Pixel Thief: Exploiting SVG Filter Leakage in Firefox and Chrome

Sioli O'Connell[1*], Lishay Aben Sour[2*], Ron Magen[2*], Daniel Genkin[3], Yossi Oren[2,4], Hovav Shacham[5], and Yuval Yarom[6]

[1]The University of Adelaide
[2]Ben Gurion University of the Negev
[3]Georgia Institute of Technology
[4]Intel Corporation
[5]UT Austin
[6]Ruhr University Bochum

Abstract

Web privacy is challenged by pixel-stealing attacks, which allow attackers to extract content from embedded `iframe`s and to detect visited links. To protect against multiple pixel-stealing attacks that exploited timing variations in SVG filters, browser vendors repeatedly adapted their implementations to eliminate timing variations. In this work we demonstrate that past efforts are still not sufficient.

We show how web-based attackers can mount cache-based side-channel attacks to monitor data-dependent memory accesses in filter rendering functions. We identify conditions under which browsers elect the non-default CPU implementation of SVG filters, and develop techniques for achieving access to the high-resolution timers required for cache attacks. We then develop efficient techniques to use the pixel-stealing attack for text recovery from embedded pages and to achieve high-speed history sniffing. To the best of our knowledge, our attack is the first to leak multiple bits per screen refresh, achieving an overall rate of 267 bits per second.

1 Introduction

In recent decades, the Internet has grown from a research-oriented network aimed for specialists into a communication network that encompasses all aspects of modern life. In typical use, a web browser accesses multiple websites, often concurrently. Each of these websites may process private, personal, and even sensitive information about their users. Even the fact that a user has merely accessed a specific website may reveal personal information about the user's beliefs, health, or social connections. Consequently, preventing information leaks within the browser is of paramount importance.

One of the main tools for preventing cross-site information leaks is the *same-origin policy (SOP)*, which prevents code of one website from accessing resources on other websites. This property holds even if the victim website and the attacker website appear together on the user's screen, through the use of the HTML `iframe` element.

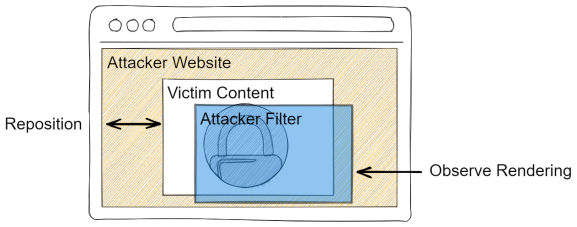


Figure 1: Overview of a pixel-stealing attack

Beyond code access, information can leak from websites through the way that they are displayed on the user's screen. This includes, for example, the rendering of inline `iframe`s, which embed content from one website into another website, and of links, which are rendered differently if the user has visited the linked site. To protect against such attacks, web browsers do not allow websites access even to their own rendered image. Browsers do, however, allow limited manipulation of the displayed contents through the use of *SVG filters*, which perform image transformations, such as recoloring, resizing, blurring, and more.

Past works have shown how to exploit minute data-dependent timing variations in SVG filters for pixel-stealing attacks [4, 36, 63]. Figure 1 shows the general structure of such filter-based pixel-stealing attacks. Here, the attacker lures the user to a malicious website that displays sensitive contents, e.g. in an `iframe`, applying SVG filters to the contents. By measuring the time it takes to render the page, the attacker can recover information about one of the pixels in the `iframe`. Changing the relative positions of the filters and the image, the attacker can target different pixels, eventually reconstructing the displayed contents.

A fundamental limitation of these attacks is their dependence on measuring the time it takes the browser to render a frame. This limits the leak to at most one bit per refresh, or no

* Equal contribution first author

more than 60 bits per second. Moreover, browser vendors are aware of the risks of timing-based pixel-stealing attacks, and have modified their filter rendering code to remove observable, content-dependent timing differences in filter execution time [12, 13, 14, 46]. Recent works have demonstrated the feasibility of exploiting CPU and GPU frequency scaling for pixel-stealing attacks [64, 68]. These attacks, however, have even lower leakage rates. Thus, we ask: *Are high-capacity pixel-stealing attacks on modern browsers feasible?*

1.1 Our Contribution

In this work we abuse SVG filters to send information through a cache-based side channel. We identify content-dependent memory access patterns in the CPU implementation of the `feComponentTransfer` filter. Because browsers default to GPU implementations, we first identify conditions under which the browsers elect to use this CPU version. We find that Chrome lacks support for many system configurations, forcing CPU execution on unsupported systems. For Firefox, we identify sequences of filters that force CPU execution even on otherwise supported systems.

To exploit the filter, we develop techniques to amplify the signal, allowing capture by a realistic side-channel attacker. We then design a communication protocol that allows the attacker to identify the cache location that the filter uses, and to transmit data through the channel. Our protocol allows us to overcome the limit of one pixel per frame, which affects all prior pixel-stealing attacks.

As an additional contribution, we show how to overcome cross-origin-isolation policies in web browsers. These policies allow websites to either use high-resolution timers or to embed contents of third-party websites, but not both. We note that the attacker can use two websites, one to embed the victim contents and the other to perform the cache attack. The attacker can therefore use high-resolution timers in the browser, without the need to resort to alternative timers [34, 59] or amplification techniques [29, 31, 54, 55].

We present two attacks that demonstrate the effectiveness of our technique. The first attack uses pixel stealing to observe text in the victim page. For that, we develop techniques for identifying a small number of regions that allow us to easily distinguish between letters, and show how to use these regions to accurately identify the displayed letters. We further show that the technique can be used as an end-to-end attack that leaks the Wikipedia identity of the victim.

Our second attack exploits pixel stealing for a history-sniffing attack. We first show a straightforward approach that leaks a limited number of links at a time. We then adapt the technique of Stone [63] for detecting whether any site in a given list has been visited. Finally, we devise an adaptive approach that further increases the speed of history sniffing under the assumption that the number of visited sites is substantially smaller than the total number of sites queried.

In summary, our contributions are as follows.

- We develop a simple approach to overcome cross-origin policies, allowing us to have access to high-resolution timers, while embedding third-party contents (Section 4).
- We show how to exploit the content-dependent memory access patterns of the `feComponentTransfer` SVG filter for pixel stealing. We design a transmitter (Section 5) and a protocol (Section 6), demonstrating the first faster-than-refresh-rate pixel-stealing attack.
- We show how to use our pixel-stealing attack to recover text from the victim page (Section 7).
- We demonstrate a fast history-sniffing attack on a modern browser (Section 8).

2 Background

2.1 Cache Attacks

Caches. To reduce the average latency of memory accesses, modern processors exploit program locality by introducing caches that store recently accessed memory locations. Modern x86 processors provide several levels of caches, where each core has private L1 and L2 caches, and all cores share access to a common last level cache (LLC). The caches in x86 processors are *set associative*, that is, the caches are organized as a collection of sets, each containing multiple *ways*, and each way in turn can store a single fixed-sized block of memory known as a *cache line*.

Cache Timing Attacks. Because the state of the cache depends on prior computation and, at the same time, affects code execution time, sharing caches can lead to information leaks [19], leading to a large number of attacks [21, 23, 40, 50, 51, 52, 53, 60, 61, 72]. Attacks typically detect the difference in access time, depending on whether the memory location is cached (a cache *hit*) or not (a cache *miss*).

In a Prime+Probe attack [40, 51, 52], the attacker first fills a cache set to be monitored with data. After waiting a while, the attacker accesses the previously cached data. A short access time indicates that the data is still cached, implying that the victim has not accessed data that maps to the monitored cache set. Conversely, a longer access time indicates that some of the attacker's data has been evicted from the cache set, presumably due to victim access.

Cache Attacks on Browsers. Cache attacks have been applied in browsers for website fingerprinting [50, 60, 61], keystroke timing [39], leaking cryptographic keys [20], and as a step in other attacks, such as Rowhammer [22] or transient execution attacks [1, 33, 41, 55].

2.2 Pixel Stealing

SVG Filters. Webpages often visually present sensitive information, such as cross-domain embedded content or links, which change visual appearance based on whether the link

was previously visited by the user. To protect this information, browsers isolate JavaScript executing on a webpage from the rendered appearance of that webpage. Pixel-stealing attacks break this isolation – they allow malicious webpages to recover their rendered appearance, and therefore to recover any rendered sensitive content.

In this work we exploit SVG filters, which are small functions that operate on web page elements after the elements are rendered, but before they are displayed [70]. Filters are composed of several *filter elements*, primitive operations that are parameterized and combined to form the complete filter, and are specified in XML markup. The possible list of filter elements is specified by the SVG standard, and includes several standard image-filtering operations such as color mapping and convolutions. Filters are used to apply artistic effects that may be difficult or impossible to achieve otherwise, such as blurring, to parts of the webpage.

Abusing Filters. Stone [63] showed that data-dependent code paths in filter implementations cause timing variations that can be used to leak pixel data. In response, browser vendors removed all data-dependent branching from their SVG filter implementations [12, 46], as is now required in the formal W3C documentation [70].

More recent works demonstrated that it is possible to leak data due to variations in the processing time of floating point instructions, even in the absence of data-dependent branches [4, 35]. To mitigate this threat, browser vendors now enable CPU flags that ensure these floating-point instructions no longer exhibit excessive slowdown when operating on specific floating-point values [13, 47]. Andrysco et al. [5] has since suggested the use of cryptographic constant-time programming in SVG filters to ensure that the execution time remains constant-time, although no vendor seems to have employed such an approach.

