Cross-origin pixel stealing: Timing attacks using CSS filters

Robert Kotcher, Yutong Pei, Pranjal Jumde*
Carnegie Mellon University
{rkotcher, ypei, pjumde}@andrew.cmu.edu

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

ABSTRACT

Timing attacks rely on systems taking varying amounts of time to process different input values. This is usually the result of either conditional branching in code or differences in input size. Using CSS default filters, we have discovered a variety of timing attacks that work in multiple browsers and devices. The first attack exploits differences in time taken to render various DOM trees. This knowledge can be used to determine boolean values such as whether or not a user has an account with a particular website. Second, we introduce pixel stealing. Pixel stealing attacks can be used to sniff user history and read text tokens.

Categories and Subject Descriptors

K.4.4 [**Computers and Society**]: Electronic Commerce Security; K.4.1 [**Computers and Society**]: Public Policy Issues - Privacy

General Terms

Security, Experimentation

Keywords

CSS-Filters, CSS-Shaders, OpenGL ES, Timing Attacks, Privacy

1. INTRODUCTION

In this section we discuss CSS filters and describe how web content is rendered in browsers. Finally we introduce other timing attacks to provide a basis for our own work.

1.1 CSS filter specification

Cascading Style Sheets, CSS, is a declarative styling language that applies and prioritizes styling rules to elements of a web document (also known as Document Object Model elements or "DOM elements"). Some of the latest members of the CSS feature family, filter effects, are descendants of the W3C SVG specification[11] that allow developers to apply personalized style to arbitrary web content. Filters are, as of recently, GPU accelerated [12, 22] giving developers a robust new way to stylize arbitrary DOM elements through simple CSS attributes. While filters are not implemented in all browsers, support does exist in WebKit browsers including Safari, and mobile browsers such as iOS Safari, Blackberry, and Chrome for Android. Google Chrome is based on Blink which is a fork of WebKit in the area of graphics [9, 11].

Figure 1: Image of the Stanford bunny with a CSS blur filter

There are three types of CSS filters. The first, built-in filters, are canned effects that are parameterized and can be specified directly as an attribute-value pair in a CSS class. Figure 1 shows the Stanford bunny with a blur filter applied to the right panel.

The second type, SVG filters, are similar effects that can be applied to SVG content.

The final type, CSS shaders, was proposed by Adobe to the W3C FX task force[18] in October, 2011 as a complement to CSS filters. Shaders are now referred to as CSS custom filter effects. The CSS custom filter effects allow web developers to write OpenGL ES shader programs that operate on arbitrary web content. Content is applied either through separate text files specified with a url or directly through HTML5 tags[18]. The proposed feature is now a part of the CSS Filter Effect draft[1], and prototype implementation has started and is available in developer builds. Because custom filters are experimental, they must first be enabled by entering chrome://flags in the URL bar.

*These author contributed equally to this paper

CCS'13, November 4–8, 2013, Berlin, Germany.
ACM 978-1-4503-2477-9/13/11.
http://dx.doi.org/10.1145/2508859.2516712

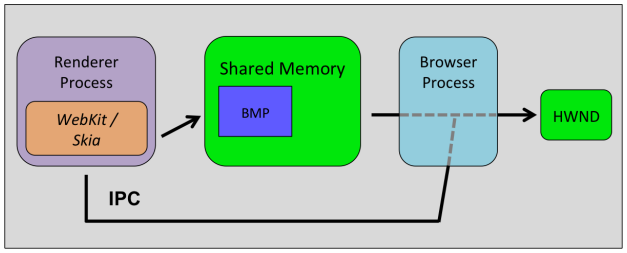

Figure 2: CPU rendering path in WebKit

1.2 Rendering DOM content

DOM content is transformed to a rasterized image in several steps. We summarize CPU and GPU rendering in WebKit in this section. Since the root cause of our timing attack is not clear, we provide this section as a possible starting point for further research. Information in this section was consolidated from the Chromium Design Documents [5, 15].

As of this paper, the majority of WebKit's rendering algorithms are performed on a single process [15] which it shares with the sandboxed web application. Figure 2 shows how content is rendered using the CPU.

DOM elements are stored internally as nodes of a tree structure whose root is the Document node for all instances. Each node of a corresponding tree structure of RenderObjects defines the physical layout of a unique DOM element. Sets of RenderObjects are mapped to elements of a RenderLayer tree. While DOM nodes model document data, the RenderObjects and RenderLayers provide hierarchical views that ultimately talk to the GraphicsContext, a wrapper for Google's Skia Graphics engine [6, 15, 20]. A distinct RenderLayer can be created for a RenderObject if any of the following are true:

- It is the root element for the page
- It has explicit CSS properties (relative, absolute, or transform)
- It is transparent
- It has an overflow, mask or reflection
- It has a CSS filter
- It corresponds to a canvas or video element

We found that WebKit reads a RenderLayer tree structure from root to leaves, using an algorithm similar to Painter's Algorithm; i.e. WebKit traverses the entire tree of RenderElements in order to resolve pixel depth[15] and create a bitmap by painting elements from furthest away to nearest. This approach to depth resolution is inefficient [7, 16] and in this paper we describe a timing attack on user privacy that is possible because of this. Only after the RenderLayers have finished providing pixels to a buffer in shared memory can a separate browser process take over rendering to the correct tab[15] or iframe element.

