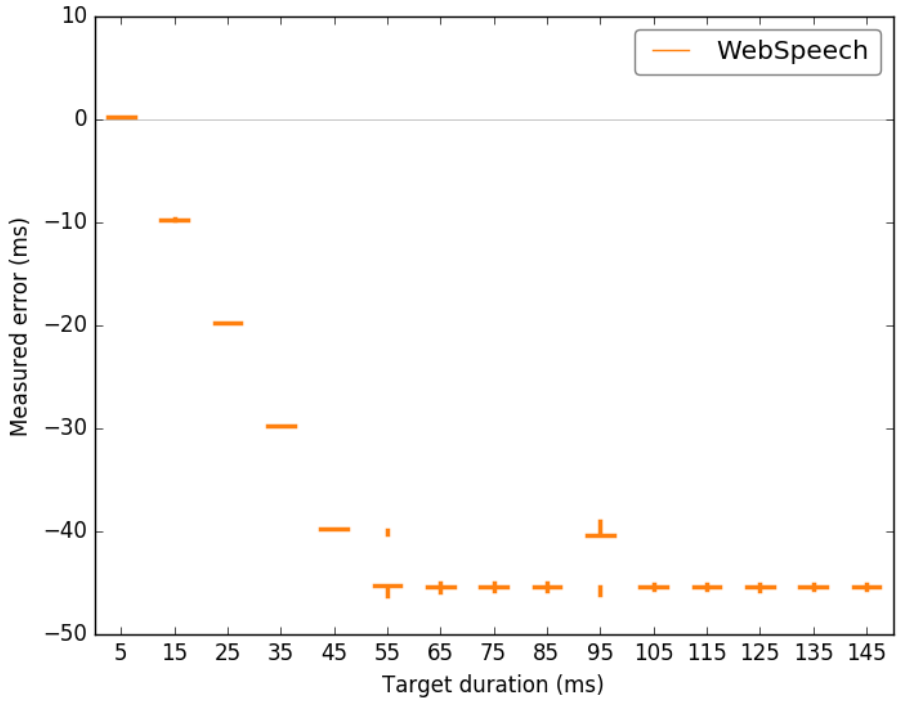
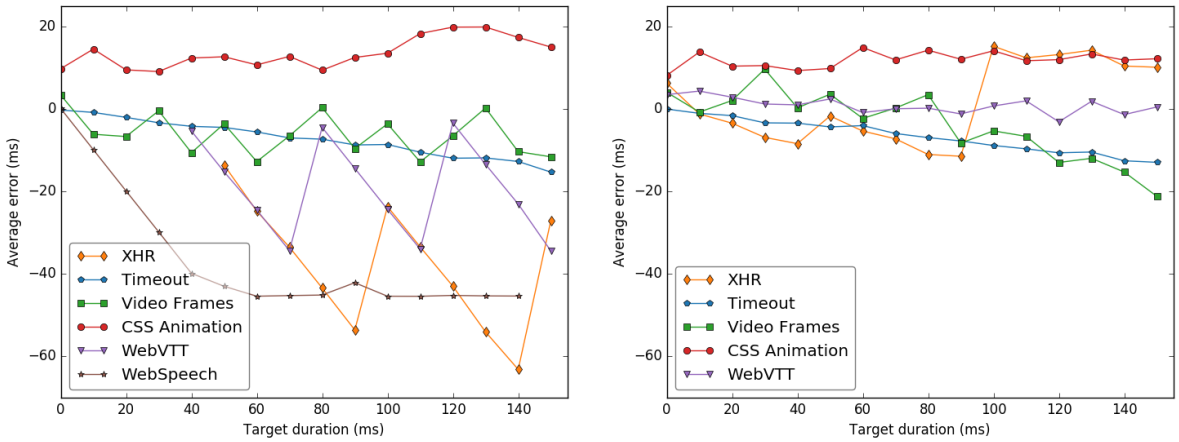


Figure 4.9. WebSpeech error measurements without clock-edge technique

history sniffing impractical. As an example, [86] indicates that an attacker may need to sniff in excess of 10,000 URLs to create a reasonable fingerprint for a user. With an attack like [81] the attacker can read 60 or more URLs per second. Previous attacks not utilizing timing side channels read in excess of 30,000 URLs per second.

We expect that Fermata would allow a channel bandwidth of ≤ 50 bits per second in the general case, and ≤ 10 for security critical workflows. The protection is even stronger than initially obvious, as attacks that rely on small timing differences are entirely unusable. Only attacks that can scale their detection thresholds up (for example, Andrysco et al [8]) can still leak data. If the attack relies on a small, inherent microarchitecture timing, such as Oren et al's [67] cache timing attack, which measured differences around 100ns, this timing difference may no longer be perceptible at all. An additional benefit is that many of these attacks require intensive learning phases, during which many measurements must be taken to establish timing profiles. Fermata would force this learning phase to take significantly longer, adding to the time-per-bit of



(a) Error without clock-edge

(b) Error with clock-edge where available

Figure 4.10. Average error for all clock techniques with and without clock-edge

information extracted. From this survey of previous attacks, we believe that a strong limitation on channel bandwidth represents an powerful defense against timing attacks in browsers.

4.4.2 Threat model

We define our attacker as the canonical web attacker who legitimately controls some domain and server. They are able to cause the victim to visit this page in Fermata and run associated JavaScript. The attacker thus has two viewpoints we must consider: any external server controlled by the attacker and the JavaScript running in Fermata.

The attacker in our case possesses a timing side-channel vulnerability they wish to use on Fermata. The specific form of the vulnerability does not matter, only that it can be abstracted as a single JavaScript function that is called either synchronously or asynchronously. The attacker uses the duration of this function to derive secret information about the victim, possibly repeatedly.

We do not present a solution for plugins like Adobe Flash or Java applets. Significant changes to the runtime of these plugins on-par with Fermata itself would need to be made for them to be similarly resistant. Considering the number of known vulnerabilities and privacy disclosures in most of these plugins, we do not believe they should be a part of a browser design

focusing on security and privacy. Alternatively, such plugins should be disabled during sensitive work flows.

The attacker succeeds against Fermata if they are able to extract bits using their side channel at a higher rate than the maximum channel bandwidth.

4.4.3 Design goals and challenges for Fermata

Fermata must mediate the execution of JavaScript to remove all exitless clocks and degrade all exiting clocks. This would include mediating and randomly delaying all network I/O, local I/O, communication between JavaScript instances (iframes, workers, etc), and communication to other processes (IPC). If Fermata were additionally able to make all DOM accesses by JavaScript asynchronous and delay them in the same principled fashion, this would accomplish our goals. The coupling of JavaScript's globally accessible variables to the DOM represents the most significant challenge to such a design and presents a shared state problem not found in the model for this work [44].

Given this shared state problem, Fermata has two options for JavaScript: redesign JavaScript execution to be entirely asynchronous or degrade explicit clocks and mediate known APIs in a principled manner. The former provides a formal guarantee but cannot be done in current browser architectures. We explore options for the latter later in this section and in Fuzzyfox.

4.4.4 Fermata guarantees

We believe that the analysis of Hu's fuzzytime by Gray in [35] applies to Fermata. The means that we can place an upper bound on the leakage rate of Fermata at $\frac{1}{g/2}$ symbols per second, assuming the median tick rate of $\frac{g}{2}$.

