On the effectiveness of mitigations against floating-point timing channels

David Kohlbrenner and Hovav Shacham, *UC San Diego*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/kohlbrenner

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

On the effectiveness of mitigations against floating-point timing channels

David Kohlbrenner* *UC San Diego*

Hovav Shacham† *UC San Diego*

Abstract

The duration of floating-point instructions is a known timing side channel that has been used to break Same-Origin Policy (SOP) privacy on Mozilla Firefox and the Fuzz differentially private database. Several defenses have been proposed to mitigate these attacks.

We present detailed benchmarking of floating point performance for various operations based on operand values. We identify families of values that induce slow and fast paths beyond the classes (normal, subnormal, etc.) considered in previous work, and note that different processors exhibit different timing behavior.

We evaluate the efficacy of the defenses deployed (or not) in Web browsers to floating point side channel attacks on SVG filters. We find that Google Chrome, Mozilla Firefox, and Apple's Safari have insufficiently addressed the floating-point side channel, and we present attacks for each that extract pixel data cross-origin on most platforms.

We evaluate the vector-operation based defensive mechanism proposed at USENIX Security 2016 by Rane, Lin and Tiwari and find that it only reduces, not eliminates, the floating-point side channel signal.

Together, these measurements and attacks cause us to conclude that floating point is simply too variable to use in a timing security sensitive context.

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. [2] 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 [15] 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.

*dkohlbre@cs.ucsd.edu
†hovav@cs.ucsd.edu

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 1: IEEE-754 Format type ranges (Reproduced with permission from [2])

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.

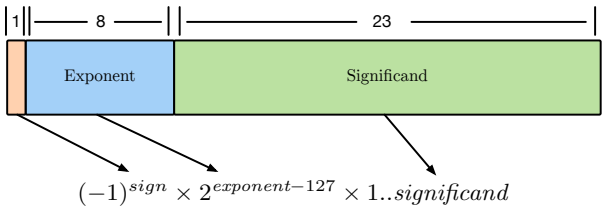

Figure 2: IEEE-754 single precision float

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 Background

Many floating point instructions are known to exhibit performance differences based on the operands. Andrysco et al. [2] 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 [16] and Kotcher et al. [13].

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 Figure 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 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 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 figure 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. [2] 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 Figure 8 for a list of non-normal IEEE-754 value types. In this paper we present additional benchmarks that demonstrate that (smaller) timing differences arise from more than just subnormal operands. Section 3 describes our benchmarking results.

2.2 SVG floating point timing attacks

Andrysco et al. [2] 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 3 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

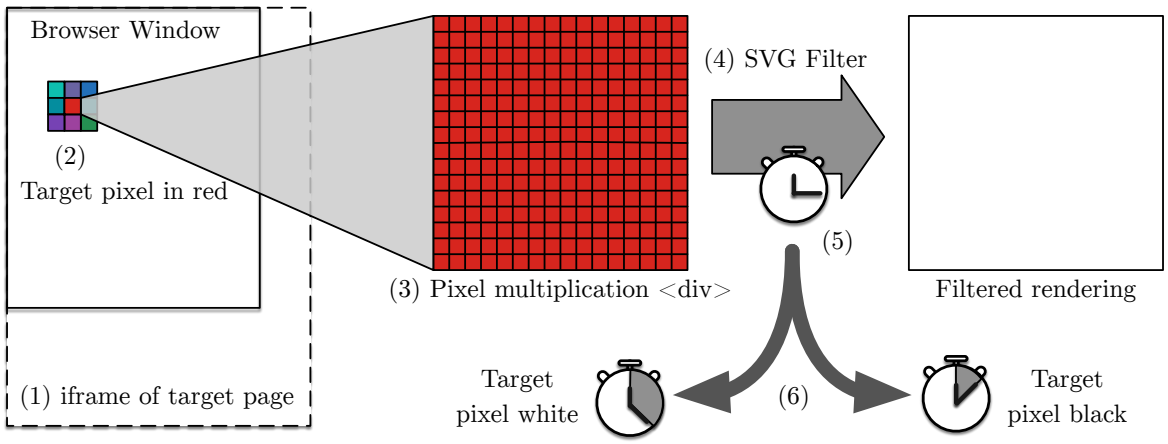

Figure 3: Cross-Origin SVG Filter Pixel Stealing Attack in Firefox, reproduced from [2] with permission

	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 4: Multiplication timing for single precision floats on Intel i5-4460

on the `<iframe>` using the `scrollTop` and `scrollLeft` properties.

3. The target pixel is duplicated into a larger container `<div>` using the `-moz-element` 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.

3 New floating point timing observations

Andryso et al. [2] 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.

Figure 9 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 4, 5, 6, and 7 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 type of operation (add, subtract, divide, etc) as the operation, and the specific combination of operands

Dividend	Divisor								
	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 5: Division timing for single precision floats on Intel i5-4460

	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 6: Multiplication timing for double precision floats on Intel i5-4460

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. [2] 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.

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. [2] to stop functioning. Many of these implementations no longer used floating point math, instead using their own fixed point arithmetic.

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.

Dividend	Divisor 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 7: Division timing for double precision floats on Intel i5-4460

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

Figure 8: IEEE-754 Special Value Encoding (Reproduced with permission from [2])

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

Figure 9: 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

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 [2].