Two recent works show that execution of SVG filters constrain GPU power or thermal budgets, causing data-dependent voltage and frequency scaling that can be observed through timing [68] or via a counting thread [64].

2.3 History Sniffing.

The threat of recovering which websites a user has visited is well established [27, 28, 69]. To protect against attacks that query the `:visited` CSS selector [8, 16, 56] browsers limit the styles that can be applied to a link using this selector, avoiding styles that change the page layout, access external resources, or require significant computation [44, 48].

Other techniques for executing history-sniffing attacks have been proposed, such as measuring various browser caches [7, 17, 25, 62], tricking users into exposing their history by interacting with the webpage [32, 49], or by measuring the time taken for a request to be processed by the server [58].

Stone [63] showed that pixel-stealing attacks can be used to execute history-sniffing attacks by observing whether a link is rendered as visited or unvisited. Several follow up works [4, 35, 36, 63, 64, 68] have found various other mechanisms for pixel stealing. They all, however, suffer from an inherent limitation – they can extract at most one bit of information per rendering of the screen.

2.4 Cross-Origin Isolation

The Cross-Origin Opener Policy and Cross-Origin Embedder Policy (COOP/COEP) along with the `frame-ancestors` directive of the `Content-Security-Policy` header are three related policies that can be configured by web developers to control how their pages can be embedded into other websites. Unlike `X-Frame-Options`, which only allow developers to block all embedding or to restrict embedding to pages from the same origin, these policies provide more fine-grained control over embedding. In particular, COOP/COEP requires that the embedder and the embedded website mutually trust each other – signalled by the use of these policies. That is, in order for a website to embed another website while using these policies, both websites must opt-in to these policies, and both websites must add each other to an allow-list.

In addition, vendors have recently started to use COOP/COEP to signal whether it is safe to allow the website to access two sensitive APIs: high-precision timers and `SharedArrayBuffer`s. Websites that do not enable COOP/COEP are allowed to use `iframe` elements to embed other websites, excluding those that enable COOP/COEP or `X-Frame-Options`, but are restricted from accessing high-precision timers and `SharedArrayBuffer`s. Conversely, websites that enable COOP/COEP have access to high-precision timers and `SharedArrayBuffer`s, but can only embed websites according to the rules specified earlier.

3 Attack Model

We consider an adversarial model used by so-called *click-jacking* attacks [26]. Specifically, we assume that the victim visits a webpage which allows an attacker to execute JavaScript, for example the attacker lures the victim to visit a malicious webpage or the attacker masquerades their attack as an advertisement and embeds it into another otherwise benign webpage. We further assume that the attacker can entice the victim into interacting with the malicious website at least once, for example by clicking on a button to accept cookies. Finally, we assume that cross-origin content embedded by the attacker is not protected with `X-Frame-Options`, `frame-ancestors`, or COOP/COEP headers.

The requirement for interaction is not strictly necessary to recover same-origin content, such as the history stealing attack presented in Section 8, and removing the requirement would result in a model similar to previous pixel-stealing works, however for simplicity of explanation we treat all attacks under the stricter *click-jacking* model.

Hardware and Software. We assume that the victim is using a machine equipped with an Intel processor featuring an inclusive cache, that the user uses either Firefox V92 (or later) or Chrome V92 (or later). Section 7 assumes a version of Firefox that does not feature the Total Cookie Protection or that the feature has been disabled. For Chrome, we were unable to find a method to force the use of the CPU to render filters. We therefore assume that the user uses a device that is on the software rendering list [15]. This includes devices using Linux with open-source drivers and devices that use Mac OS with hardware that has not been added to an allow list.

Experimental Setup. Our experiments are performed on a machine equipped with an Intel i7-6700K CPU with 8 GiB of RAM, running at 2133 MT/s. Measurements throughout the paper are collected using Firefox V102. We have confirmed that the pixel-stealing attack primitive works on both Firefox V108 and V112, enabling the history-sniffing attack to be carried out. We note that in these versions of Firefox an unrelated open bug prevents cross-site filtering from working altogether, inadvertently preventing our cross-site pixel-stealing attack as well. On Chromium V113, we use the --disable-gpu flag to emulate the device being included in the Software Rendering List thereby causing filters to be rendered on the CPU. The attack code can be found at https://github.com/0xADE1A1DE/PixelThief

4 Overcoming Cross-Origin Isolation

Recall that for a website to access high-resolution timers or `SharedArrayBuffers`, the website must use COOP/COEP headers. If that website is to embed or be embedded within a cross-origin website, then that cross-origin website must also use COOP/COEP. Further, both websites must mutually add each other to allow-lists for embedding.

To mount an attack in such a setting, the victim website is not only required to use COOP/COEP, but the victim website must also add the malicious website to an allow-list. While wildcards are allowed, and it may be plausible that overly permissive allow-lists exist in the wild, this still prevents an attacker from mounting attacks on the plethora of websites that do not use COOP/COEP. For these reasons, we consider a setting in which the victim website does not use COOP/COEP.

Such a setting seems to imply a mutually exclusive scenario for the attacker: either they can access high-resolution timers and `SharedArrayBuffers`, or they can embed victim websites, but not both at the same time.

We note that several recent works have investigated the setting of reduced-precision timers, and there are now several advanced techniques that use transient execution to allow high temporal accuracy side-channel attacks to be mounted with reduced-precision timers [29, 31], but before we deploy such advanced techniques, we first ask: *Does the mutual exclusion implied by COOP/COEP actually exist?*

Throughout the rest of this section, we show that the answer is no, provided that we are willing to swap the standard pixel-stealing adversarial model with the stricter *click-jacking* [26] model. The key difference between the two models is the requirement of interaction from the victim.

Bypassing COOP/COEP. We use this interaction to open up a second tab in the browser. The first tab loads a page served with COOP/COEP headers. This page can thus access high-resolution timers and *SharedArrayBuffer*s, but it cannot embed any cross-origin content. The second tab loads a page that is not served with COOP/COEP headers. In this page the opposite holds – cross-origin content can be embedded with `iframe` elements, but high-resolution timers and *SharedArrayBuffer*s cannot be accessed.

We then split our attack between these two pages. Any part of the attack that needs to access high-resolution timers or *SharedArrayBuffer*s is placed on the first page, and any part of the attack that needs to interact with cross-origin content is placed on the second page. The two pages cannot directly communicate with each other because of the COOP/COEP headers. They can both, however, send messages to a server using `WebSocket`s, and the server can relay the messages between the two pages, as we demonstrate in this work.

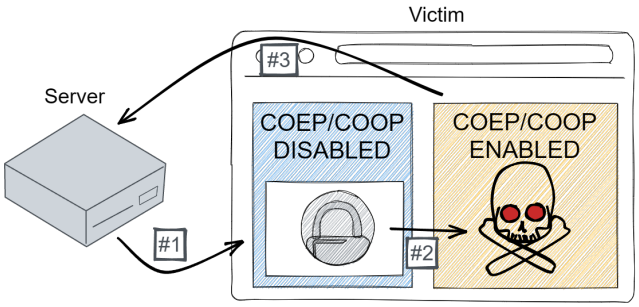


Figure 2: Achieving a high resolution timer. The server sends two pages to the victim operating under different security models, the first embeds the victim page (blue webpage, left) the second mounts the cache attack.

Figure 2 gives an overview of this approach. The server serves two pages to the victim: the first page operates under the old security model, and can therefore embed the victim page (blue webpage on the left), while the second page operates under the COOP/COEP security model, and is therefore able to mount the cache attack (orange webpage on the right). (#1) The server sends a command to the first page to manipulate the victim page into leaking data. (#2) Data leaks through the cache from the victim to the second page. (#3) The second page recovers the leaked data and sends it back to the server.

Same-Origin Content. In cases where the attack interacts with same-origin content, such as the history-sniffing attack presented in Section 8, the two-page architecture is not necessary because same-origin content is allowed to be embedded

into the page while COOP/COEP headers are present.

Measurements throughout the paper use the single-page architecture for same-origin content and the two-page architecture for cross-origin content.

5 Leaking Pixels

In this section we describe the pixel-stealing attack that forms the basis for the other attacks we present in this work. Our pixel-stealing attack reveals sensitive data displayed in parts of rendered webpages by exploiting input-dependent memory accesses in SVG filters.

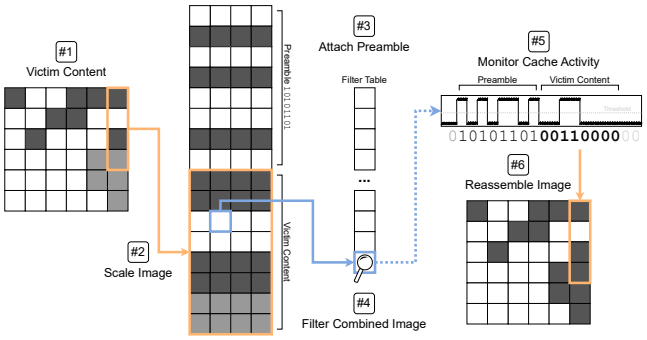


Figure 3: An overview of our attack. (#1) Embed victim content. (#2) Isolate pixels in victim content. (#3) Embed attacker content. (#4) Apply filter to both images. (#5) Record memory accesses. (#6) Find memory accesses correlating to attacker content then recover victim content.