As in [35], we assume that increasing the size of the alphabet used will provide negligible benefits. Thus, this bound is an upper bound for the bits-per-second leakage rate of Fermata. We view the vulnerable functionality targeted by the attacker in the strongest possible way: the

attacker has complete control over when and how it leaks timing information. This is effectively the high/low privilege *covert channel* scenario the fuzzytime disk contention channel is analyzed under. Similarly, in Fermata, the leaking feature may have access to the same fuzzy clock as the attacker. This allows them to synchronize instantly from "low to high" privilege as in the fuzzytime analysis. Thus, the side channel threat model Fermata operates under is a subset of the fuzzy time model.

There is further analysis of the capacity of covert channels with fuzzy time defenses in [37]. The general case problem of covert channel capacity under fuzzy time appears to be intractable but can be bounded under specific circumstances.

Transmitted bits vs information learned

Fermata makes a guarantee about the actual transmitted bitrate of some side channel. This has obvious benefits in the case of leaking a CSRF token or a cryptographic key: the bits the attacker needs to learn equals the number of bits in the key or token. However, this becomes trickier to quantify with a goal like history sniffing where the details of the side channel can influence what the attacker learns with each leaked bit.

Consider a timing side channel that can indicate if a single URL has been visited by the victim one at a time. Each time the channel is used one bit of information (visit status of the URL) is leaked. If the attacker wishes to learn the visit status of 10,000 URLs they must check each individually.

If instead a timing side channel could indicate if any URLs from an arbitrary set were visited, the attacker could use this along with prior knowledge that almost all URLs have not been visited to learn about more URLs in less bits. Given some set of 10,000 URLs, the side channel indicates that at least one was visited and then, in a divide-and-conquer approach, the first half indicates that none were visited. How many bits were leaked? Two bits were transmitted: that some URLs were visited in the 10,000, and that no URLs in the first 5,000 were visited. However, we have learned the visit status of 5,000 URLs. This is only possible because the attacker can

assume the majority of URLs are not visited.

We believe that Fermata's guarantees still constitute a valuable defense against using timing side channels for history sniffing. First, not all history sniffing side channels have allowed checking the visit status of batches of URLs. In these cases Fermata limits learning the visit status of each URL individually. Second, if the attacker wishes to learn specific URLs from the browsing history (ex: to launch a targeted phishing attack), rather than just learn a rough fingerprint, they will still need to examine each individual URL regardless of how the side channel can operate.

Fermata cannot provably *prevent* a timing side channel from operating; it can only constrain the rate of bits transmitted across the channel. For any side channel it is important to consider the attacker's goals along with how the side channel operates to understand what level of mitigation Fermata will provide. There are multiple reasons (compression, prior knowledge, etc.) that might lead to a side channel exhibiting behavior like described above. In all of these cases Fermata provides the same guarantee about channel bandwidth.

4.4.5 Isolating JavaScript from the world

A potential solution for JavaScript is to remove all run-to-completion violations, effectively ensuring that JavaScript cannot observe any state changes to the DOM or otherwise during a single execution. This necessarily includes all realtime clock accesses, as well as any other discovered exitless clocks. Since JavaScript will always have access to a fine grained minor clock (the `for` loop), it is critical that all exitless *major* clocks be removed. In the case of `performance.now`, this will result in the feature becoming an exiting clock, requiring that JavaScript stop execution before the available clock value changes.

The catch of the latter method is in how to remove all potential exitless clocks. If the upcoming SharedArrayBuffer API becomes available, this presents a highly accurate exitless clock that Fermata cannot mitigate without returning it to a message passing interface. Removing all of these potential exitless clocks requires an examination of all interfaces the JavaScript

runtime has.

With all exitless clocks removed, the design need only focus on degrading exiting clocks to meet the target maximum channel bandwidth.

4.4.6 Degrading explicit clocks

Explicit clocks (ex: `performance.now`, `Date`, etc.) are degraded to some granularity g and update unpredictably. As in Hu [44], we accomplish this by performing updates to the clock value (at the granularity g) at randomized intervals. g is a multiple of the native OS time grain g_n (generally 1ns). Each randomized interval is a "tick," during which the available explicit clocks do not change. At the beginning of each tick, we update the Fermata clock to the rounded-down wallclock. Since the tick duration is not the same as g, the Fermata clocks will not always change in value every tick. This design guarantees that the available explicit clocks are only ever behind and are behind by a bounded amount of time, $g - g_n + (g/2)$. Note that a clock's granularity does not alone define the accuracy to which it can be used to time some event, as seen with section 4.2.

Tick duration is not constant but is instead drawn from a uniform distribution with a mean of $g/2$. If intervals were constant and thus clock updates occurred exactly on the grain, the attacker could use the same clock-edge technique as in section 4.2.

4.4.7 Delaying events

The randomized update intervals (ticks) are further divided into alternating upticks and downticks for the purposes of delaying events and I/O. This mimics their usage in Hu [44]. Downticks cause outbound queued events to be flushed, and upticks cause inbound events to be delivered.

4.4.8 Tuning Fermata

Since the defensive guarantee provided by Fermata is only a maximum channel bandwidth, a few users may want to change the tradeoff between responsiveness and privacy. Fermata will provide this option via a tunable privacy setting that allows setting the acceptable leaking channel bandwidth. In turn, this will modify the average tick duration and the explicit time granularity, both of which affect usability. We expect that only developers (including of browser forks like Tor Browser) or users with specific privacy needs would interact with these settings.

4.5 Fuzzyfox prototype implementation

In this section we describe *Fuzzyfox*[4], a prototype implementing many of the principles of the Fermata design in Mozilla Firefox. Fuzzyfox is not a complete Fermata solution but does show that the removal of exitless clocks and the delaying of events is a feasible design strategy for a browser.

Fuzzyfox attempts to mitigate the clocks of sections 4.2 and 4.3 by using the ideas in Fermata. Web browsers have an interest in degrading clocks available to JavaScript to reduce the impact of both known and unknown timing channel attacks. Fuzzyfox is a concrete demonstration of techniques that will make a browser more resistant to such timing attacks. As in Fermata, Fuzzyfox has a clock grain setting (g) and an average tick duration ($t_a = g/2$). All explicit clocks in Fuzzyfox report multiples of g.

We will refer to Firefox when discussing default behavior and Fuzzyfox when discussing the changes made.

4.5.1 Why Fuzzyfox?

We built Fuzzyfox for three reasons:

1. Building a new web browser is a monumental task.

[4]Fuzzyfox is available as a branch at https://github.com/dkohlbre/gecko-dev. It should be treated as an engineering prototype.

2. We did not know if a Fermata-style design would result in a usable experience. It was entirely possible that the delays induced would render any Fermata-style designs unusable.
3. We want to deploy the insights of channel bandwidth mitigation to real systems like Tor Browser.

Fuzzyfox does *not* have the complete auditability advantages that Fermata would. However, we believe that our insights about principled fuzzying of explicit clocks can be directly applied to Tor Browser as an improvement to their ongoing efforts.

4.5.2 PauseTask

The core of the Fuzzyfox implementation is the `PauseTask`, a recurring event on the main thread event queue. The `PauseTask` provides two primary functions: it implicitly divides the execution of the event queue into discrete intervals, and it serves as the arbiter of uptick and downtick events.

Once Firefox has begun queuing events on the event queue, Fuzzyfox ensures that the first `PauseTask` gets added to the queue. From this point on, there will always be exactly one `PauseTask` on the event queue.

`PauseTask` does the following on each execution: determines remaining duration, generates retroactive ticks, sleeps remaining duration, updates clocks, flushes queues, and queues the next `PauseTask`.

Determine remaining duration

The `PauseTask` checks the current OS realtime clock (T_1) with microsecond accuracy using `gettimeofday`. Comparing this against the expected time between ticks (D_e) and the end of the last `PauseTask` (T_2) gives the actual duration (D_a). If $D_a \leq D_e$, `PauseTask` skips directly to sleeping away the remaining duration, $D_e - D_a$.

