Trusted Browsers for Uncertain Times

David Kohlbrenner and Hovav Shacham, ***University of California, San Diego***

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kohlbrenner

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Trusted browsers for uncertain times

David Kohlbrenner* *UC San Diego*
Hovav Shacham† *UC San Diego*

Abstract

JavaScript in one origin can use timing channels in browsers to learn sensitive information about a user's interaction with other origins, violating the browser's compartmentalization guarantees. Browser vendors have attempted to close timing channels by trying to rewrite sensitive code to run in constant time and by reducing the resolution of reference clocks.

We argue that these ad-hoc efforts are unlikely to succeed. We show techniques that increase the effective resolution of degraded clocks by two orders of magnitude, and we present and evaluate multiple, new implicit clocks: techniques by which JavaScript can time events without consulting an explicit clock at all.

We show how "fuzzy time" ideas in the trusted operating systems literature can be adapted to building trusted browsers, degrading all clocks and reducing the bandwidth of all timing channels. We describe the design of a next-generation browser, called Fermata, in which all timing sources are completely mediated. As a proof of feasibility, we present Fuzzyfox, a fork of the Firefox browser that implements many of the Fermata principles within the constraints of today's browser architecture. We show that Fuzzyfox achieves sufficient compatibility and performance for deployment today by privacy-sensitive users.

In summary:

- We show how an attacker can measure durations in web browsers without querying an explicit clock.
- We show how the concepts of "fuzzy time" can apply to web browsers to mitigate all clocks.
- We present a prototype demonstrating the impact of some of these concepts.

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 [24]. 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 [19].

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 1), reducing timer resolution to 5 µs to defeat Oren et al.'s cache timing attack [19].

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 [3]. Indeed, the Mozilla fix for `feMorphology` did not consider the possibility that

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

floating-point instructions execute faster or slower depending on their inputs, allowing pixel-stealing attacks even in supposedly "constant-time" code [1].

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 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 [28], and our fuzzy mitigation strategy technique is directly inspired by Hu [10] — both papers resulting from the VAX VMM Security Kernel project, which targeted an A1 rating [12].