Figure 3 presents an overview of how our attack functions. The attacker embeds the victim image into their page (#1) and uses CSS to isolate and scale chosen pixels (#2). The attacker then inserts their own image above these pixels (#3) to cause the filter to produce a predetermined sequence of memory accesses. When the browser later renders the page, the filter is applied to both images. The filter uses the value of each of the pixels as an offset into a table (#4). In parallel, the attacker mounts a cache attack to record memory accesses to this table over time (#5). Finally, the attacker analyzes the recorded memory accesses to find the predetermined sequence of memory accesses that correlates with their own image. Memory accesses after this sequence are attributed to the victim image and can be recovered and rearranged with other recovered pixels to reconstruct the full victim image (#6).

While the attack functions in both Chrome and Firefox, Firefox's implementation of the relevant filters is significantly easier to describe. We therefore focus on Firefox for the remainder of this section. We begin with a description of the vulnerable SVG filter. Then, we introduce and overcome the browser's use of GPU-based implementations. Finally, we end the section with a discussion of measurement frequency and how it affects the performance of the attack.

5.1 The feComponentTransfer Filter

`feComponentTransfer` is an SVG filter element that allows designers to remap the colors of their webpages. It can be used to implement various recoloring effects, such as sepia or greyscale. To that aim, the filter element operates over each pixel in the input image individually. First, it splits each pixel into its component red, green and blue values. It then maps each of the components' values, in addition to its alpha value, to target values based on functions defined by the designer. Finally, it merges the components back together to form the pixel color in the output image. Listing 1 (Lines 2–6) shows an example of how such a filter is defined.

```
1  <filter id="filter_id">
2      <feComponentTransfer>
3          <feFuncR type="discrete" tableValues="0
           ↪  1"></feFuncR>
4          <feFuncG type="discrete" tableValues="1
           ↪  0"></feFuncG>
5          <feFuncB type="discrete" tableValues="0.3
           ↪  0.6 0.9"></feFuncB>
6      </feComponentTransfer>
7  <feGaussianBlur stdDeviation="0" />
8  </filter>
```

Listing 1: A filter that executes feComponentTransfer and then uses feGaussianBlur to blur the result.

```
1  void TransferComponents(uint8_t input[N],
2   uint8_t output[N],
3   uint8_t tables[3][256]
4   ) {
5       for (int32_t i = 0; i < N; i += 3) {
6          for (uint32_t c = 0; c < 3; c++) {
7              output[i + c] = tables[c][input[i + c]];
8          }
9       }
10    }
```

Listing 2: Firefox's Implementation of feComponentTransfer.

Firefox's feComponentTransfer Implementation. The `feComponentTransfer` specification allows for several ways of defining the mapping of each channel to its target color, including identity mapping, linear and gamma maps, and interpolated and discrete tables [43]. Regardless of the definition chosen, internally Firefox uses a 256-entry lookup table for implementing each of the mappings. These tables are filled in during a precomputation phase, in which the values in the filter definition are mapped from floating point numbers between 0 and 1 into integers between 0 and 255, and the rest of the values of the table are interpolated according to the SVG filter specification. Listing 2 shows a simplification of Firefox's implementation for `feComponentTransfer` in C-like pseudo-code. Lines 1–4 define the function and its parameters.

The `input` and `output` parameters are the filter's input and output images, represented as interleaved 8-bit red, green, and blue values. The last parameter, `tables`, holds precomputed tables obtained by computing the functions defined in Listing 1. Line 5 iterates over each pixel in the input and output images. Line 6 iterates over each of the component values for each pixel. Finally, Line 7 applies the filter by performing a lookup in the appropriate table.

Analysing `feComponentTransfer` Cache Access Patterns. We now proceed to analyze the cache access patterns of Listing 2. Throughout the analysis we assume that each data element cached in the LLC, commonly referred to as a *line*, is 64 bytes long. To increase cache efficiency, the cache mapping function of the CPU aims to assign consecutive lines in memory to different sets in the cache [42]. Assuming that the data structure holding the tables is stored on a 4K page boundary, each 256-byte table will therefore span four different cache sets.

Consider a picture with two pixels, where the first pixel is black $(0,0,0)$ and the second pixel is white $(255,255,255)$. As before Lines 6 and 7 of Listing 2 iterate over each component of each pixel, with Line 8 performing table lookups. A side effect of accessing the table entries during lookup is that accessed entries are cached within the CPU cache hierarchy. That is, applying the filter to the black pixel causes the first 64 entries of the table to be loaded into the cache. Similarly, processing the white pixel causes the last 64 entries of each table to be cached. Consequently, an attacker who can tell apart accesses to the cache set holding the first 64 entries from the last can detect whether the pixel was black or white.

5.2 Executing `feComponentTransfer` on the CPU

To improve rendering performance, Firefox version 92.0 and newer attempt to offload image rendering to the GPU. This poses a challenge to adversaries that attempt to exploit vulnerabilities in CPU implementations of filter elements, simply because the vulnerable implementations do not get executed. We now describe how to overcome this issue, forcing Firefox to execute any filter element on the CPU.

Filter Elements without GPU Implementations. Although Firefox includes GPU-based implementations for many filter elements, some filters are still only supported through CPU-based implementation, see Appendix B for a complete list. One such example is `feGaussianBlur`, which applies Gaussian blur [67] to an image. We observe that none of the CPU-only filters are suitable for performing a side-channel attack. Most of these filters, including `feGaussianBlur`, achieve their effect through a convolution of the image and, as such, do not exhibit data-dependent memory access patterns that leak image data through the cache.

Forcing CPU Computation. Being unable to construct an attack using filters with only CPU-based implementations, we searched for methods for forcing Firefox to fall back to executing `feComponentTransfer` on the CPU. To that aim, we investigated Firefox's behavior when combining filter elements, especially when mixing CPU- and GPU- based filters. More specifically, when we combine `feComponentTransfer` with `feGaussianBlur`, we discover that both filters are executed on the machine's CPU (Firefox still computes `feComponentTransfer` on the GPU if the filters are combined in the opposite order). We notice that this observation holds true for all CPU- and GPU-based implementations: in the case that the last filter is CPU-based, the entire filter stack will be executed on the CPU, regardless of the number of GPU-based filters present.[1]

Our Filter Stack. To exploit this behavior, we use the filter shown in Listing 1. It consists of an `feComponentTransfer` filter element (Lines 2–6), connected to the inputs of an `feGaussianBlur` filter element (Line 7). As Firefox does not provide a GPU implementation for `feGaussianBlur`, it uses the CPU implementation for this filter. Moreover, as observed, this also results in using the CPU-based implementation showed in Listing 1 for `feComponentTransfer`. We note that the choice to use `feGaussianBlur` is arbitrary. We can use any other CPU-only filter to force execution of `feComponentTransfer` on the CPU.

Chrome. Chrome exhibits similar behavior to Firefox, in which Chrome will prefer to execute `feComponentTransfer` on the GPU. While we were unable to find an analogous method to force Chrome to execute `feComponentTransfer` on the CPU, we note that Chrome has an extensive GPU block-list that disables various aspects of GPU acceleration for specific hardware configurations. This includes devices with outdated drivers, devices running Windows Vista or earlier, devices running Linux with third-party drivers, and devices running MacOS X without a GPU on the allow-list [15]. An attacker would therefore be limited to attacking a victim with a device operating in one of these configurations. In our experiments, we emulate this scenario using the --disable-gpu flag, which disables GPU support in Chrome.

Signal Amplification. Cache attacks have a limited measurement frequency, which affects their ability to distinguish between events that occur rapidly [3, 53]. Recall that our goal is to perform a cache attack on the implementation of `feComponentTransfer` in Listing 2. To recover the image, we want to monitor each table lookup performed by the filter. The filter, however, is too fast – it performs an access every few cycles, whereas a typical cache attack requires hundreds or even thousands of cycles per measurement [3, 40, 53, 72].

Past works have explored methods for reducing the execution speed of the transmitter [2, 3, 23, 45, 65], but these rely on features that are not available from JavaScript, hence we cannot use them. Instead, we build on past works [4, 36, 63]

[1] We note that explicit linking of filter-elements using `in` attributes interferes with this behavior and prevents GPU-based filters from executing on the CPU.

that stretch the image such that the filter must perform repeated filter table accesses with identically colored pixels. As Firefox breaks up the screen into 256×256 px tiles, we stretch each pixel horizontally 256 times and vertically 256 times, divided by the number of pixels we want to measure in every attack iteration, as described below.

6 Recovering Pixels

In Section 5, we construct the transmitter side of our pixel-stealing attack. We now turn our attention to the receiver side. We first explain how we prepare for mounting a cache attack. We then describe the cache attack we use. Our aim is to construct a fast attack that allows a high transfer rate. We selected the Prime+Probe [40, 51, 52] attack for this purpose. As discussed in Appendix A, we also evaluated the window-less Prime+Scope method of Purnal et al. [53], but discovered that it is less appropriate for our setting. We now discuss how we synchronize the transmitter and the receiver, then we evaluate our pixel-stealing setup described in Section 3.