Optional: Retroactive ticks

Otherwise, `PauseTask` must retroactively generate the upticks and downticks that should have occurred. This ensures that even by being long running JavaScript cannot force a 0 sleep duration `PauseTask`.

Sleep remaining duration

PauseTask finishes out the remaining duration via `usleep`. `usleep` is not perfectly accurate, and has a fixed overhead cost. In our testing, `usleep` error varies based on the duration but is never enough to be an issue for Fuzzyfox.

Update all system clocks and flush queues

`PauseTask` now generates the new canonical system time. This is accomplished by taking the OS realtime clock and rounding down to the Fuzzyfox clock grain setting.

There are two underlying explicit time sources available to JavaScript, `Time` and `performance`. `PauseTask` directly updates the canonical TimeStamp time, which is used by `performance`, and delivers a message to the JavaScript runtimes to update `Time`'s canonical time. Our review found that all of the other time sources we knew of used TimeStamp.

In our prototype, the only I/O queue that needs to be flushed is the DelayChannelQueue (see section 4.5.3.) This only occurs if the currently executing `PauseTask` is a downtick.

Queue next `PauseTask` event

Finally, `PauseTask` queues the next `PauseTask` on the event queue. This sets the start time (T_1), marks the new `PauseTask` as either uptick or downtick, as well as drawing a random duration from the uniformly random distribution between 1 to $2 \times t_a$. `PauseTask`s are queued exclusively on the main thread to ensure they block JavaScript execution as well as all DOM manipulation events.

4.5.3 Queuing

All events visible to JavaScript must be queued in Fuzzyfox. Unfortunately, there is not a singular place or even explicit queues available for all events in Firefox. We use `PauseTask` to create implicit queues for all main thread events (including JavaScript callbacks, all DOM manipulations, all animations, and others) and construct our own queuing for network connections.

Timer events (including CSS animations, `setTimeout`, etc.) do not need to be explicitly modified from Firefox behavior, as they run in a separate thread that checks when timers should fire based on TimeStamp. As Fuzzyfox ensures all TimeStamps are set to our canonical Fuzzyfox time, this is not a problem.

DelayChannelQueue

We implemented a simple arbitrary length queue for outgoing network connections called `DelayChannelQueue`. This queue contains any channels that have started to open and stops them from connecting to their external resource. In the Fuzzyfox prototype, we only queue outgoing HTTP requests, although it could easily be extended to more channel types. Upon receiving a downtick notification from `PauseTask`, the queue is locked and all currently queued channel connections are completed and flushed from the queue.

4.6 Fuzzyfox evaluation

We evaluated our prototype Fuzzyfox in both effectiveness (how it degrades clocks) and performance.

All evaluations are compared against a clean Firefox build without the Fuzzyfox patches. Firefox trunk[5] was used as the basis and built with default build settings. Fuzzyfox patches are then applied on top of this commit and built with the same configuration. All tests were

[5]Firefox tests were done with commit 0ec3174fe63d8139f842ce9eb6639349759ff4e5 for clock tests, and c4afaf3404986ccc1d221bc7f4f3f1dcf39b06fc for the page load tests

performed on an updated Ubuntu 14.04 machine with an Intel i5-4460 and 14GB of RAM. The only applications running during testing were the XFCE window manager and Fuzzyfox. Fuzzyfox and Firefox were both tested using the experimental e10s Firefox architecture. NSPR logging was enabled to capture data about Fuzzyfox internals.

4.6.1 Limitations

Fuzzyfox is not a complete Fermata implementation and is unable to guarantee a maximum channel bandwidth. Since we did not isolate the JavaScript engine from the DOM or all I/O operations, we did not interpose on all interfaces as would be required in a Fermata implementation. This is purely a practical decision, as accomplishing this in Firefox would require manually auditing the entire codebase. We do not, for example, interpose on synchronous IPC calls from JavaScript. See section 4.6.2 for an example of how this can break the Fermata guarantees.

Unfortunately, since our `PauseTasks` can be delayed by long running JavaScript on the main thread, we can no longer bound the difference between the OS realtime clock and the available explicit clocks. We do still guarantee that all explicit clocks are only ever behind realtime.

While we experimented with a number of different grain settings, the settings providing very high privacy guarantees (100s of milliseconds) have severe usability impact. We believe that a clean Fermata implementation may not incur such a strong usability impact at similar grain settings.

4.6.2 Effectiveness

Effectiveness is measured as the available resolution for a given clock. In the ideal case, all clocks in Fuzzyfox should be degraded provide a resolution no less than g. We measure the observed properties of the clocks described in section 4.3 between Firefox and Fuzzyfox. We set the explicit time granularity (g) to 100ms and the average `PauseTask` interval (t_a) to 50ms for

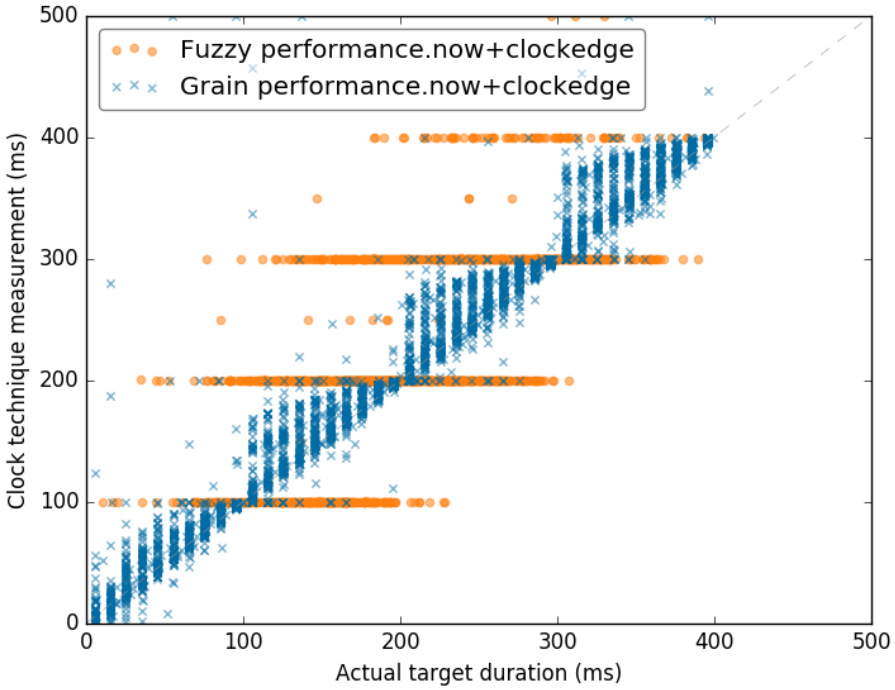


Figure 4.11. `performance.now` measurements with clock-edge on Fuzzyfox (exiting) and Firefox (exitless, 100ms grain)

these tests. We chose $g = 100ms$ because a large g value most clearly illustrates the difference between Fuzzyfox and Firefox. See section 4.6.3 for an evaluation of the impact of high g values on performance.

The following figures show scatter plots for several clock techniques as they operate in Firefox and in Fuzzyfox. In each, a perfectly accurate clock would follow the dashed grey line on $x = y$. Note that these figures show actual duration and clock technique duration, rather than target duration and error as in section 4.3.3. This is due to Fuzzyfox being unable to dependably schedule targets less than g (100ms) in duration. Thus, while the same testing code was used in Fuzzyfox and in Firefox, the actual durations of events are much longer in Fuzzyfox. Finally, there are no exitless clocks that we know of in Fuzzyfox to test, which would have been a closer comparison.