In this paper, we show that "fuzzy time" ideas due to Hu [10] 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 1: Google Chrome `performance.now` rounding code

```
// 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 2: Clock-edge fine-grained timing attack in JavaScript

2 Clock-edge attack

Web browser vendors have attempted to mitigate timing side channel attacks like [19] 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 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. Experimen-

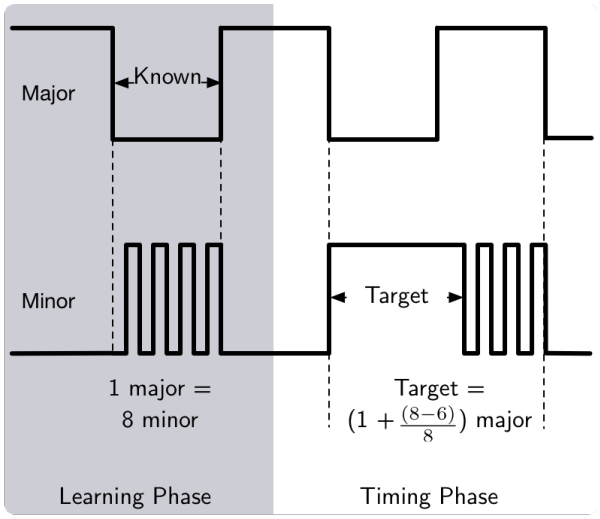

Figure 3: Clock-edge learning and timing

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

Table 1: Results for running the clock-edge fine-grained timing attack against various grain settings. Averages for 100 runs shown.

tally, 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 2 and 3 show how a page might execute this technique and a visual representation of the process.

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 measurement 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 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 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.

As table 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.

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.

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.

3.1.1 Exiting targets: privacy breaches with `requestAnimationFrame`

Previous work [1] [24] 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 [1] 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.

3.1.2 Exitless targets: cache timing attacks from JavaScript

Conversely, there are *exitless* `target`s, such as Oren et al's [19] 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.

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.

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

Table 2: Implicit clock type in different browsers
L Exitless , X Exiting , — Not implemented, + Buggy

We observe that just as with exiting and exitless `target`s, 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 2 for how the following clocks manifest in Chrome 48 (stable), Firefox[3], and Safari 9.0.3.

3.2.1 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[18] 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 [23], this can be used as a very precise clock in real attacks.

3.2.2 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 `setTimeout`s 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.

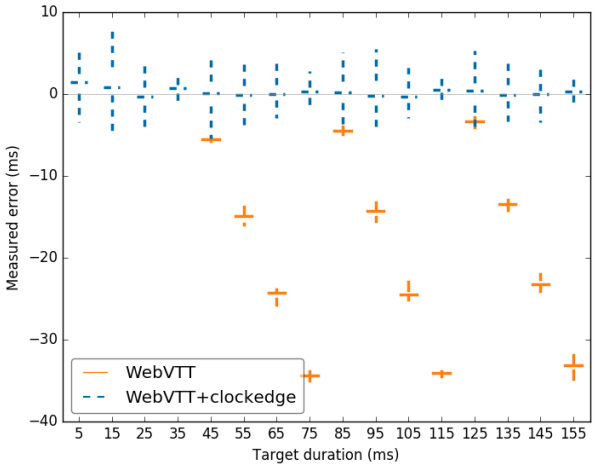

Figure 4: WebVTT error measurements with and without clock-edge technique

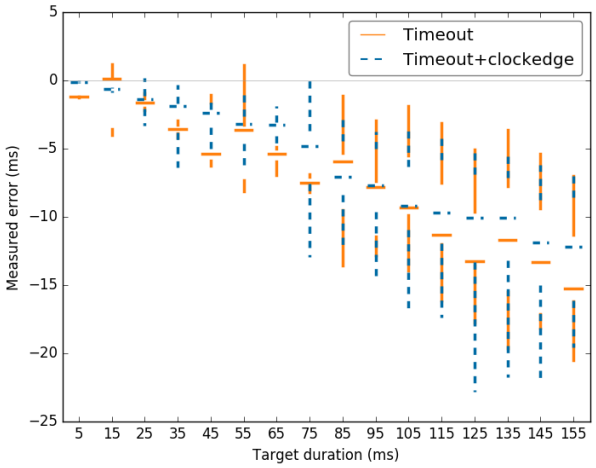

Figure 5: `setTimeout` error measurements with and without clock-edge technique

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.

3.3 Performance of implicit clocks