6.1 Detecting Transmitter Communications

Recall that unlike other attacks mounted on filters [4, 36, 63], we measure leakage in parallel to filter execution, in order to perform multiple measurements and leak multiple bits of sensitive information from a single filter execution. The attacker, however, does not know if its cache measurement occurred while the browser filtered the webpage, or while the browser was performing some other tasks. Throughout this section, we describe how we detect whether cache measurements are transmitter communications or noise from other processes, and how we leverage the ability to detect transmitter communications to establish the communication channel. We first assume that the cache sets to be monitored are known to the attacker, and then show how these sets can be discovered.

Adding a Preamble. To detect the beginning of a transmission, we follow a standard technique in communications and introduce a packet preamble. The preamble serves two functions – first, it allows the receiver to detect the start of transmission. Second, because its contents are known, it allows the receiver to match its clock rate, which is based on a free-running counting thread, to the rate of the sender. Recall that the signal is encoded into cache activity by moving an attacker-controller filter over a page containing victim content. We insert the preamble into the signal by prepending a known image into the page, just before the victim content, such that the filter will operate over the preamble first, and then process the sensitive information.

Listing 3 shows how the preamble is placed onto the attacker webpage. Lines 1 and 6 apply the filter to the preamble and the sensitive content. Line 2 displays the preamble as an image on the screen. Lines 3–5 display the sensitive content

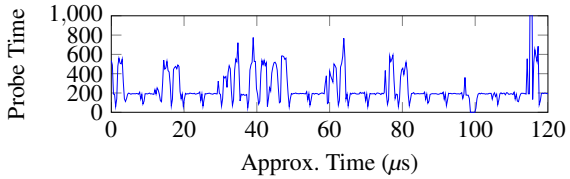
```
1 <div style="filter: url(#filter_id);">
2     <img src="preamble.png" />
3     <div>
4         <!-- sensitive content -->
5     </div>
6 </div>
```

Listing 3: Preamble construction

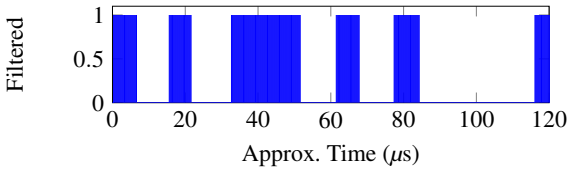
on the screen. Since the filter goes over the image first horizontally and then vertically, the filter will first process the preamble image, and then the sensitive content.

Detecting a Preamble. With the transmitter equipped to send a preamble, we move to preamble detection in the receiver. Recall that the exact rate of the receiver's counting thread is not known to the attacker, and may even change over time. Thus, it is not possible to simply search the side-channel trace for the preamble, since it is stretched in time by an unknown factor. Instead, we search for the preamble at multiple different possible stretching factors. We speed up this process by first processing both the preamble and the trace using run-length encoding. In this representation, stretching in time is equivalent to multiplying by a constant. Once the preamble is found, the attacker knows both when the frame starts, and what is the appropriate data rate at which it should be sampled.

We implement the detector in the following manner. Input first passes in to a low-pass filter to remove high-frequency noise, then samples are compared to a threshold to classify them into cache hits and cache misses, then the recorded trace is run-length encoded, and finally, a substring search is performed to find the preamble. An example of cache activity measured by our attack before and after filtering the signal can be seen in figure Figure 4.



(a) Raw Data



(b) After Filtering

Figure 4: Cache Access Pattern Corresponding to a Preamble of '101011101010001', as Recorded using Prime+Probe.

Packet Payload. As mentioned in Section 5.2, our attack can be configured to leak different amounts of pixels per filter in-

vocation, depending on the chosen tradeoff between speed and error rate. Each invocation of the filter essentially transmits a "packet" of side-channel data, starting with the preamble and following with the sensitive payload. After the packet preamble has been detected using the method described above, and the scaling factor has been determined, the receiver can apply the same factor to the payload and read out the sensitive data.

Finding The Target Set. Until this point, we assumed the attacker already knows which cache set will be used by the filter processing code, and the only needs to detect when the data is actually being transmitted. To detect which cache set is being used by the filter processing code, the attacker begins by constructing an eviction set for each cache set, using the technique of Vila et al. [66]. After all eviction sets have been constructed, the attacker repeatedly applies the filter to a known image containing only the preamble, thus encoding the preamble into the cache. Next, the attacker goes over each possible set, recording a trace which is guaranteed to be long enough to include at least one screen drawing operation (practically, a little over 16 milliseconds), and searches for the preamble in this trace. If the attacker detects the preamble, they record the eviction set, so that future measurements do not need to perform this calibration step. If the attacker does not detect the preamble, they move to the next possible set.

6.2 Evaluation

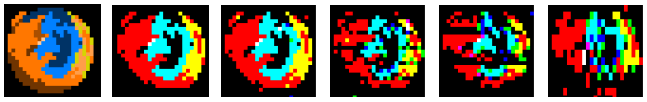


Figure 5: A pixel-art Firefox logo leaked using our pixel-stealing attack. From left to right: original image, ideal leakage, and actual leakage with payloads of one, four, eight, and 32 pixels.

In the previous section we described the final steps to detect cache leakage from `feComponentTransfer`. In this section we evaluate the effect of varying the payload size, the time required to identify the target set, and the effect of noise on the leakage rate and accuracy of the attack.

Throughout this section we mount our pixel-stealing attack on an image of a pixel-art Firefox logo in Figure 5 (First from the left). We serve a webpage that contains both the image and the pixel-stealing attack code from a local HTTP server. In Section 6.2.1, we evaluate the speed and accuracy trade off in selecting the payload size. In Section 6.2.2, we evaluate the capability of the attack to identify the correct target set. Finally, in Section 6.2.3, we evaluate the attack in a more realistic setting, with an additional concurrent workload.

We mount Prime+Probe only on the final entry in the `feComponentTransfer` filter table, which corresponds to indexes 192–256 of the table, and ignore accesses to any other

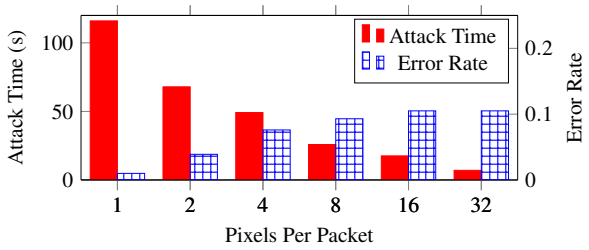


Figure 6: Attack time and error rates per packet sizes.

index. For each pixel, if an access to this cache line is observed, then we assume that index 256 of the table was accessed. Otherwise, we assume that index 0 of the table was accessed. Under these assumptions Figure 5 (Second from the left) is what the image would look like if there were no errors in the cache channel.

6.2.1 Varying Payload Size

First, we conduct an experiment to explore the trade off between speed and accuracy when selecting the payload size. We measure the time requried to recover the entire image after the attack setup has completed (i.e. setup times are excluded from the measurement). We recover an image with 1, 4, 8, and 32 pixels per packet. We recover the image 100 times at each packet size, and report the median elapsed time and the median error rate. Figure 5 (Third to sixth from the left) shows representative recovered images at each packet size.

The main type of error introduced is a skewing of the location of pixels in the image, since we leak a vertical column of pixels, with as many pixels as can fit in a single packet, all at once. Thus, if the attack incorrectly gauges the length of a pixel in the signal, a pixel may be inserted or excluded during recovery. Then, all following pixels in the column will be incorrectly offset. The effects of these errors are more visually apparent as the packet size increases.

Figure 6 shows quantitative results. We apply the Levenshtein distance [38] to measure the rate of errors, and then select the median value from the 100 measurements. The median runtime to leak the entire 25px by 25px image in three separate color channels (red, green, and blue) is 116s (16px per second) at the slowest speed to 7s (267px per second) at the fastest speed. As expected, the median error rate increases as the speed increases, starting at 1% at the slowest speed and increasing to 10% at the highest speed.

6.2.2 Finding the Target Set

Thus far, our evaluation has assumed an attacker that is ready to leak an arbitrary number of pixels. We now move to measuring the time taken to identify sets containing filter tables.

Work	Side-Channel	Measurement Target	Mitigated	Speed (b/s)
Stone [63]	Filter Optimizations	Page Rendering	✓	10
Kotcher et al. [36]	Filter Optimizations	Page Rendering	✓	<1
Andrysco et al. [4]	Subnormal Floats	Page Rendering	✓	16
Kohlbrenner and Shacham [35]	Subnormal Floats	Page Rendering	✓	60
Wang et al. [68]	Power Consumption	Page Rendering	✗	3
Taneja et al. [64]	Power Consumption	Page Rendering	✗	<1
This Work	Memory Accesses	Memory Accesses	✗	267

Table 1: Comparison of pixel-stealing attacks. We report which channel pixels are leaked through, how each work extract information from that channel, whether the presented attack has been mitigated, and the speed of the attack in pixels per second (roughly).

Setup. To identify the set, we use the technique described in Section 6.1. To validate our results, we used a patched version of Firefox that prints the virtual address of the cache set that holds the table. We use the methods of Maurice et al. [42] to infer the ground-truth set index in the cache, and compare it to the found index.

method. Since we may incorrectly detect a transmitted preamble, we record several traces and if we detect a preamble at least once while recording a set then we consider the set to be a candidate. We then record the same number of traces from the candidate again and if we detect preambles in at least three of the traces then we consider the set to be the target set. We repeat this process 21 times for each number of recorded traces and report the median time to identify the target set and the accuracy in Table 2. The median time to identify the target set is 372–1257 seconds and 48–77% of the runs identify the target set correctly on the first iteration of all eviction sets.

samples	Time (s)	Accuracy
40	1257.59±169	77%
30	999.56±152	67%
20	644.57±115	56%
10	372.39±60	48%

Table 2: Median time of identifying a target set and accuracy of the first iteration.