The GPU can be used to render content more quickly. While a description of GPU rendering mechanisms is out of the scope of this paper, its important to know that additional structures are built for GPU rendering and may also account for timing differences.

1.3 Timing attacks

Timing attacks work when a system takes different amounts of time to process different inputs [4, 10, 14, 19]. Vulnerabilities tend to occur as a result of inherent behavior, not "buggy code". Scenarios allowing for successful timing attacks have been found in performance optimized code [2, 14] browser caching [4, 10], DNS caching [10], CBC encryption, and RAM cache hits [14]. By timing these properties, an attacker can gain insight into a large amount of information.

Timing attacks are usually regarded as an invasion of a user's privacy [4, 10]. Exposing Private Information by Timing Web Applications [4] by Bortz et al defines two families of web timing attacks:

A *Direct timing attack* can be used to determine boolean values such as a user's login status or content data size such as the relative size of a shopping cart [4]. A direct timing attack is carried out by the attacker's browser against a victim web server.

In contrast to direct timing attacks, *cross-site timing* attacks rely on a user to visit, and remain on, a malicious page while a cross-site exchange is made. Our proposals are all cross-site timing attacks.

2. RELEVANT SECURITY TOPICS

New features usually arrive with their own set of security considerations, and CSS filters are no exception. Much work has already gone into safe integration of CSS filters into web browsers. We first present a threat model for our timing attacks. Then we discuss security in CSS custom filters, browsers, and browser policies.

2.1 Threat model

The threat model for our timing attacks is a web attacker who operates a malicious domain. The attacker is either able to provide content that keeps a user engaged on a webpage for an extended period of time, or is able to open another window in front of the window being attacked.

2.2 Security in CSS custom filters

Our attempts to look for new timing channels in browser rendering engines stemmed from an interesting attack proposal by Adam Barth [2], who showed that simply removing access to sensitive information in custom filters is not enough to protect web pages from malicious filter developers. He suggested that a fragment shader could contain code similar to the following:

Listing 1: Abstract timing attack on color

```
1 if (pixelColor != 0x000) {
2     for (int i = 0; i < MAX_INT; i++) {;}
3 }
```

As a result we can determine whether or not pixelColor is black by observing the amount of time taken for the shader to output a final pixel value. There are many ways an attacker could use this approach to steal an unsettling amount of sensitive information.

W3C has now recommended that shaders restrict all direct access to rendered content [19]. Some researchers suspected that custom shaders are still vulnerable. They did not have a proof-of-concept to support their claim[2].

2.3 Security in Browsers

While we were able to get our attack to work in several browsers, Chromium was the one we spent the most time with. This section is about security in browsers, but focuses on Chromium's implementation specifically. The two concepts in this section are relevant to our work and include flow of fragment data and multi-process browser architecture.

1. **Flow of fragment data.** One security feature that makes GPU-accelerated graphics rendering difficult to measure is a one-directional flow of fragment data. The application's control over hardware-accelerated rendering is revoked as soon as commands and data are passed to the *command buffer*. There is no mechanism that can be used to retrieve GPU output.

2. **Multi-process architecture.** This is another important feature is Chromium's multi-process model. By separating the browser kernel from rendering engine instances, Chromium wants to prevent the attacker from installing persistent malware/keyloggers and stealing local files [3, 5, 15]. These measures are effective at what they are designed to do. They are unable, however, to hide timing differences in the rendering engine. If anything, isolation of the rendering engine makes timing differences more noticeable.

2.4 Security in Web Policies

It turns out that CSS filters are indifferent as to whether or not the actual rendered content is cross-origin[18], and we believe that browser policies regarding filters should only be part of a larger effort to contain cross-origin information. CSS filters still violate Same-Origin Policy (SOP) because they access cross-origin content when X-Frame-Options are not used. As a result, setting X-Frame-Options to Deny is the only way to ensure timing attacks will fail to work on a framed web page.

3. OUR PROPOSED TIMING ATTACKS

In this section we demonstrate how cross-site timing attacks may be used to invade a user's privacy and to steal sensitive data. First we explain our approach to collecting timing data. Second we show how timing data can be used to determine a user's login status on arbitrary websites. Finally we present pixel stealing and how it can be used to sniff user history and steal cross-origin text tokens.

3.1 Collecting data using requestAnimationFrame

We found that the entire rendering process can be timed through *requestAnimationFrame*[8], which allows a web page to provide a callback for the browser to call when an animation frame is available for painting. The function *requestAnimationFrame* is implemented in Gecko and WebKit, and the code below demonstrates how it can be used to measure the average framerate of a browser window.