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.

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. 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.

   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 a subnormal `surfaceScale` and an attached `fePointLight` as the timing vulnerable filter.

5. Steps 5 and 6 are the same as section 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 4), 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.

5 Safari

At the time of writing this paper, Safari has not implemented any defensive mechanisms that hamper the SVG timing attack presented in [2]. 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 have disclosed the attack to Apple, and discussed options for entirely disabling cross-origin SVG filtering. Apple is working to address the issue.

We have removed details on the needed technical modifications to the attack for Safari as a patch is not yet available for all users. A full description of the modifications required for the Safari variant will be released upon a patch being available.

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.

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 and figure 3.

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 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

```
<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 10: HTML and style design for the pixel multiplying structure used in our attacks on Safari and Chrome

	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 11: Division timing for double precision floats on Intel i5-4460+FTZ/DAZ

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 `feConvolveMatrix` 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 figures 11 and 9. 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 fix is in progress.

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

7 Revisiting the effectiveness of Escort

Escort [15] proposes defenses against multiple types of timing side channels, notably a defense using SIMD vec-

```
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

tor operations to protect against the floating point attack presented by Andrysco et al in [2].

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.

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

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

Figure 12: 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	S	S
Mul	S	–
Div	S	–
Sqrt	S	–
Double Precision		
Add/Sub	S	S
Mul	S	–
Div	S	–
Sqrt	S	–

Figure 13: 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

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 [4]) 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 be-

Dividend	Divisor 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 14: Division timing for double precision floats on Intel i5-4460+Escort

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 15: Division timing for double precision floats on Intel i5-4460 macro-test

fore 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`.

However, operations that do not receive subnormals can still exhibit timing differences. As seen in figure 7 and summarized in figure 9 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 end-to-end runtime of toy programs compiled under Escort.

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 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.

7.2.1 Results on Intel i5-4460

Our results for the Intel i5-4460 CPU roughly correspond to the variations presented in [15] (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 14. Figure 12 summarizes the timing variations we observed.

For division, it appears that the numerator has no im-

pact 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 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 figure 9)

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.

7.2.2 AMD Phenom II X2 550

Figure 13 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.

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 15 shows the crosstable for these results. Note that cells are colorized if they differ by 2% rather than 1 cycle.

7.4 `libdrag` modified Firefox

We modified a build of Firefox 25 in consultation with Rane et al [15] 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 [2]. 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 7.2.2 we believe that multiple SVG filters would be timing side channel vulnerable under Escort on that CPU.

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 7.2.1 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.

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.

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

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 16: Division timing for single precision floats on Nvidia GeForce GT 430

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.

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 16. All other results (add, sub, mul) were constant time regardless of the inputs..

As figure 16 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.

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 [6]. Felten and Schneider's history sniffing attack was later refined by Zalewski [18]. 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. [10].

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 [3], who showed they could be used to learn the number of items in a user's shopping cart. Evans [5] and, later, Gelernter and Herzberg [7], 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 [17] 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, 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 [14], and used them to learn gross system activity. Recently, Gras et al. [8] 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 [12] proposed a browser architecture that degrades the clocks available to JavaScript in a more principled way, drawing on ideas from the "fuzzy time" mitigation [9] in the VAX VMM Security Kernel [11].

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 [16] and, independently, Kotcher et al. [13], showed that such pixel-stealing attacks are feasible; the filters they exploited had pixel-dependent branches. Andrysco et al. [2] 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 [15] 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.

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 [1].

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 disallow 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.

References

[1] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 53–70.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proceedings of IEEE Security and Privacy ("Oakland") 2015*, L. Bauer and V. Shmatikov, Eds. IEEE Computer Society, May 2015.

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

[4] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[5] C. Evans, "Cross-domain search timing," Online: https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html, Dec. 2009.

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

[7] N. Gelernter and A. Herzberg, "Cross-site search attacks," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Oct. 2015, pp. 1394–1405.

[8] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proceedings of NDSS 2017*, A. Juels, Ed. Internet Society, Feb. 2017.

[9] W.-M. Hu, "Reducing timing channels with fuzzy time," *J. Computer Security*, vol. 1, no. 3-4, pp. 233–54, 1992.

[10] Y. Jia, X. Dong, Z. Liang, and P. Saxena, "I know where you've been: Geo-inference attacks via the browser cache," in *Proceedings of W2SP 2014*, L. Koved and M. Fredrikson, Eds. IEEE Computer Society, May 2014.

[11] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A retrospective on the VAX VMM security kernel," *IEEE Trans. Software Engineering*, vol. 17, no. 11, pp. 1147–65, Nov. 1991.

[12] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 463–80.

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

[14] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Oct. 2015, pp. 1406–18.

[15] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast floating-point operations on x86 processors," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 71–86.

[16] P. Stone, "Pixel perfect timing attacks with HTML5," Presented at Black Hat 2013, Jul. 2013, online: https://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf.

[17] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Aug. 2015, pp. 1382–93.

[18] M. Zalewski, "Rapid history extraction through non-destructive cache timing," Online: http://lcamtuf.coredump.cx/cachetime/, Dec. 2011.

Notes

[1] `https://skia.org/`