6.2.3 System Noise

Finally, we evaluate our attack in a more realistic setting in which the victim has other tabs open in the browser. We continue to use the same experimental setup as in Section 6.2.1, but now we open a new tab in the browser and have that tab play a video from YouTube. Figure 7 shows a representative qualitative result, where we can see slight degradation in the quality of the recovered image. With 100 runs of the experiment and comparing to the results from Section 6.2.1, the median runtime of the attack is 476s (4.09×) with a median error rate of 2.72% (+2.68×).

Figure 7: Left to right: Original, recovered image with zero errors, recovered image without system noise, recovered image with system noise.

6.3 Comparisons to Existing Works

Table 1 compares the speed of our work with other works that mount cross-origin pixel stealing attacks. All listed works mount timing side-channel attacks on filters to extract cross-origin pixels. We compare the channel that is used to leak pixels, how each work extracts information from that channel, whether the presented attack has been mitigated, and the speed of the attack in pixels per second.

Earlier works exploit optimizations in the filters themselves while later works, including our own, exploit leakage through microarchitectural side-channels.

All previous works extract side-channel information by measuring the time taken to render the page, indirectly measuring the execution time of the filter. In contrast, our attack extracts side-chanel information via measuring the execution time of victim memory accesses, indirectly measuring memory behavior of the victim. This difference is what allows us to achieve speeds that exceed the refresh rate of the display.

As with recent pixel stealing attacks, our work is yet to be mitigated. Fortunately, generic countermeasures for cross-origin pixel stealing attacks have been proposed, which would mitigate all attacks, including our own. We discuss these countermeasures in more detail in Section 9.

7 From Pixel Stealing to Text Stealing

We now show how to build upon our basic pixel-stealing attack to create a text-stealing attack that can recover sensitive text content from third-party websites. Unlike previous works [63] which mount pixel stealing attacks on monospaced fonts that can be easily pixelated, our work deals with text rendered with vector-based graphics using proportional fonts that feature kerning. In this section we describe the challenges that this introduces and how to overcome them.

Figure 8: Pixel-based content continues to stay pixellated after stretching (Left), whereas vector-based content is resampled after stretching (Right). Both regions in the original images are the same 3×3px size.

Text Rasterization. Our pixel-stealing attack scales the image to select individual pixels and amplify them. Pixel-based content can be scaled in such a way that it preserves the discrete pixels of the original image. (Figure 8 Left). In contrast, vector-based content, such as text, is represented using mathematical expressions, which are sampled to produce an image. When a vector-based image is scaled, scaling is applied to the mathematical expressions, and then the image is sampled at the display resolution (Figure 8 Right). Due to this difference, using transformations to isolate the color of a single pixel is simply not as effective for text.

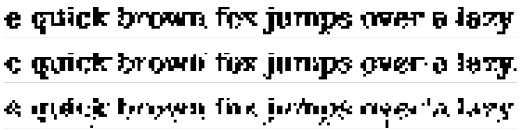


Figure 9: Naive Text Stealing at 1, 2, and 4 Pixels per Packet

Pixel Stealing on Vector-Based Content. To illustrate the problem, we perform our pixel-stealing attack on a classic pangram, and present the recovered text in Figure 9.

Compared to pixel-based content there is greater visual degradation, even when recovering one pixel at a time. Despite the visual degradation, there is sufficient redundancy within this pangram to be able to easily read it. While this is often the case for large pieces of text written in natural languages, this is often not the case for data of interest to attackers: victim's names, identification numbers, and passwords are all examples of sensitive data that is short and often contains very little redundant information.

A natural solution to this problem is to sample the image at a higher resolution to minimize this effect. However to double the resolution requires four times the samples, because grows with the square of the side length, and therefore we ask whether there is a better technique to extract text?

Stone's Method. We answer the question in the affirmative and present a solution that adapts Stone [63] binary-lookup technique. Stone's technique is based on the observation that given an image of an unknown character, if we discover the color of a carefully chosen pixel, we can exclude roughly half of the possible characters. However, Stone applied their tecnique to rasterized monospaced fonts, not proportional-width fonts that are more commonly used to display text.

Figure 10: An example region in the letters 'A' through 'F' in the MS Sans Serif font. The region is entirely inside or outside each letter.

Choosing a 'Pixel'. We begin by resolving the problem of identifying the color of 'pixels' with vectorized content. The observation that underlies the ability to adapt Stone's method to vectorized content is that we can isolate and arbitrarily scale any rectangular region of a character. Further, we can select regions that lie entirely inside or entirely outside the shaded part of a character, which guarantees the region only contains a single color, and therefore we do not observe any measurement artefacts. Figure 10 illustrates such a region for the letters 'A' through 'F'.

Finding Regions. We find such regions in an offline step, we randomly choose `x` and `y` coordinates and use a fixed width and height – these values are on the scale of one millionth of a pixel. We use the Selenium browser automation framework[2] to verify that the region lies entirely inside or entirely outside all characters we want to distinguish.

After collecting enough of these regions, we can distinguish each character but we will have redundant regions. We construct a minimized set of these regions using a greedy search algorithm – we find which region provides the most information and add it to a new set. This is repeated until every symbol can be distinguished. While this approach does not guarantee the theoretical minimum number of regions, we find that it builds a set very close to the theoretical minimum.

Kerning. Next we resolve the primary problem that proportional width fonts pose to the attack: kerning – the practice of changing the spacing between letters to achieve a better visual effect. With kerning, the position of a letter depends

[2] https://www.selenium.dev/

on the one or two letters that precede it. While kerning can happen between three or more letters, we limit ourselves to the common case of two letter kerning.

During region collection, we select a character at random (including no character) and prepend it to all characters we want to distinguish, we then continue with region collection as normal. This results in a set of regions for every starting character, which we then apply the minimization procedure to.

7.1 Text Stealing Results

We now move to evaluate our text stealing technique. We conduct two experiments. The first experiment measures the rate at which we can recover random characters of text. The second experiment we perform a proof-of-concept attack on Wikipedia, recovering the username of a logged in user.

Measuring Data Rate. First, we measure the rate at which our technique can steal randomly generated characters. We use the technique as described in the previous section, starting with the offline region collection on the default browser font over the set of all letters. We then select the letters *N*, *a*, and 1 and leak each one ten times. To reduce the effects of noise we sample the cache twelve times and perform a majority vote. The median time to leak a letter is 56 seconds with an accuracy of 93.3%.

Proof-of-Concept. We now move to show a proof-of-concept attack on cross-origin content, specifically we recover the username from a logged in Wikipedia account. We start by creating a new Wikipedia account and logging into it. Since we are performing a cross-origin attack, we deploy our two-page architecture as explained in Section 4 to bypass COOP/COEP. For this experiment, we manually interact with the page so that the second page is opened in a second tab.

Identifying Username Location. In order to steal the username, we first need to identify where the username is. We were able to side-step this in the previous experiment because we could simply apply the filter directly to the element containing the text. However, when we apply the filter to cross-origin content we must apply the filter to the `iframe` which in turn applies the filter to the entire page.

We manually use `getBoundingClientRect` to roughly guess where the username begins then refine this guess through the use of a Selenium script. This script slightly adjusts the starting position and then simulates the attack using screenshots. This step is performed once as an offline preparation step and once the script identifies the location of the text, we hardcode it directly into the attack.

Stealing the Username. Finally, we perform the attack to validate that we are able to steal text in a cross-origin setting and recover the username of the account.

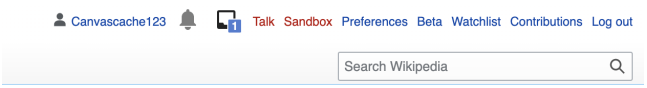


Figure 11: Layout of a Wikipedia User Name

```css
.leak {
  display: inline-block;
  width: 1px;
  height: 1px;
  background-color: black;
  font-size: 0;
}

.leak:visited {
  background-color: white;
}
```

Listing 4: CSS Style for History Sniffing

8 History Sniffing

In this section, we show how to utilize our pixel-stealing attack for history sniffing. We first describe a straightforward approach that tests one or a small number of URLs at a time. We then show how to use batch testing to improve sampling rate when the number of visited URLs is expected to be low compared to the total amount of possible URLs.

8.1 Straightforward History Sniffing

Several prior history-sniffing attacks rely on the observation that the color of a link indicates whether it was visited or not [4, 25, 36, 49, 63] We build on the same observation.

To implement the history-sniffing attack, we first create a pixel whose color depends on whether a URL has been visited. For that, the attacker page includes a CSS style similar to Listing 4. The first set of rules in the code specifies that elements of class `leak` are rendered as colored boxes whose size is 1×1 pixels and whose color is black. The second rule changes the background of such elements to white if they have the property `:visited`, i.e. if they are a link to a URL that the user has visited in the past. When the class is applied to a link, e.g. using the HTML code `<a href="https://example.com" class="leak">`, the browser renders a 1×1 pixel whose color is white or black depending on whether the URL (in this example `https://example.com`) has been visited or not.