The granularity, precision, and accuracy of implicit clocks varies widely by technique. We observe that

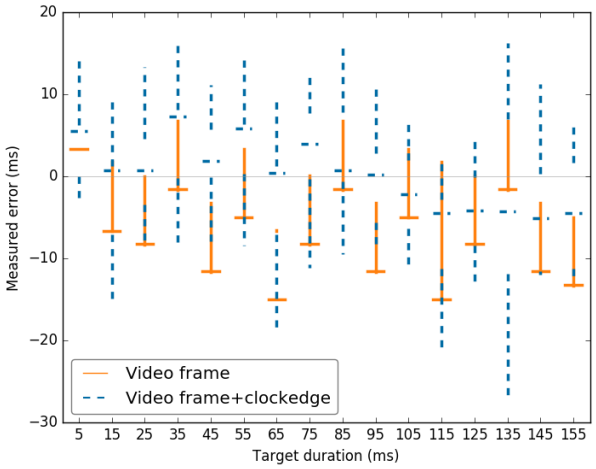

Figure 6: Video frame error measurements with and without clock-edge technique

most implicit clocks can be improved with the clock-edge technique from section 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.

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

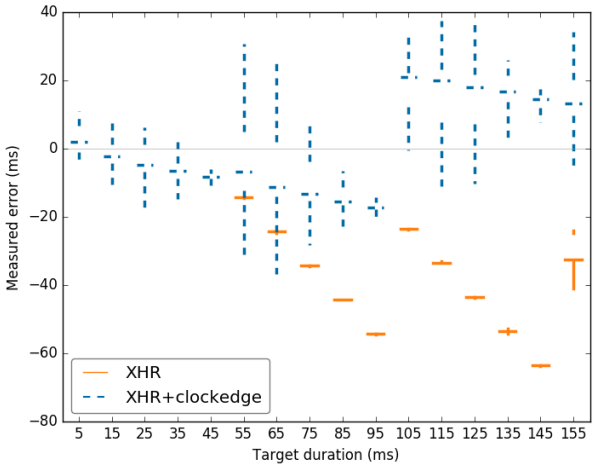

Figure 7: Throttled XMLHTTPRequest error measurements with and without clock-edge technique

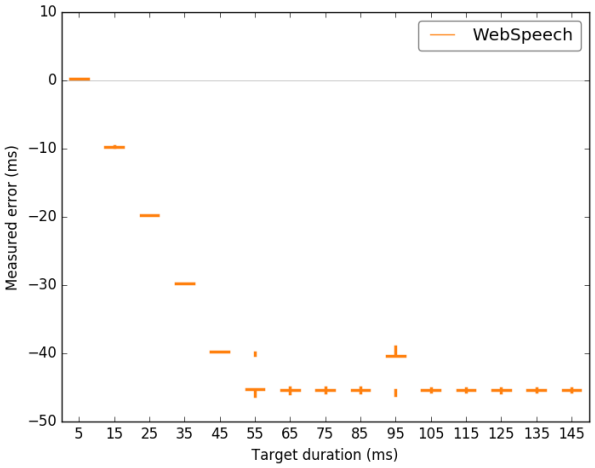

Figure 9: WebSpeech error measurements without clock-edge technique

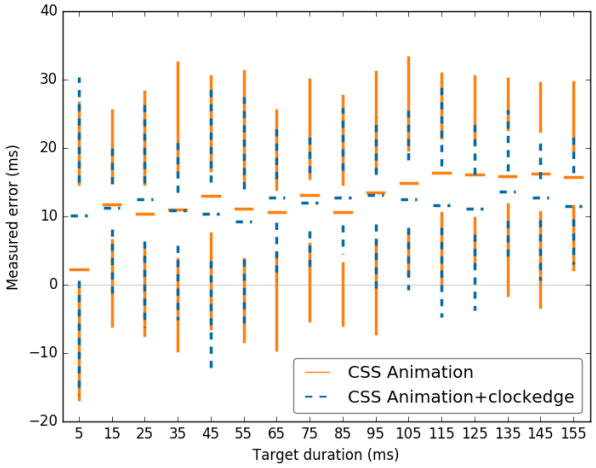

Figure 8: CSS animation error measurements with and without clock-edge technique

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 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 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, 5, 6, 7, 8, and 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 3.2.1. 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 be.

Figures 11 and 10 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 Fermata

In this section we describe *Fermata*, a theoretical browser design that provably degrades all attacker visible clocks. Sections 5 and 6 describe our prototype implementation, Fuzzyfox, and an evaluation. Fermata is

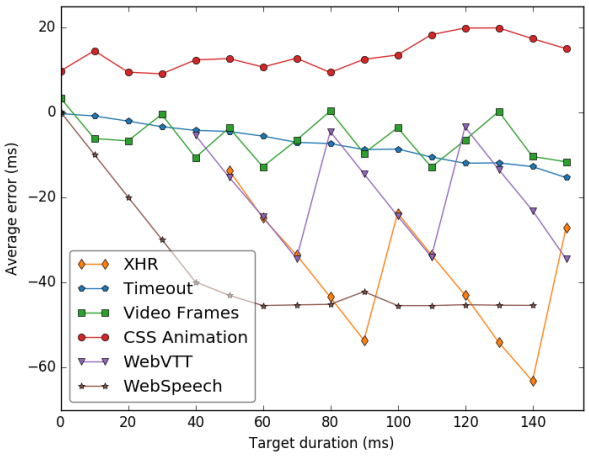

Figure 10: Average error for all clock techniques without clock-edge

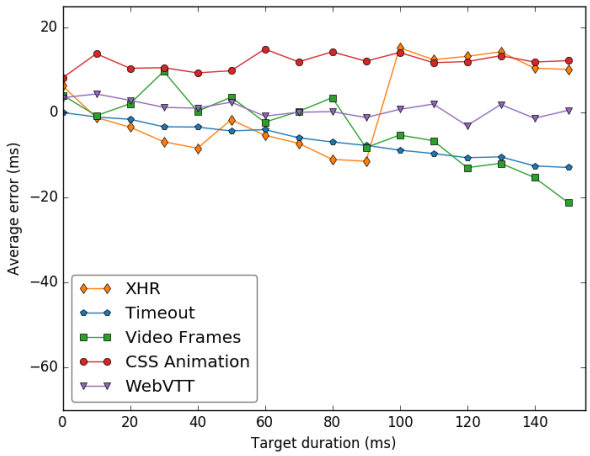

Figure 11: Average error for all clock techniques with clock-edge where available

an adaptation of the *fuzzy time* operating systems concept detailed in [10] 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 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 3 as well as any other such clock unknown to us.

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 [3]. 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" methodology 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 [1] [24] 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 [27]. 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 history sniffing impractical. As an example, [27] 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 [24] 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 [1]) can still leak data. If the attack relies on a small, inherent microarchitecture timing, such as Oren et al's [19] 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 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.

4.2 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.

4.3 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 [10].

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 Fermata guarantees