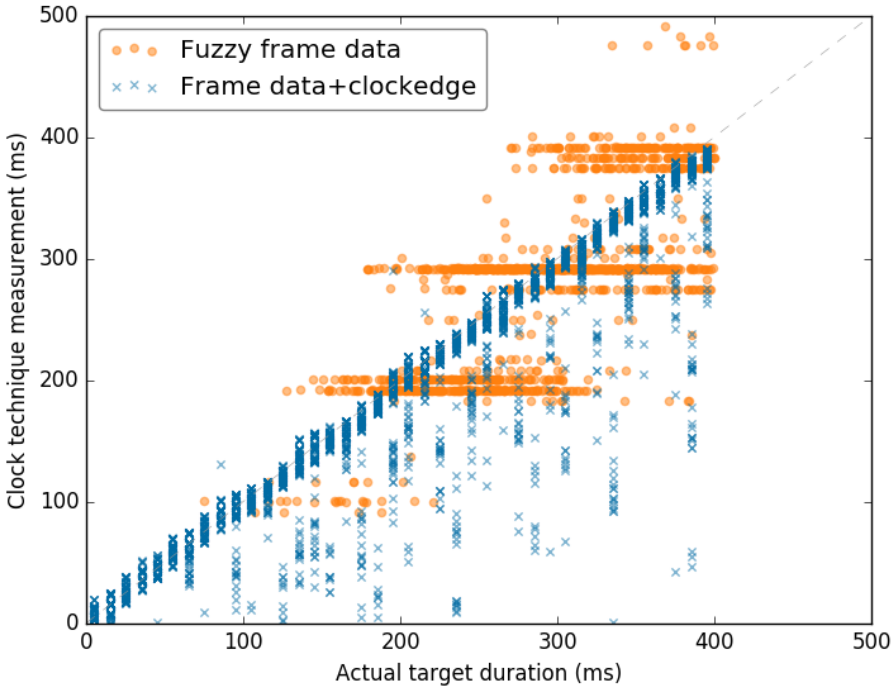


Figure 4.12. Frame data clock measurements on Firefox and Fuzzyfox

`performance.now`

Since time no longer moves forward during JavaScript execution, `performance.now` is now an exiting clock. Figure 4.11 shows the results of using the clock-edge technique on `performance.now` for both Fuzzyfox and Firefox with a grain set to 100ms. Notably, clock-edge no longer improves the accuracy of the measurements! This demonstrates that the Fuzzyfox model successfully degrades explicit clocks.

Video frame data

Unexpectedly, Fuzzyfox transforms the video frame data clock from exitless to exiting. This is probably because the frame extracted for canvas is determined using the current explicit clock values (TimeStamp.) Since time does not move forward during JavaScript execution, frame data is now an exiting clock. In general, we expect that run-to-completion violations (and by extension most exitless clocks) would not be properly degraded by Fuzzyfox. Figure 4.12 shows the exiting frame data clock on Fuzzyfox and Firefox.

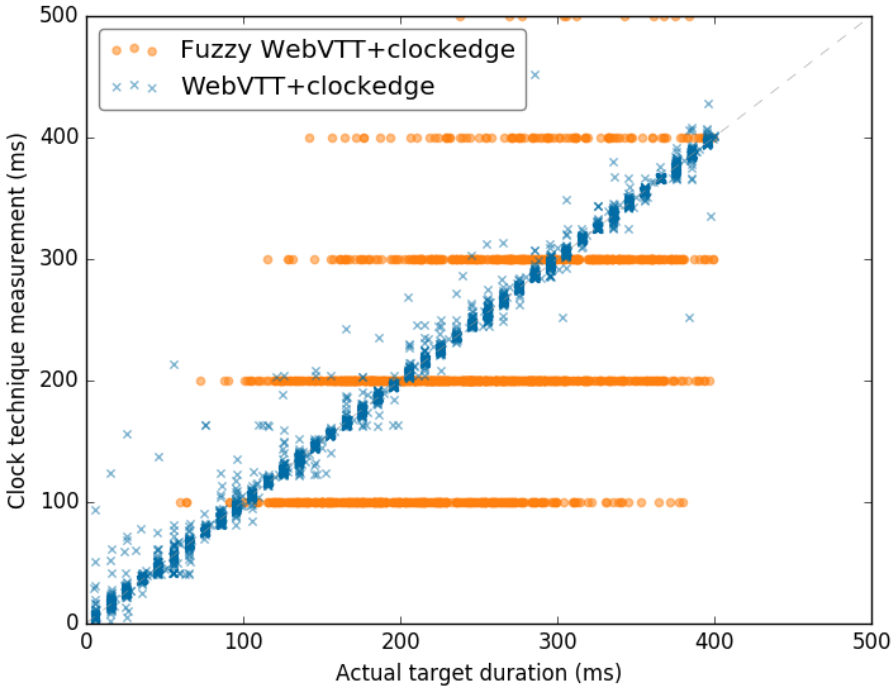


Figure 4.13. WebVTT clock measurements on Firefox and Fuzzyfox

WebSpeech API

Fuzzyfox degrades the WebSpeech API only because the `elapsedTime` field is drawn using the explicit clocks in Fuzzyfox. The starting and stopping of the speech is still synchronous, so it is possible some other piece of information passed back by the speech synthesis provider could provide a more accurate clock. WebSpeech should not be considered properly isolated by Fuzzyfox. Only if the *starting* and *stopping* of speech synthesis were queued like other events would Fuzzyfox correctly handle WebSpeech.

`setTimeout`

As `setTimeout` events are fired from the timer thread based on the degraded explicit clocks, they are no longer able to fire more often than the explicit time grain g of 100ms.

CSS Animations

As with `setTimeout`, CSS animation events are fired from the timer thread based on the degraded explicit clocks. Thus, they too are not able to be used as a clock of finer grain than the explicit time grain g.

XMLHTTPRequests

XMLHTTPRequests are properly degraded by Fuzzyfox. Since the callbacks for `on-progress` are queued on the main event queue and then gated by `PauseTask`, they are no longer timely when processed.

WebVTT subtitles

We examined the WebVTT subtitle implicit exiting clock in detail, as it performed among the best with the clock-edge technique on vanilla Firefox. Figure 4.13 shows the results for the same WebVTT clock techniques as described in section 4.3.2 on both Fuzzyfox and Firefox. Note that the clockedge code provided no benefits to the Fuzzyfox case.

4.6.3 Performance

Generalized performance impact is difficult to measure, as most performance tools for browsers rely on accurate time measurements via JavaScript.