Having created a pixel that shows whether a URL has been visited, we can use the techniques of Section 5 and Section 6 to recover the color of the pixel, and therefore recover whether the URL has been visited by the victim.

To sniff multiple URLs we repeat these steps to produce several pixels for several different URLs, such that the color of each pixel indicates whether its corresponding URL was

visited. We then apply the technique of Section 6 to leak the color of each pixel simultaneously. After we have leaked the color of the pixel, we replace the URL for each link which causes the browser to recalculate the appearance of the link which in turn updates the color of the pixel to reflect whether the new URL has been visited or not.

8.2 Overview

Recall that history-sniffing attacks do not directly reveal victim browsing history, rather the browser is used as an oracle to reveal whether a given URL has been previously visited by the victim. Our attack is no different, we create a link with a specific URL such as *example.com* and apply a style so that the color of the link depends on whether the user has previously visited the URL. We then use our pixel-stealing attack to leak the color of the link which reveals whether that specific URL has been visited. We call this the *simple* method.

In addition to the *simple* method, we use a second method that exploits the behavior of the cache to query over a thousand URLs in parallel. This variation of the technique varies from other history-sniffing attacks. Typically the information revealed is whether a specific URL has been visited or not. Instead, this method uses the browser as an oracle to reveal if *any* of the URLs in a given set have been visited but does not reveal which. We call this the *set query* method.

The advantage of the set query method is that if none of the URLs are visited then $O(1)$ queries are sufficient to reveal this information and if one URL is visited then a binary search can be performed and $O(\log n)$ queries are sufficient to reveal which one of n URLs has been visited. In contrast, the *simple* method would require $O(n)$ queries to reveal the information.

The set query method requires vastly less queries when the number of visited URLs is small. However, as the number of visited URLs increases, the number of required queries using the set query method approaches $O(n \log n)$. For this reason, our attack employs both the *simple* and the set query methods, and switches between the two depending on which is likely to reveal the information with the least queries.

8.3 Set Query

In this section we describe the set query optimisation.

Recall that our pixel-stealing attack crafts an SVG filter with memory accesses that correlate to pixel values and these memory accesses are measured through the use of Prime+Probe. However, because memory accesses can be executed much faster than Prime+Probe the SVG filter can perform several memory accesses that correlate to different pixels between each measurement. The effect is that memory accesses are 'blurred' together as the first access to a given piece of memory brings it to the cache but any additional accesses do not have any effect since the memory is already cached.

Avoiding Blurring. Such behavior is generally undesirable in the pixel-stealing attack since the attack does not know which pixels correspond to which memory accesses. For example, a set of pixels containing exactly one white pixel has the same cache behavior as a set of pixels containing multiple white pixels since they are 'blurred' together. In the pixel-stealing attack we completely remove blurring by stretching the image, duplicating pixels, to such an extent that between any two cache measurements, the majority of memory accesses by the SVG filter correspond to one pixel value.

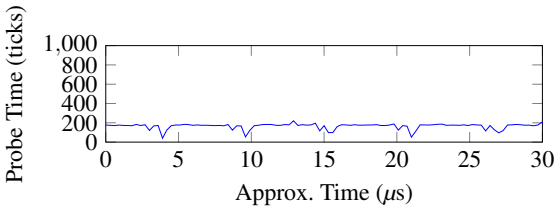
Behavior of Cache. In contrast, the set query method embraces this behavior to achieve much higher data rates. The high-level idea is that while we do not know the precise pixel values when they are 'blurred' together, we do know if all of the pixels share the same value. For example, when all of the pixels are black, only one location in memory will be accessed, the location of black in the lookup table. Similarly if all pixels are white only the location of the white color in the lookup table will be accessed. However, when some pixels are black and others are white, both the black and white color in the lookup table are accessed.

Set Query. The set query method exploits this property. It encodes unvisited links as black and visited links as white and observes the cache. If it only observes memory accesses to the black color in the lookup table, then it concludes that none of the links have been visited. *Detecting visited Links* As we did in Section 6.1 we construct a packet including the preamble and a payload, this time the *expanded* payload will include only one pixel of each website mapping to one access to the filters table. We also use the same method of payload detection in order to know the location and size of the payload cache activity. An example of the cache activity on what we detected as the payload part of the packet using the set query method can be seen in Figure 12. As can be seen the activity of one table access make the table line stay in the cache until our Prime+Probe measurement evicts it and thus we get at least one sample that it took longer time to retrieve.

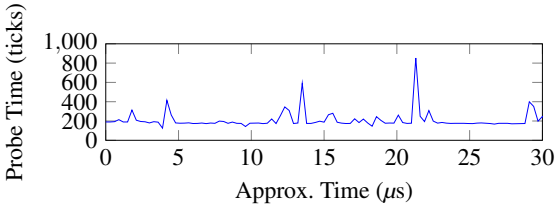
Performance Problems. This can be extremely powerful when we expect few URLs in a set to be visited. In the ideal case we observe that none of the URLs have been visited after only $O(1)$ samples. Otherwise we observe that some of the URLs have been visited. We determine which URLs by splitting the set into two and sampling each, if we observe that a subset has at least one visited URL we repeat this process for the subset. In the event that precisely one URL has been visited, we can perform a binary search and find which URL is visited after $O(\log n)$ samples. However, as the number of visited URLs increases the number of samples required approaches $O(n \log n)$ and it can be faster to simply use the *simple* method which always determines which URLs are visited with $O(n)$ samples.

Method Selection. We resolve this problem by dynamically switching between the *simple* and the set query methods depending on whether the website is in the Alexa Top 1,000. If

the website is in the top 1,000 websites, the attack assumes there is a high probability of visiting the URL and uses the *simple* method. Otherwise, the set query method is used.



(a) No Visited Links



(b) Some Visited Links

Figure 12: Access Patterns for the Set-Query Attack.

8.4 Experiment Description

In this section we describe the experiments we perform to characterize the performance of our history-sniffing attack.

Fabricating History. We begin the experiment by fabricating browsing history. We first sample the Alexa Top 50,000 websites, such that popular websites are more likely to be included in the fabricated history.[3] We populate the browser history using Selenium. Our script opens each website and then verifies that the website was correctly loaded into the history by reading the contents of the screen. We use this as the ground truth when evaluating the attack.

History Sniffing. The history-sniffing attack runs over the top 50,000 websites. As described above, we are ignoring the top 1,000 websites and split the remaining 49,000 into 49 sets of 1,000 URLs, queried using the set query method.

8.5 Results

In this section we investigate the effect the number of samples has on the attack runtime and attack accuracy. We record several samples and use majority-voting to select the result. We vary the number of samples and measure the execution time, the recall (proportion of websites that the attack was able to recover from the victim history), and the precision (proportion of websites the attack claimed to be in the victim history that were actually in the victim history). In each experiment we use Selenium to fabricate a browsing history that is then recovered by the attack. Table 3 summarizes the results. As expected the attack accuracy and attack execution time both increase as the number of samples increases.

Fixed Sample Count

Samples	Runtime (s)	Recall	Precision
1	26.27	10%	3%
2	22.75	0%	0%
3	91.21	32%	26%
4	81.62	36%	9%
5	158.26	42%	42%
6	165.71	47%	26%
7	286.61	40%	60%
8	254.34	55%	47%
9	354.77	52%	59%
10	354.62	52%	56%
11	441.97	47%	57%
12	420.86	64%	65%
13	525.45	47%	80%
14	477.48	68%	73%

Table 3: Accuracy of the set query method using different measurements counts and a majority vote. Recall is the proportion of websites that the attack was able to recover from the victim history. Precision is the proportion of websites claimed by the attack to be in the victim history that were actually in the victim history. The time excludes the search for the victim eviction set.

9 Countermeasures

Total Cookie Protection. Total Cookie Protection is a new feature that enforces protection on HTTP cookies by validating that each requested cookie was requested from the domain that set it. Otherwise, the request is blocked. Mozilla enabled total cookie protection worldwide for all users on 14.6.2022. Practically, the update applies for Firefox versions starting from Firefox 101, all versions of Firefox Nightly, and none of the versions of Firefox ESR. Although this feature mitigates all cross-origin pixel stealing attacks, it does not affect the history-sniffing attack because the way links are rendered as visited or not is independent of any cookies. Safari features a countermeasure called Intelligent Tracking Protection that prevents cross-origin pixel stealing attacks in a similar manner.

Constant Time Programming. is a programming style that aims to prevent the use of known-leaky constructs with program secrets [9, 10, 51]. In particular, secrets cannot be used in the condition of control flow statements, as the address of a memory access, or as an argument for a variable time

[3] We include websites in the history with a probability of roughly $\frac{1}{n}$ where n is the position of the website in the Top 50,000.

instruction (e.g. division). These constraints have been shown to be highly successful at preventing side-channel leakage but often cause significant program overhead and it can often be unclear how to adhere to such constraints in high level languages, such as the SVG filter language. For these reasons, we only recommend this solution to browser vendors for any fallback filters that are executed on the CPU.

frame-ancestors. The `frame-ancestors` directive of the `Content-Security-Policy` is a widely supported feature that provides developers the ability to deny loading from within an `iframe` to prevent malicious embedding. While this solution does not address leakage from a webpage, it prevents a malicious webpage from embedding and then leaking content from an otherwise safe victim webpage. Considering that this technique also protects against other attacks, such as Click-Jacking [26, 57], we recommend this for all website operators. For website operators that need to support embedding within arbitrary webpages, consider removing sensitive information when embedding content.