We believe that the analysis of Hu's fuzzytime by Gray in [5] 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 [5], 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 [6]. The general case problem of covert channel capacity under fuzzy time appears to be intractable but can be bounded under specific circumstances.

4.4.1 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.

4.5 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.

4.6 Degrading explicit clocks

Explicit clocks (ex: `performance.now`, `Date`, etc.) are degraded to some granularity g and update unpredictably. As in Hu [10], 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 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 2.

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 [10]. Downticks cause outbound queued events to be flushed, and upticks cause inbound events to be delivered.

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.

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 2 and 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.

5.1 Why Fuzzyfox?

We built Fuzzyfox for three reasons:

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

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 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.

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.

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 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.

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 6.2.3 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.

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 3 between Firefox and Fuzzyfox. We set the explicit time granularity (g) to 100ms and the average `PauseTask` interval (t_a) to 50ms for these tests. We chose $g = 100ms$ because a large g value most clearly illustrates the difference between Fuzzyfox and Firefox. See section 6.3 for an evaluation of the impact of high g values on performance.

The following figures show scatter plots for several

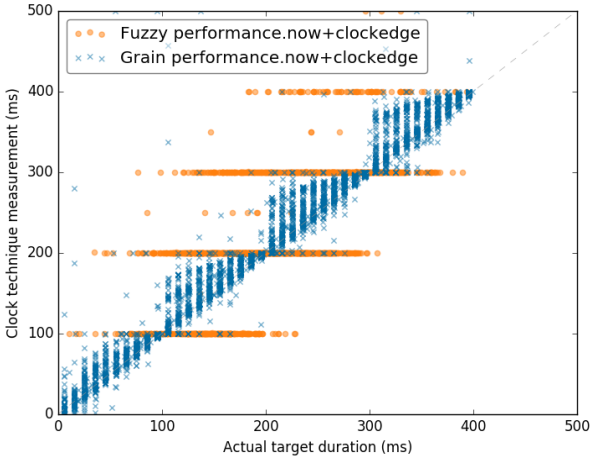

Figure 12: `performance.now` measurements with clock-edge on Fuzzyfox (exiting) and Firefox (exitless, 100ms grain)

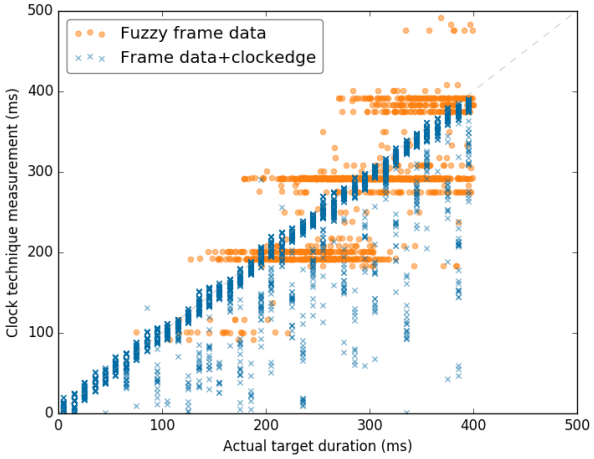

Figure 13: Frame data clock measurements on Firefox and Fuzzyfox

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 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.

6.2.1 `performance.now`

Since time no longer moves forward during JavaScript execution, `performance.now` is now an exiting

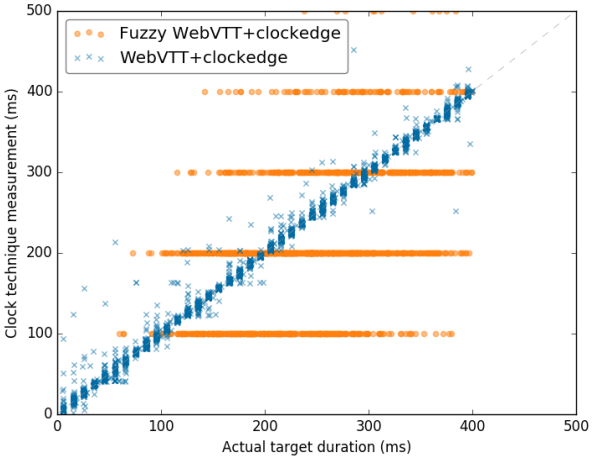

Figure 14: WebVTT clock measurements on Firefox and Fuzzyfox

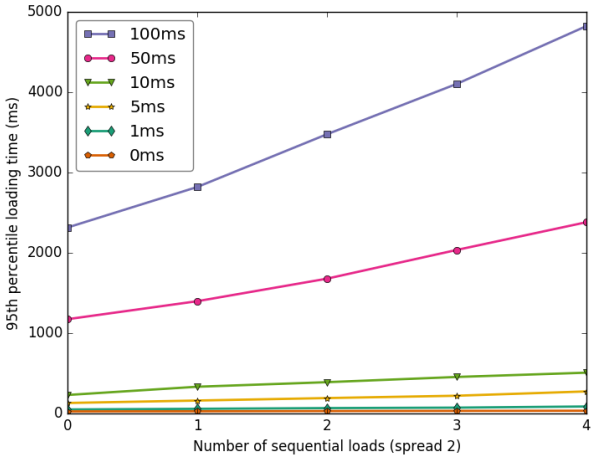

Figure 15: Page load times with variable depth for all Fuzzyfox configurations at a spread of 2

clock. Figure 12 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.