We performed a series of page load time tests, which show predictable results. We measure the impact of both *depth* of page loads and the *spread* of initial requests. Our testing setup consisted of 20 test pages and 5 different fuzzyfox/Firefox configurations. The depth of the test pages represents how many sequential requests are made. Each request consists of inserting a script file of the form in figure 4.15. Each one has the loaded script be the next "layer" down, with layer 0 being an empty script. Thus, a test page that is 3 deep makes 4 sequential requests: `page.html`, `layer2.js`, `layer1.js`, `layer0.js`. Spread is achieved by the base `page.html` performing several duplicate initial requests to the top layer. Thus, a

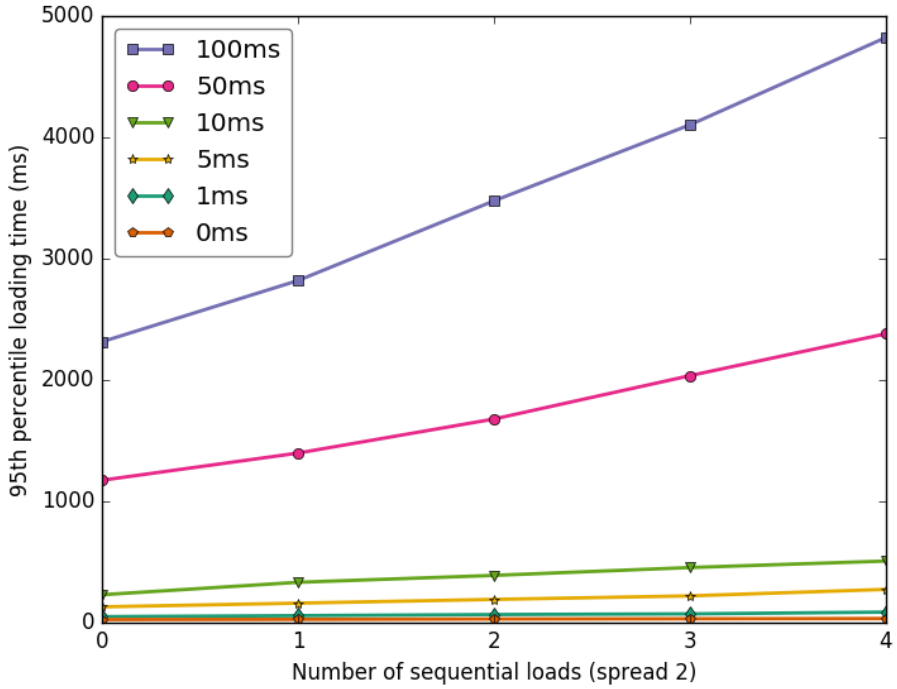


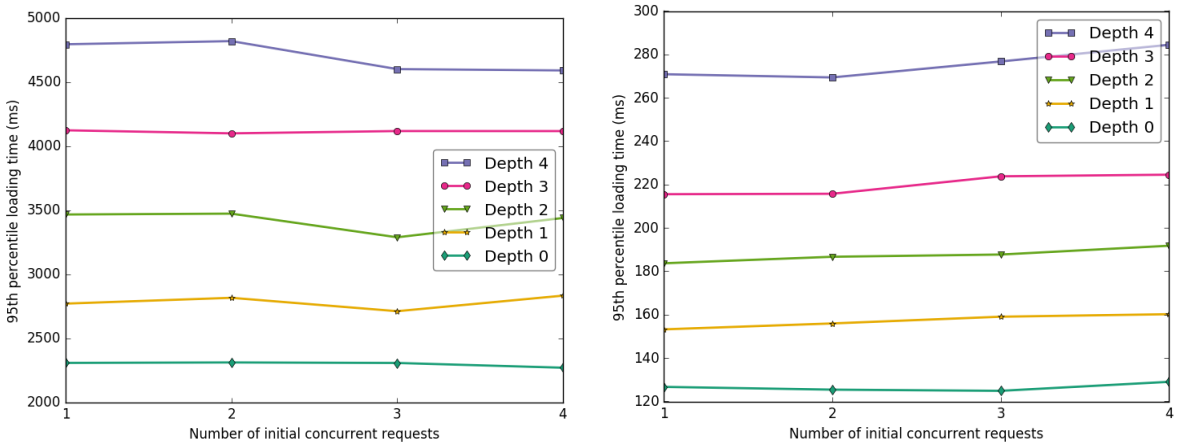
Figure 4.14. Page load times with variable depth for all Fuzzyfox configurations at a spread of 2

```
var njs=document.createElement('script')
njs.setAttribute('type','text/javascript')
njs.setAttribute('src','layer2.js')
document.getElementsByTagName('head')[0].appendChild(njs)
```

Figure 4.15. Iterative page load JavaScript

spread of 2 and a depth of 2 results in requests for: `page.html`, `layer1.js`, `layer1.js`, `layer0.js`, `layer0.js`. After the final page load completes, the total time from initial page navigation until completion is stored, and this process is repeated 1000 times per page test. We generate 20 test pages by combining up to 5 layers of depth with a spread from 1 to 5. We served the test pages via a basic nginx configuration running on the same host as the browser.

Figures 4.14 and 4.16a show two different views of some of the results, with the 95th percentile of load times being shown for $g = 100ms$. As expected, increasing the spread for a given depth (as shown in figure 4.16a) results in almost no change to load times. All other browser configurations (see figure 4.16b for $g = 5ms$) had nearly identical results, with differing



(a) Page load times for $g = 100ms$

(b) Page load times for $g = 5ms$

Figure 4.16. Page load times with variable spread and depth

y-intercepts based on g. This occurs because outgoing HTTP requests in Fuzzyfox are batched, so queuing multiple requests at once does not incur any g-scaled penalties. However, as figure 4.14 shows, increasing the depth incurs a linear overhead with the slope and intercept scaled by the value of g. The worst case for Fuzzyfox are pages that do large numbers of sequential loads, each requiring JavaScript to run before the next load can be queued. Unfortunately, many modern webpages end up performing repeated loads of various libraries and partial content. One potential solution would be more widespread use of HTTP2's Server Push which would alleviate the repeated g scaled penalties for resource requests.

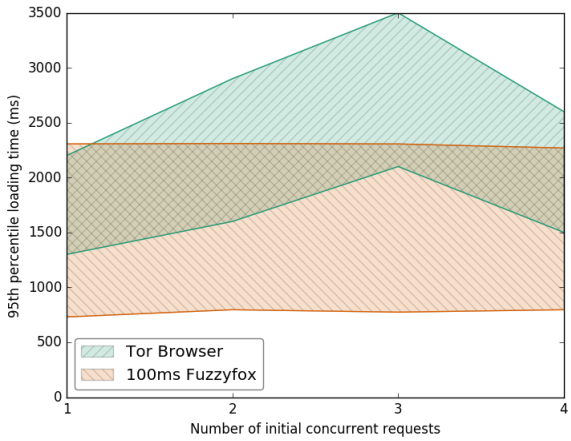
JavaScript engine tests, such as JetStream, reported identical scores of 181 for both Firefox and Fuzzyfox.[6] Fuzzyfox predictably records a maximum FPS equal to the average `PauseTask` fire rate or 20 FPS for $g = 100ms$, as compared to 60 FPS in the Firefox case.

Tor Browser

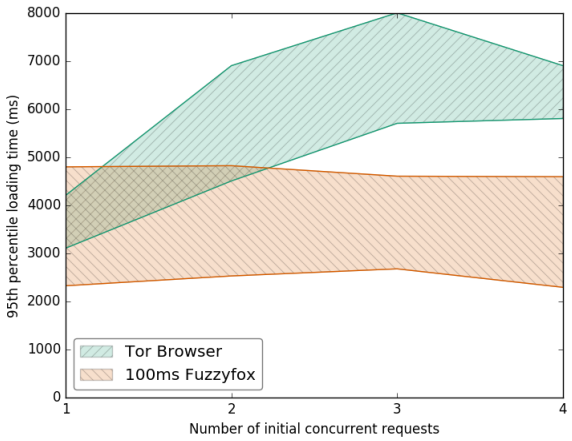
We also ran our page load tests on vanilla Tor Browser[7]. Rather than access the pages over the localhost interface, they are accessed over the Tor network. No other changes to the

[6]Fuzzyfox was modified to report valid `performance.now` results for performance testing

[7]Tor Browser git revision: b60b8871fa08feaaca24bcf6dff43df0cd1c5f29 modified to report accurate performance.now values



(a) Page load completion times at a spread of 0



(b) Page load completion times at a spread of 4

Figure 4.17. Range of page load completion times with variable depth and spread for Tor Browser and Fuzzyfox $g = 100ms$

test setup were made. Due to the major changes in routing, the load times we observed are far more variable than in the Firefox or Fuzzyfox case and show no significant trends on the whole. If we compare the range of page load times between Fuzzyfox ($g = 100ms$) and Tor Browser in figures 4.17a and 4.17b, we see that Tor Browser imposes a significantly *higher* overhead most of the time in both initial page load and in page load completion. Other spread levels show similar behavior. As in previous figures we show the 95th percentile load completion times but we additionally show the range from the minimum completion (`onload` fires) time as a shaded region.