10 Limitations & Future Work

Pixel Stealing. Our attack is limited to environments in which SVG filters are executed on the CPU rather than on the GPU. If the user uses Firefox, then any environment is applicable. If the user uses Chrome, then the user must have an environment on the Software Rendering List [15]. Our attack is limited to Intel CPUs featuring inclusive caches. In principle, the attack could likely be extended to CPUs that feature non-inclusive caches, for example by mounting an attack on coherence directories [71], but we leave such extension to future work.

Text Stealing. Our attack is limited to scenarios in which the attacker is able to guess the font used by the user. In cases where this is not possible, for example if the user changes the font, the user is not vulnerable. Moreover, while the technique extends to languages with very large alphabets, such as Mandarin, in practice as the alphabet size increases the attack becomes less practical especially compared to simply sampling each pixel multiple times.

Cross-Origin Content. Our attack assumes that cross-origin content does not use COOP/COEP, `X-Frame-Options`, or `frame-ancestors`. Unfortunately, the majority of websites still do not use these features and are therefore vulnerable to our attack [6, 11, 18, 24, 30, 37]. In addition, we assume that the browser does not support or does not enable `Total Cookie Protection` or similar countermeasures.

Theoretical Maximum Bitrate. Our pixel stealing attack in its maximum pixels-per-packet configuration has a theoretical maximum throughput of $32 \times 60 = 1920$ bits per second. In practice our attack achieves a maximum speed of 267 bits per second, while our attack is the fastest side-channel based pixel stealing attack it is roughly 14% of the theoretical maximum.

This is in part because the attack cannot spend all of its time recording cache activity, it must also spend some time analysing recorded traces to identify and extract packets. We measure the amount of time the attack spends analysing recorded cache traces and find that it spends roughly half of its time analysing traces.

In principle, we could offload analysing traces to a separate thread in order to allow the attack to record a greater portion of cache activity. We found naive attempts at doing this simply moved the problem from analysis to message passing overhead between threads. We leave a more thorough investigation of efficient methods to implement multi-threaded collection and analysis to future work.

While this shrinks the gap between the theoretical maximum and the actual throughput by half, there is still a significant gap remaining. We find that this is due to the attack missing on average a quarter of the packets – after accounting for the packets that would be missed while analysing. We believe this is due to system noise obscuring the preamble such that it is no longer recognized by our signal processing chain. While we could deploy a more robust signal processing chain that features more advanced signal processing techniques, it is unclear if the additional captured packets would outweigh the packets lost due to the additional analysis time. We leave a more thorough investigation of this trade off to future works.

11 Conclusions

In this work we present a cache-based pixel-stealing primitive that targets Firefox's SVG filtering engine. Despite several efforts to mitigate leakage, we show that pixel stealing is not only possible but is highly practical. To that aim, we develop an asynchronous attack architecture where the attacker measures cache leakage in parallel to SVG filter execution. This allows us to increase the total data rate, overcoming the previous hard limit of 60 pixels per second. It also allows the attack to completely bypass the Cross Origin Resource Sharing (CORS) policies of the browser and avoid its side-channel countermeasures. To the best of our knowledge, our attack is the first attack on SVG filters to leak data faster than the screen refresh rate, and is the first side-channel attack mounted on browsers that shows a generic method for bypassing CORS. The move from the timing side channel to a high-resolution cache side channel significantly expands the attack landscape for pixel-stealing attacks in particular, and for browser-borne privacy attacks in general. Browser vendors should ensure their filter code is resistant to cache side-channel attacks, not just traditional timing attacks.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher

Award DE200101577; an ARC Discovery Project number DP210102670; CSIRO's Data61; the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112390029 and W912CG-23-C-0022, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; the National Science Foundation under grant CNS-1954712; and gifts by Cisco and Qualcomm.

Parts of this work were undertaken while Yuval Yarom was affiliated with the University of Adelaide and with Data61, CSIRO.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE SP*, pages 699–715, 2022. doi: 10.1109/SP46214.2022.9833711.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In *USENIX Security*, pages 2801–2818, 2022. URL https://www.usenix.org/system/files/sec22-aldaya.pdf.

[3] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435, 2016. doi: 10.1145/2976749.2978353.

[4] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE SP*, pages 623–639, 2015. doi: 10.1145/3243734.3243766.

[5] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *CCS*, pages 1369–1382, 2018. doi: 10.1145/3243734.3243766.

[6] The HTTP Archive. The HTTP archive almanac. http://almanac.httparchive.org/en/2022/security.

[7] Chetan Bansal, Sören Preibusch, and Natasa Milic-Frayling. Cache timing attacks revisited: Efficient and repeatable browser history, OS and network sniffing. In *SEC*, pages 97–111, 2015.

[8] L. David Baron. :visited support allows queries into global history. https://bugzilla.mozilla.org/show_bug.cgi?id=147777, 2002.

[9] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014. doi: 10.1145/2660267.2660283.

[10] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. 2006. URL https://eprint.iacr.org/2006/052.

[11] William J. Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in HTTP. In *IET Information Security*, pages 118–126, 2018. doi: 10.1049/iet-ifs.2016.0621.

[12] Chromium Project. SVG filter timing attack. https://bugs.chromium.org/p/chromium/issues/detail?id=251711, 2013.

[13] Chromium Project. Timing attack on denormalized floating point arithmetic in SVG filters circumvents same-origin policy. https://bugs.chromium.org/p/chromium/issues/detail?id=615851, 2016.

[14] Chromium Project. Cross-origin pixel reading and history sniffing via SVG filter timing attack. https://bugs.chromium.org/p/chromium/issues/detail?id=686253, 2017.

[15] Chromium Project. Software rendering list. https://chromium.googlesource.com/chromium/src/gpu/+/refs/heads/main/config/software_rendering_list.json, 2023.

[16] Andrew Clover. CSS visited pages disclosure. https://seclists.org/bugtraq/2002/Feb/271, 2002.

[17] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *CCS*, pages 25–32, 2000. doi: 10.1145/352600.352606.

[18] The OWASP Foundation. The OWASP secure headers project. https://owasp.org/www-project-secure-headers/.

[19] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018. doi: 10.1007/s13389-016-0141-6.

[20] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018. doi: 10.1007/978-3-319-93387-0_5.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015. URL https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf.

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016. doi: 10.1007/978-3-319-40667-1_15.

[23] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011. doi: 10.1109/SP.2011.22.

[24] Scott Helme. Top 1 million analysis - june 2022. https://scotthelme.co.uk/top-1-million-analysis-june-2022/.

[25] Anxin Huang, Chen Zhu, Dewen Wu, Yi Xie, and Xiapu Luo. An adaptive method for cross-platform browser history sniffing. In *Measurements, Attacks, and Defenses for the Web Workshop*, pages 1–7, 2020. doi: 10.14722/madweb.2020.23006.

[26] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, pages 413–428, 2012. URL https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf.

[27] Artur Janc and Lukasz Olejnik. Web browser history detection as a real-world privacy threat. In *ESORICS*, pages 215–231, 2010. doi: 10.1007/978-3-642-15497-3_14.

[28] Artur Janc and Lukasz Olejnik. Feasibility and real-world implications of web browser history detection. In *Web 2.0 Security and Privacy*, 2010. URL https://www.ieee-security.org/TC/W2SP/2010/papers/p26.pdf.

[29] David A Kaplan. Optimization and amplification of cache side channel signals. In *arXiv*, 2023. doi: 10.48550/arXiv.2303.00122.

[30] Georgios Karopoulos, Dimitris Geneiatakis, and Georgios Kambourakis. Neither good nor bad: A large-scale empirical analysis of HTTP security response headers. In *TrustBus*, pages 83–95, 2021. doi: 10.1007/978-3-030-86586-3_6.

[31] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security*, 2023. URL https://www.usenix.org/system/files/usenixsecurity23-katzman.pdf.

[32] Hiroaki Kikuchi, Kota Sasa, and Yuta Shimizu. Interactive history sniffing attack with Amida lottery. In *IMIS*, pages 599–602, 2016. doi: 10.1109/IMIS.2016.109.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, pages 463–480, 2016. URL https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_kohlbrenner.pdf.

[35] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, pages 69–81, 2017. URL https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-kohlbrenner.pdf.

[36] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS*, pages 1055–1062, 2013. doi: 10.1145/2508859.2516712.

[37] Arturs Lavrenovs and F. Jesus Rubio Melon. HTTP security headers analysis of top one million websites. In *CyCon*, pages 345–370, 2018. doi: 10.23919/CYCON.2018.8405025.

[38] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, pages 707–710, 1966.

[39] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS*, pages 191–209, 2017. doi: 10.1007/978-3-319-66399-9_11.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. doi: 10.1109/SP.2015.43.

[41] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018. doi: 10.1145/3243734.3243761.

[42] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, pages 48–65, 2015. doi: 10.1007/978-3-319-26362-5_3.

[43] MDN Contributors. The <fecomponenttransfer> svg filter primitive. https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feComponentTransfer, 2022.