6.2.2 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 13 shows the exiting frame data clock on Fuzzyfox and Firefox.

6.2.3 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.

6.2.4 `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.

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

Figure 16: Iterative page load JavaScript

6.2.5 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.

6.2.6 XMLHTTPRequests

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

6.2.7 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 14 shows the results for the same WebVTT clock techniques as described in section 3.2.2 on both Fuzzyfox and Firefox. Note that the clockedge code provided no benefits to the Fuzzyfox case.

6.3 Performance

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.

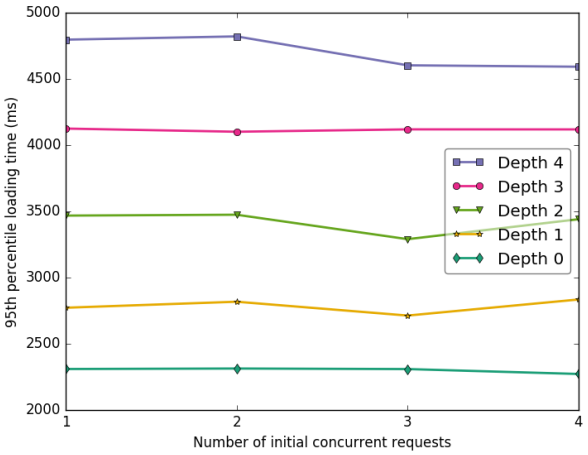

Figure 17: Page load times with variable spread and depth for $g = 100ms$

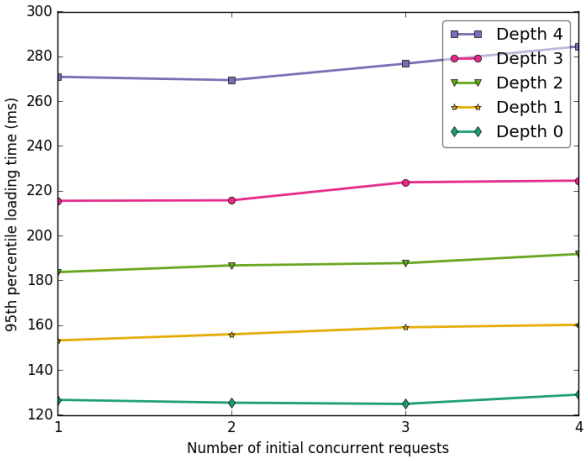

Figure 18: Page load times with variable spread and depth for $g = 5ms$

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 16. 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 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 15 and 17 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 17) results in almost no change to load times. All other browser configurations (see figure 18 for $g = 5ms$) had nearly identical results, with differing 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 15 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.

6.3.1 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 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 19 and 20, 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.

6.3.2 Real world page loads

Table 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

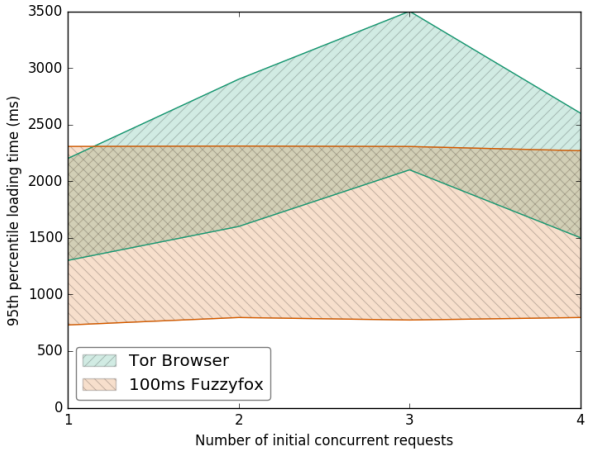

Figure 19: Range of page load completion times with variable depth at a spread of 0 for Tor Browser and Fuzzyfox $g = 100ms$

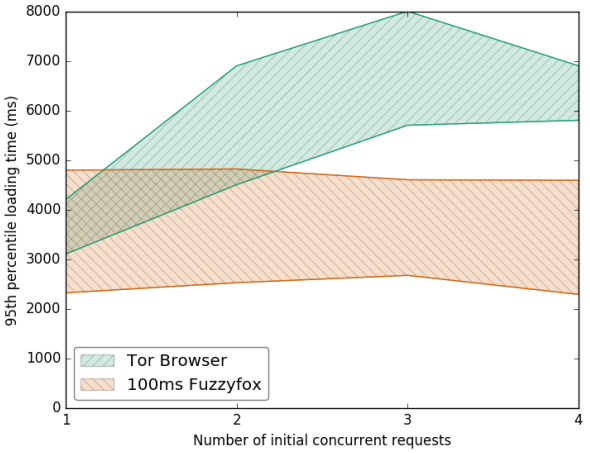

Figure 20: Range of page load completion times with variable depth at a spread of 4 for Tor Browser and Fuzzyfox $g = 100ms$

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

Table 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

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.

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" [20].

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.

7 Related work

Popek and Kline [21] 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 [16] 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 [12]) 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. [11, 12], gave an overview of the system. The second, by Wray [28], 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 [9, 10], described the VAX VMM's fuzzy time system and is the inspiration for our paper. (A 2012 retrospective [15], 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 [9] of Hu's paper gives a more complete security analysis than does the 1991 conference version [10]. 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 [8] is implemented.