Real world page loads

Table 4.3 shows a rough macro-benchmark of real-world page load times for Firefox, Fuzzyfox (various grains), and Tor Browser. In each case, the same Google search results page was loaded. These tests were manually performed and the reported page load time comes from the Firefox developer tools. Each load requested between 9 and 12 resources. The "force reload" column corresponds to a cache-less reload of the page, whereas the "reload" column indicates the load time with caching allowed. Minor differences between the reload and force reload results for a given browser are not statistically significant as we only have 10 samples.

Table 4.3. Average page load times for https://www.google.com/?gws_rd=ssl#q=test+search with 10 reloads and 10 force reloads (no caching) on Firefox, Fuzzyfox, and Tor Browser

Browser or Grain(ms)	**Reported load time(s)**	
	Reload	Force Reload
Firefox	0.82	0.86
0.5	0.84	0.79
1	0.85	0.85
5	0.94	0.94
10	1.03	1.04
50	2.09	1.71
100	2.86	2.60
Tor	3.78	7.18

While a larger study of more real-world pages would be valuable, such a study is larger in scope than this paper can cover. To perform such a measurement, we would need to individually determine a "load complete" point for each test page and re-instrument Fuzzyfox to enable measurements at these exact points. Google search results were chosen specifically because they do not continue to load resources indefinitely as many major websites do. (Ex: nytimes.com, youtube.com, etc.) We therefore leave a more detailed real-world page load time and user experience impact study to future work.

These metrics are incomplete, as they do not measure interactivity of the pages, which can suffer in the Fuzzyfox case more than in Tor Browser. We leave further analysis of various performance impacts to future work.

While higher g settings cause significant page load time increases, these overheads are acceptable to some privacy conscious users and developers as demonstrated by Tor Browser. We do not have metrics for the impact of using both Tor Browser and our Fuzzyfox patch set, but we expect the overheads to be additive in the worst case. One option for integration with Tor Browser specifically would be to tune the value of g based on the setting of the "security slider" [68].

In light of these metrics, a g setting of $g \leq 5$ms is likely tolerable for average use cases, while higher settings (up to and including $g = 100ms$) would likely be tolerated by users of Tor

Browser. Ideally the clock fuzzing and other features as appropriate will be deployed in Firefox, and can be configured for a higher g in Tor Browser. If a more complete version of Fermata is developed, it will be worthwhile to run user studies before deploying g settings.

4.7 Related work

Popek and Kline [69] were the first to observe that the presence of clocks opens covert channels. They suggested that virtual machines be presented only with virtual clocks, not "a real time measure." Lipner [60] responded that keeping virtual machines from correlating virtual time to real time is a "difficult problem," since time is "the one system-wide resource [...] that can be observed in at least a coarse way by every user and every program." Lipner suggested "randomizing the relation of virtual and real time" to add noise to the channel. Lipner also reported private communication from Saltzer that timing channels had been demonstrated in Multics by mid-1975.

Digital's VAX VMM Security Kernel project(initiated in 1981 and canceled in 1990 before its evaluation at the A1 level could be completed [49]) was the first system to attempt to randomize the relationship of virtual and real time. The VAX VMM Security Kernel team published three important papers describing their system. The first, by Karger et al. [48, 49], gave an overview of the system. The second, by Wray [87], presented a theory of time ("[w]e view the passage of time as being characterized by a sequence of events which can be distinguished one from another by an observer") and of timing channels and is the source for our view, in this paper, of timing channels as arising from the comparison of a reference clock with a modulated clock. Wray noted that a process that increments a variable in a loop can be used as a clock. The third, by Hu [44, 45], described the VAX VMM's fuzzy time system and is the inspiration for our paper. (A 2012 retrospective [59], though not the contemporaneous papers, reveals that the fuzzy time idea was developed in collaboration with the National Security Agency's Robert Morris.) We describe many of the details of the fuzzy time system elsewhere in the paper. The

1992 journal version [45] of Hu's paper gives a more complete security analysis than does the 1991 conference version [44]. In particular, it notes that fuzzy time would be defeated if the VM could devote a processor thread to incrementing a counter in memory shared with its other processor threads. This attack did not affect the Vax VMM Security Kernel, since it limited virtual machines to a single processor and did not support shared memory; it would apply to browsers if the proposed Shared Memory and Atomics specification [41] is implemented.

Several followup papers examined the security of fuzzy time. Trostle [82] observed that if scheduler time quanta coincide with upticks and if the scheduler employs a simple FIFO policy, then the scheduler can be used as a covert channel with 50 bps channel capacity. To send a bit, a high process either takes its entire time quantum or yields the processor; low processes try to send messages to each other in each time quantum. Which and how many messages arrived reveals the high process' bit. Gray showed attacks on fuzzy time that exploit bus contention [36] and calculated a channel capacity for shared buses under fuzzy time under the assumption (satisfied in the case of the VAX VMM Security Kernel) that a low receiver can immediately notify the high sender when it receives an uptick [35]. A later tech report combines both papers by Gray [37].

Martin et al. [61] translated fuzzy time to the microarchitectural setting, proposing and evaluating a new microarchitecture in which execution is divided into variable-length "epochs." The `rdtsc` instruction delays execution until the next epoch and returns a cycle count randomly chosen from the last epoch. Because their focus is microarchitectural timing channels, Martin et al. argue that other sources of time, such as interrupt delivery, are inherently too coarse grained to need fuzzing. Martin et al. observe that simply rounding `rdtsc` to some granularity would be susceptible to clock-edge effects.

The success of infrastructure-as-a-service cloud computing brought with it the risk of cross-VM side channels [72]. Aviram et al. [10] proposed to close timing channels in cloud computing by enforcing deterministic execution and experimented with compiling a Linux kernel and userland not to use high-resolution timers like `rdtsc`, observing a drop in throughput. Vattikonda et al. [84] showed that it is possible to virtualize `rdtsc` for Xen guests, reducing

its resolution (but allowing clock-edge attacks). Ford [31] proposed timing information flow control, or TIFC, "an extension of DIFC for reasoning about [...] the propagation of sensitive information into, out of, or within a software system via timing channels," and proposed two mechanisms for implementing TIFC: deterministic execution and "pacing queues," which are an extension of the VAX VMM Security Kernel's interrupt queue mechanism.

Li et al. [56, 57] describe StopWatch, a virtual machine manager designed to defeat timing side channel attacks. In StopWatch, clocks are virtualized to "a deterministic function of the VM's instructions executed so far"; multiple replicas of each VM are run in lockstep, and I/O timing for all of them is determined by the (virtual) time observed by the median replica.

Finally, Wu et Al. [88] present Deterland, a hypervisor that runs legacy operating systems deterministically. Deterland splits time into ticks and allows I/O only on tick boundaries. As in StopWatch, virtual time in Deterland is a function of the number of instructions executed.

4.8 Conclusions and future work

Restricting or removing timing side channels is a complex task. Simple degradation of available explicit clocks is an insufficient solution, allowing clock-edge techniques and implicit clocks to obtain additional timing information.