[44] MDN Contributors. Privacy and the :visited selector. https://developer.mozilla.org/en-US/docs/Web/CSS/Privacy_and_the_:visited_selector, 2022.

[45] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90, 2017. doi: 10.1007/978-3-319-66787-4_4.

[46] Mozilla Bug Tracker. SVG filter timing attack. https://bugzilla.mozilla.org/show_bug.cgi?id=711043, 2013.

[47] Mozilla Bug Tracker. Pixelstealing and history-stealing through floating-point timing side channel with svg filters. https://bugzilla.mozilla.org/show_bug.cgi?id=1336622, 2017.

[48] Mozilla Security Blog. Plugging the CSS history leak. https://blog.mozilla.org/security/2010/03/31/plugging-the-css-history-leak/, 2010.

[49] Keith O'Neal and Scott Yilek. Interactive history sniffing with dynamically-generated QR codes and CSS difference blending. In *WOOT*, pages 335–341, 2022. doi: 10.1109/SPW54247.2022.9833863.

[50] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418, 2015. doi: 10.1145/2810103.2813708.

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. doi: 10.1007/11605805_1.

[52] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005. URL https://www.daemonology.net/papers/htt.pdf.

[53] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920, 2021.

[54] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. Showtime: Amplifying arbitrary cpu timing side channels. In *AsiaCCS*, 2023. doi: 10.1145/3579856.3590332.

[55] Stephen Röttger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021.

[56] Jesse Ruderman. Css on a:visited can load an image and/or reveal if visitor been to a site. https://bugzilla.mozilla.org/show_bug.cgi?id=57351, 2000.

[57] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks. In *WOOT*, 2010. URL https://www.usenix.org/legacy/events/woot10/tech/full_papers/Rydstedt.pdf.

[58] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Cookies from the past: Timing server-side request processing code for history sniffing. *Digital Threats: Research and Practice*, 1(4), 2020. doi: 10.1145/3419473.

[59] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography*, pages 247–267, 2017. doi: 10.1007/978-3-319-70972-7_13.

[60] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, pages 639–656, 2019. URL https://www.usenix.org/system/files/sec19-shusterman.pdf.

[61] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, pages 2863–2880, 2021. URL https://www.usenix.org/system/files/sec21-shusterman.pdf.

[62] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *WOOT*, 2018. URL https://www.usenix.org/system/files/conference/woot18/woot18-paper-smith.pdf.

[63] Paul Stone. Pixel perfect timing attacks with HTML5. In *Black Hat*, 2013. https://media.blackhat.com/us-13/US-13-Stone-Pixel-Perfect-Timing-Attacks-with-HTML5-WP.pdf.

[64] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: Frequency, power, and temperature attacks on GPUs and ARM SoCs. *USENIX Security*, 2023. URL https://www.usenix.org/system/files/usenixsecurity23-taneja.pdf.

[65] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6, 2017. doi: 10.1145/3152701.3152706.

[66] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE SP*, pages 39–54, 2019. doi: 10.1109/SP.2019.00042.

[67] Frederick M Waltz and John WV Miller. Efficient algorithm for Gaussian blur using finite-state machines. In *Machine Vision Systems for Inspection and Metrology VII*, pages 334–341, 1998. doi: 10.1117/12.326976.

[68] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In *IEEE SP*, 2023. doi: 10.1109/SP46215.2023.10179326.

[69] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. A practical attack to de-anonymize social network users. In *IEEE SP*, pages 223–238, 2010. doi: 10.1109/SP.2010.21.

[70] World Wide Web Consortium (W3C). Filter effects module level 1. https://drafts.fxtf.org/filter-effects, 2019.

[71] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. doi: 10.1109/SP.2019.00004.

[72] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. URL https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf.

A Modelling Prime+Scope False-Negatives

In this appendix we analyze Prime+Scope as a primitive to measure cache activity in our pixel stealing attacks. We find that Prime+Scope is significantly more sensitive to some kinds of noise than Prime+Probe. Specifically, Prime+Scope is sensitive to noise which causes a cache miss to be incorrectly classified as a cache hit. Such a situation can leave Prime+Scope in a state where it is unable to detect cache events until the next time the cache state is refreshed by a *prime* operation.

Prime+Scope. The prime operation of Prime+Scope performs two functions on a target cache set. First, similar to Prime+Probe, it clears victim cache lines from the set. Second, it sets a so-called *scope* line as the next line to be evicted from the target set. That is, if some other line needs to be cached in the set, then the scope line is evicted. Prime+Scope abuses this fact to detect victim memory accesses by only measuring access times to the scope line. If a cache hit to the scope line is detected, then the state of the cache remains unchanged and the attack can measure the access time to the scope line again. On the other hand, if a cache miss is detected, then this indicates a victim memory access. In this case, the state of the cache has changed and another prime operation is required to refresh the desired cache state.

However, if a cache miss is incorrectly categorized as a cache hit, then the prime operation will not be performed and Prime+Scope will not be able to detect any future victim memory accesses.

Model Description. To investigate this behavior, we model Prime+Scope as an abstract cache attack that detects cache events without disrupting the cache state. If the attack performs the prime operation, then the cache transitions to the *primed* state. If a victim accesses memory, then the cache transitions to the *dirty* state. The attack can only detect victim memory accesses by detecting the transition from the primed state to the dirty state. If the cache is in the dirty state when a victim memory access occurs, then the attack will not be able to detect it. After each detected victim memory access, or after w samples, the attack performs the prime operation setting the cache back to the primed state.

We limit ourselves to the following outcomes for each measurement: *false negative* or *some other outcome*. We do this so that we can model the attack using a weighted coin with finite trials and ignoring all other outcomes has no effect on measuring the effect of false negatives. We calculate the probability of flipping tails, the probability of a *false negative*, in the following way. Let d be the probability to incorrectly measure a victim memory access and v be the probability that the victim accesses memory on any given sample. The false negative rate is then dv, the probability that both events happen, the victim accesses memory and the attack incorrectly measures it. In the coin flipping model dv is the probability to flip tails.

Analytical Solution. Next we ask *What is the average number of coin flips until we flip tails?* this corresponds to *What is the average number of samples until we have a false negative?*. We answer these questions using a geometric distribution given x the number of trials and dv the probability of a false negative we find n_x the average number of coin flips (samples) until we flip tails (have a false negative). The portion of samples that are observed on average is $\frac{n_x}{x}$.

$$n_x = \sum_{i=1}^{x} (1-dv)^i = \frac{(dv-1)((1-dv)^x - 1)}{dv}$$

.

We extend this solution to attacks that prime the cache every w samples by noting that it is possible to break any run of s samples into $\lfloor \frac{s}{w} \rfloor$ identical runs of length w and one run of length $s \mod w$. We can calculate the average number of samples that can be observed by substituting w and $s \mod w$ into the equation above to yield n_w and $n_{s \mod w}$. These values are then combined to find n_s the average number of observed samples $n_s = \lfloor \frac{s}{w} \rfloor n_w + n_{s \mod w}$.

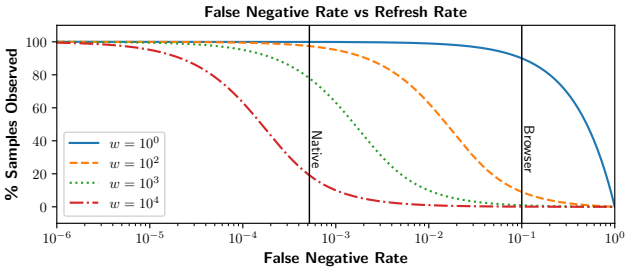


Figure 13: Comparison between False Negative Rate and Refresh Rate. Refresh Rate is defined as w the number of samples until the state of the cache is refreshed by the attack. False Negative Rate is defined as dv the probability that the attack fails to detect a victim memory access. captured.

Results. Figure 13 compares the refresh rate w with the probability of a false negative dv. The vertical lines marked *Native* and *Browser* show measured values for d in the two respective environments. Note that we do not know v and as

such these lines show the maximum expected false negative rate for these environments.

The figure shows that a false negative rate that is $10\times$ larger requires a refresh interval that is $10\times$ smaller to achieve a similar accuracy. The opposite is also true if the false negative rate is $10\times$ smaller than a refresh interval that is $10\times$ larger can achieve a similar accuracy. Importantly, this property holds for d and v independently. If the victim accesses memory $10\times$ more frequently, then the refresh interval must be $10\times$ smaller to achieve a similar accuracy.

Finally, the figure also provides a comparison to attacks that prime the cache for each sample ($w = 1$) such as Prime+Probe. Note that in this case the probability to observe a sample is simply the probability to not observe a false negative ie. The portion of samples observed is $1 - dv$.

B GPU Filters

Table 4 provides a list of filter elements in Firefox that do not have a GPU implementation.

Filter Element	GPU Support
feBlend	✓
feColorMatrix	✓
feComponentTransfer	✓
feComposite	✓
feConvolveMatrix	✗
feDiffuseLighting	✗
feDisplacementMap	✗
feFlood	✓
feGaussianBlur	✗
feImage	—
feMerge	—
feMorphology	✗
feOffset	✓
feSpecularLighting	✗
feTile	✗
feTurbulence	✗

Table 4: List of filter elements. Filter elements marked ✓ have a GPU implementation and elements marked ✗ do not. feImage and feMerge use a separate system that is out of scope for this work and are marked —.