Several followup papers examined the security of fuzzy time. Trostle [25] 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 [7] 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 [5]. A later tech report combines both papers by Gray [6].

Martin et al. [17] 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 [22]. Aviram et al. [2] 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. [26] showed that it is possible to virtualize `rdtsc` for Xen guests, reducing its resolution (but allowing clock-edge attacks). Ford [4] 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. [13, 14] 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. [29] 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.

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.

References

[1] 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.

[2] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating timing channels in compute clouds," in *Proceedings of CCSW 2010*, A. Perrig and R. Sion, Eds. ACM Press, Oct. 2010.

[3] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on seL4," in *Proceedings of CCS 2014*, M. Yung and N. Li, Eds. ACM Press, Nov. 2014, pp. 570–81.

[4] B. Ford, "Plugging side-channel leaks with timing information flow control," in *Proceedings of HotCloud 2012*, R. Fonseca and D. Maltz, Eds. USENIX, Jun. 2012.

[5] J. W. Gray, "On analyzing the bus-contention channel under fuzzy time," in *Proceedings of CSFW 1993*, C. Meadows, Ed. IEEE Computer Society, Jun. 1993, pp. 3–9.

[6] ——, "Countermeasures and tradeoffs for a class of covert timing channels," Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS94-18, 1994, online: http://hdl.handle.net/1783.1/25.

[7] ——, "On introducing noise into the bus-contention channel," in *Proceedings of IEEE Security and Privacy ("Oakland") 1993*, R. Kemmerer and J. Rushby, Eds. IEEE Computer Society, May 1993, pp. 90–98.

[8] L. T. Hansen, "ECMAScript shared memory and atomics," Online: http://tc39.github.io/ecmascript_sharedmem/shmem.html, Feb. 2016.

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

[10] ——, "Reducing timing channels with fuzzy time," in *Proceedings of IEEE Security and Privacy ("Oakland") 1991*, T. F. Lunt and J. McLean, Eds. IEEE Computer Society, May 1991, pp. 8–20.

[11] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A VMM security kernel for the VAX architecture," in *Proceedings of IEEE Security and Privacy ("Oakland") 1990*, D. M. Cooper and T. F. Lunt, Eds. IEEE Computer Society, May 1990, pp. 2–19.