By drawing upon the lessons learned from trusted operating systems literature, we believe that browsers can be architected to mitigate all possible timing side channels. We propose Fermata as a design goal for such a verifiably resistant browser. Our Fuzzyfox patches to Firefox show that a Fermata-like design can intelligently make tradeoffs between performance and security, while not breaking the current interactions with JavaScript. Fuzzyfox empirically degrades clocks in a way that is not susceptible to clock-edge techniques, protecting timing information.

Fuzzyfox requires a number of engineering improvements before it is ready to deploy to users, but it has proved that the fuzzy time concept can be applied to browsers. Notably,

more experiments with setting channel bandwidth and exposing such settings to users need to be performed. Additionally, Fuzzyfox does not hook inbound network events, which a cooperating server could use to derive the duration of events in Fuzzyfox. Other interfaces (WebSockets, WebAudio, other media APIs) should be investigated for behavior that would break the Fuzzyfox design. We expect that with these changes Fuzzyfox could be adapted for use in projects like Tor Browser and protect real users against timing attacks.

Acknowledgements

We thank Kyle Huey, Patrick McManus, Eric Rescorla, and Martin Thomson at Mozilla for helpful discussions about this work, and for sharing their insights with us about Firefox internals. We are also grateful to Keaton Mowery and Mike Perry for helpful discussions, and to our anonymous reviewers and to David Wagner, our shepherd, for their detailed comments.

We additionally thank Nina Chen for assistance with editing and graph design.

This material is based upon work supported by the National Science Foundation under Grants No. 1228967 and 1514435, and by a gift from Mozilla.

Chapter 4, in part, is a reprint of the material as it appears in USENIX Security 2016. Kohlbrenner, David; Shacham, Hovav, 2016. The dissertation author was the primary investigator and the primary author of this paper.

Bibliography

[1] FDIV replacement program: Description of the flaw. Whitepaper: Onnline: http://www.intel.com/support/processors/pentium/sb/CS-013007.htm, July 2004. Fetched: Nov 12, 2014.

[2] 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, 2009.

[3] Onur Acıiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In Çetin Kaya Koç, editor, *Cryptographic Engineering*, chapter 18, pages 475–504. Springer-Verlag, 2009.

[4] Johan Agat. Transforming out timing leaks. In Thomas Reps, editor, *Proceedings of POPL 2000*, pages 40–53. ACM Press, January 2000.

[5] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofi'ene Tahar, and John Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. *The Computer Journal*, 53(4):465–488, May 2010.

[6] Mohammed I. Al-Saleh and Jedidiah R. Crandall. Application-level reconnaissance: Timing channel attacks against antivirus software. In Christopher Kruegel, editor, *Proceedings of LEET 2011*. USENIX, March 2011.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *Proceedings of USENIX Security 2016*, pages 53–70. USENIX, August 2016.

[8] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In Lujo Bauer and Vitaly Shmatikov, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2015*. IEEE Computer Society, May 2015.

[9] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In Angelos Keromytis and Vitaly Shmatikov, editors, *Proceedings of CCS 2010*, pages 297–307. ACM Press, October 2010.

[10] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In Adrian Perrig and Radu Sion, editors, *Proceedings of CCSW 2010*. ACM Press, October 2010.

[11] Manuel Barbosa, Andrew Moss, and Dan Page. Constructive and destructive use of compilers in elliptic curve cryptography. *J. Cryptology*, 22(2):259–81, April 2009.

[12] L. David Baron. Preventing attacks on a user's history through CSS :visited selectors, April 2010. Online: http://dbaron.org/mozilla/visited-privacy.

[13] Adam Barth, Collin Jackson, and John Mitchell. Robust defenses for cross-site request forgery. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 75–88. ACM Press, October 2008.

[14] Gilles Barthe, Gustavo Betarte, Juan Diego, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Moti Yung and Ninghui Li, editors, *Proceedings of CCS 2014*. ACM Press, November 2014.

[15] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, May 2006.

[16] Nicola Tuveri Billy Bob Brumley. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Proceedings of ESORICS 2011*, volume 6879 of *LNCS*, pages 355–71. Springer-Verlag, September 2011.

[17] Andrew Bortz, Dan Boneh, and Palash Nandy. Exposing private information by timing Web applications. In Peter Patel-Schneider and Prashant Shenoy, editors, *Proceedings of WWW 2007*, pages 621–28. ACM Press, May 2007.

[18] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–16, aug 2005.

[19] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In Moti Yung and Ninghui Li, editors, *Proceedings of CCS 2014*, pages 570–81. ACM Press, November 2014.

[20] Jerome Coonen, William Kahan, John Palmer, Tom Pittman, and David Stevenson. A proposed standard for binary floating point arthmetic. *SIGNUM Newsl.*, 14(si-2):4–12, October 1979.

[21] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In Andrew Myers and David Evans, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2009*, pages 45–60. IEEE Computer Society, May 2009.

[22] Bruce Dawson. Floating-point determinism. Online: http://randomascii.wordpress.com/2013/07/16/floating-point-determinism/, July 2013. Fetched: Nov 14, 2014.

[23] Bruce Dawson. Intel underestimates error bounds by 1.3 quintillion. Online: http://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/, October 2014. Fetched: Nov 14, 2014.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[25] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In David Evans and Giovanni Vigna, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2010*, pages 109–24. IEEE Computer Society, May 2010.

[26] Isaac Dooley and Laxmikant Kale. Quantifying the interference caused by subnormal floating-point values. In Matthew Sottile, Fabrizio Petrini, and Ronald Mraz, editors, *Proceedings of OSIHPA 2006*, September 2006. Online: http://osihpa.cs.utep.edu/2006/DooleySubnormal06.pdf.

[27] Cynthia Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54(1):86–95, January 2011.

[28] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, August 2014.

[29] Chris Evans. Cross-domain search timing. Online: https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html, December 2009.

[30] Edward W. Felten and Michael A. Schneider. Timing attacks on Web privacy. In Sushil Jajodia, editor, *Proceedings of CCS 2000*, pages 25–32. ACM Press, November 2000.

[31] Bryan Ford. Plugging side-channel leaks with timing information flow control. In Rodrigo Fonseca and Dave Maltz, editors, *Proceedings of HotCloud 2012*. USENIX, June 2012.

[32] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In Christopher Kruegel and Ninghui Li, editors, *Proceedings of CCS 2015*, pages 1394–1405. ACM Press, October 2015.

[33] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[34] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In Ari Juels, editor, *Proceedings of NDSS 2017*. Internet Society, February 2017.

[35] James W. Gray. On analyzing the bus-contention channel under fuzzy time. In Catherine Meadows, editor, *Proceedings of CSFW 1993*, pages 3–9. IEEE Computer Society, June 1993.

[36] James W. Gray. On introducing noise into the bus-contention channel. In Richard Kemmerer and John Rushby, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 1993*, pages 90–98. IEEE Computer Society, May 1993.

[37] James W. Gray. Countermeasures and tradeoffs for a class of covert timing channels. Technical Report HKUST-CS94-18, Hong Kong University of Science and Technology, 1994. Online: http://hdl.handle.net/1783.1/25.

[38] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Donghoon Lee and Seokhie Hong, editors, *Proceedings if ICISC 2009*, volume 5984 of *LNCS*, pages 176–92. Springer-Verlag, 2010.

[39] Gaël Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In Çetin K. Koç and Christof Paar, editors, *Proceedings of CHES 2000*, volume 1965 of *LNCS*, pages 293–301. Springer-Verlag, August 2000.