[12] ——, "A retrospective on the VAX VMM security kernel," *IEEE Trans. Software Engineering*, vol. 17, no. 11, pp. 1147–65, Nov. 1991.

[13] P. Li, D. Gao, and M. K. Reiter, "Mitigating access-driven timing channels in clouds using StopWatch," in *Proceedings of DSN 2013*, G. Candea, Ed. IEEE/IFIP, Jun. 2013.

[14] ——, "StopWatch: A cloud architecture for timing channel mitigation," *ACM Trans. Info. & System Security*, vol. 17, no. 2, Nov. 2014.

[15] S. Lipner, T. Jaeger, and M. E. Zurko, "Lessons from VAX/SVS for high-assurance VM systems," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 26–35, Nov.–Dec. 2012.

[16] S. B. Lipner, "A comment on the confinement problem," *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5, pp. 192–96, Nov. 1975.

[17] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *Proceedings of ISCA 2012*, J. Torrellas, Ed. ACM Press, Jun. 2012, pp. 118–29.

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

[19] 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.

[20] M. Perry, "Tor browser 4.5 is released," Apr. 2015, online: https://blog.torproject.org/blog/tor-browser-45-released.

[21] G. J. Popek and C. S. Kline, "Verifiable secure operating system software," in *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*. ACM, May 1974, pp. 145–51.

[22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds," in *Proceedings of CCS 2009*, S. Jha and A. Keromytis, Eds. ACM Press, Nov. 2009, pp. 199–212.

[23] M. 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.

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

[25] J. T. Trostle, "Modelling a fuzzy time system," in *Proceedings of IEEE Security and Privacy ("Oakland") 1993*, R. Kemmerer and J. Rushby, Eds. IEEE Computer Society, May 1993, pp. 82–89.

[26] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen (short paper)," in *Proceedings of CCSW 2011*, T. Ristenpart and C. Cachin, Eds. ACM Press, Oct. 2011.

[27] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 223–238.

[28] J. C. Wray, "An analysis of covert timing channels," in *Proceedings of IEEE Security and Privacy ("Oakland") 1991*, T. F. Lunt and J. McLean, Eds. IEEE Computer Society, May 1991, pp. 2–7.

[29] W. Wu, E. Zhai, D. I. Wolinsky, B. Ford, L. Gu, and D. Jackowitz, "Warding off timing attacks in Deterland," in *Proceedings of TRIOS 2015*, L. Shrira, Ed. ACM Press, Oct. 2015.

Notes

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=711043`

[2] `https://trac.torproject.org/projects/tor/ticket/1517`

[3] commit 0ec3174fe63d8139f842ce9eb6639349759ff4e5

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

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

[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