[40] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. Differential privacy under fire. In David Wagner, editor, *Proceedings of USENIX Security 2011*, pages 507–21. USENIX, August 2011.

[41] Lars T. Hansen. ECMAScript shared memory and atomics. Online: http://tc39.github.io/ecmascript_sharedmem/shmem.html, February 2016.

[42] Mark Harris. CUDA pro tip: Flush denormals with confidence. Online: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-flush-denormals-confidence/, January 2013. Fetched: Nov 13, 2014.

[43] Daniel Hedin and David Sands. Timing aware information flow security for a JavaCard-like bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):163–82, December 2005.

[44] Wei-Ming Hu. Reducing timing channels with fuzzy time. In Teresa F. Lunt and John McLean, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 1991*, pages 8–20. IEEE Computer Society, May 1991.

[45] Wei-Ming Hu. Reducing timing channels with fuzzy time. *J. Computer Security*, 1(3-4):233–54, 1992.

[46] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. I know where you've been: Geo-inference attacks via the browser cache. In Larry Koved and Matt Fredrikson, editors, *Proceedings of W2SP 2014*. IEEE Computer Society, May 2014.

[47] William Kahan. Why do we need a floating-point arithmetic standard? Whitepaper: Online: http://www.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf, February 1981. Fetched: Nov 12, 2014.

[48] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In Deborah M. Cooper and Teresa F. Lunt, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 1990*, pages 2–19. IEEE Computer Society, May 1990.

[49] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Software Engineering*, 17(11):1147–65, November 1991.

[50] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In Giovanni Vigna and Somesh Jha, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2011*, pages 413–28. IEEE Computer Society, May 2011.

[51] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In Tadayoshi Kohno, editor, *Proceedings of USENIX Security 2012*, pages 189–204. USENIX, August 2012.

[52] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Proceedings of Crypto 1996*, volume 1109 of *LNCS*, pages 104–13. Springer-Verlag, August 1996.

[53] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In Thorsten Holz and Stefan Savage, editors, *Proceedings of USENIX Security 2016*, pages 463–80. USENIX, August 2016.

[54] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Trans. Comput.*, 62(7):1276–88, July 2013.

[55] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: Timing attacks using CSS filters. In Virgil Gligor and Moti Yung, editors, *Proceedings of CCS 2013*, pages 1055–62. ACM Press, November 2013.

[56] Peng Li, Debin Gao, and Michael K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In George Candea, editor, *Proceedings of DSN 2013*. IEEE/IFIP, June 2013.

[57] Peng Li, Debin Gao, and Michael K. Reiter. StopWatch: A cloud architecture for timing channel mitigation. *ACM Trans. Info. & System Security*, 17(2), November 2014.

[58] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In Steve Blackburn, editor, *Proceedings of PLDI 2011*, pages 109–20. ACM Press, June 2011.

[59] Steve Lipner, Trent Jaeger, and Mary Ellen Zurko. Lessons from VAX/SVS for high-assurance VM systems. *IEEE Security & Privacy*, 10(6):26–35, Nov.–Dec. 2012.

[60] Steven B. Lipner. A comment on the confinement problem. *ACM SIGOPS Operating Systems Review*, 9(5):192–96, November 1975.

[61] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In Josep Torrellas, editor, *Proceedings of ISCA 2012*, pages 118–29. ACM Press, June 2012.

[62] Frank McSherry. Privacy integrated queries. In Alexandros Labrinidis, editor, *Proceedings of ACM SIGMOD 2009*. ACM Press, June 2009.

[63] Ilya Mironov. On significance of the least significant bits for differential privacy. In George Danezis and Virgil Gligor, editors, *Proceedings of CCS 2012*, pages 650–61. ACM Press, October 2012.

[64] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Proceedings of ICISC 2005*, volume 3935 of *LNCS*, pages 156–68. Springer-Verlag, February 2006.

[65] J Strother Moore, Thomas W. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5$_K$86 floating-point division program. *IEEE Trans. Computers*, 47(9):913–26, September 1998.

[66] Mozilla. Javascript concurrency model and event loop, 2016. Online: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion.

[67] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In Christopher Kruegel and Ninghui Li, editors, *Proceedings of CCS 2015*, pages 1406–18. ACM Press, October 2015.

[68] Mike Perry. Tor browser 4.5 is released, April 2015. Online: https://blog.torproject.org/blog/tor-browser-45-released.

[69] Gerald J Popek and Charles S Kline. Verifiable secure operating system software. In *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, pages 145–51. ACM, May 1974.

[70] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In Thorsten Holz and Stefan Savage, editors, *Proceedings of USENIX Security 2016*, pages 71–86. USENIX, August 2016.

[71] Rick Regan. Bug #53632: PHP hangs on numeric value 2.2250738585072011e-308. Online: https://bugs.php.net/bug.php?id=53632, December 2010. Fetched: Nov 12, 2014.

[72] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In Somesh Jha and Angelos Keromytis, editors, *Proceedings of CCS 2009*, pages 199–212. ACM Press, November 2009.

[73] Indrajit Roy, Srinath T.V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In Miguel Castro and Alex C. Snoeren, editors, *Proceedings of NSDI 2010*. USENIX, March 2010.

[74] David 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.*, 1:148–200, 1998.

[75] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[76] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In Collin Jackson, editor, *Proceedings of W2SP 2010*, May 2010.

[77] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In Paul F. Syverson, editor, *Proceedings of CSFW 2000*, CSFW '00, pages 200–14. IEEE Computer Society, July 2000.

[78] Mark Seaborn. Security: Chrome provides high-res timers which allow cache side channel attacks, 2015. Online: https://bugs.chromium.org/p/chromium/issues/detail?id=508166.

[79] Geoffrey Smith. A new type system for secure information flow. In Steve Schneider, editor, *Proceedings of CSFW 2001*, pages 115–25. IEEE Computer Society, June 2001.

[80] Paul Stone. Bug 711043 – (CVE-2013-1693) SVG filter timing attack. Online: https://bugzilla.mozilla.org/show_bug.cgi?id=711043, June 2011. Fetched: Nov 13, 2014.

[81] Paul Stone. Pixel perfect timing attacks with HTML5. Presented at Black Hat 2013, July 2013. Online: https://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf.

[82] Jonathan T. Trostle. Modelling a fuzzy time system. In Richard Kemmerer and John Rushby, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 1993*, pages 82–89. IEEE Computer Society, May 1993.

[83] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In Christopher Kruegel and Ninghui Li, editors, *Proceedings of CCS 2015*, pages 1382–93. ACM Press, August 2015.

[84] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen (short paper). In Tom Ristenpart and Christian Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, October 2011.

[85] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In Simon Foley, editor, *Proceedings of CSFW 1997*, pages 156–69. IEEE Computer Society, June 1997.

[86] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 223–238. IEEE, 2010.

[87] John C. Wray. An analysis of covert timing channels. In Teresa F. Lunt and John McLean, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 1991*, pages 2–7. IEEE Computer Society, May 1991.

[88] Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in Deterland. In Liuba Shrira, editor, *Proceedings of TRIOS 2015*. ACM Press, October 2015.

[89] Michal Zalewski. Rapid history extraction through non-destructive cache timing. Online: http://lcamtuf.coredump.cx/cachetime/, December 2011.

[90] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In Frank Tip, editor, *Proceedings of PLDI 2012*, pages 99–110. ACM Press, June 2012.

[91] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In George Danezis and Virgil Gligor, editors, *Proceedings of CCS 2012*, pages 305–16. ACM Press, October 2012.

[92] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In Moti Yung and Ninghui Li, editors, *Proceedings of CCS 2014*. ACM Press, November 2014.