Listing 2: Measuring Framerate with javascript

```
FramerateFinder = function() {
  var tFrames = 0,
      avgFramerate = 0,
      startTime;

  var updateAvgFramerate = function() {
    var elapsed = timer.now() - startTime;
    avgFramerate = tFrames / elapsed;
    avgFramerate = avgFramerate * 1000;
  }

  var step = function() {
    tFrames += 1;
    requestAnimationFrame(step);
  }

  var start = function() {
    startTime = timer.now();
    step();
  }
}
```

The function *start* begins a series of calls to *step*, which increases the total frame count by one and passes itself to *requestAnimationFrame*. The total frame count is incremented each frame, and the current average framerate can be calculated with *updateAvgFramerate* at any time.

3.2 Determining a user's login status

The simplest attack we implemented is to detect a user's login status. While simple, it is important to protect against attacks such as this because they can reveal a significant amount of insight about a user's online behavior. For instance we could use this data to quickly determine which web users are T-Mobile customers.

For many websites, a user with an authentication cookie will be directed to a personalized home page, while other users are sent to a 'catch-all' home page. Normally, a login page has a different DOM structure than that of a user's homepage. If a CSS filter is applied to documents with different tree structures, we found the corresponding frame rates to be different since WebKit always traverses the entire RenderObject tree when building a RenderLayer tree. Assume example.com/home is the user's homepage and example.com/login is the login page.

An attacker could determine whether or not a user has authentication cookies for a website in the following way:

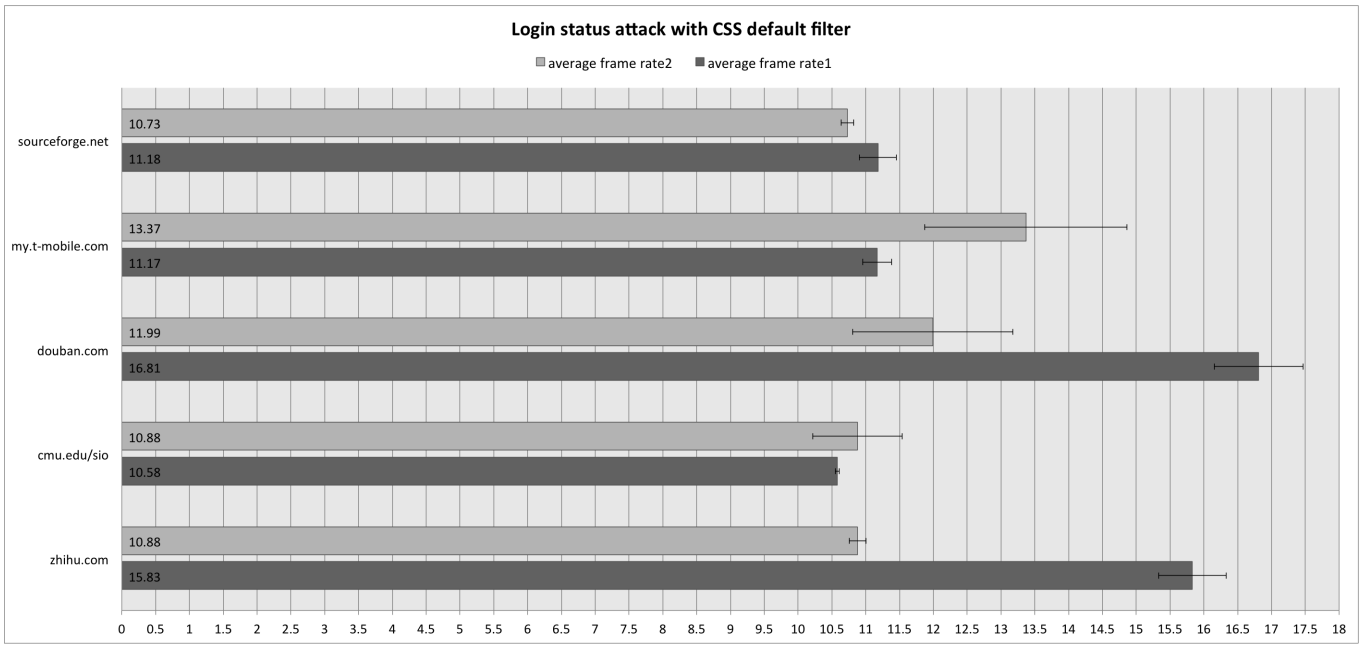

Figure 3: We performed the login attack on five high-traffic websites. (hardware: 2012 MacbookPro Retina with Intel Core i7 3720QM / Nvidia GT650M; browser: Google Chrome).

1. **Malicious reference page created.** The attacker copies the example.com login page and all referenced content from the victim's web server at example.com/login.html to a location on his server as a "reference page", and the attacker saves it to www.attacker.com/reference.html.
2. **Dummy page is created.** The attacker creates a dummy page with <iframe id="attack-iframe" src="example.com">. This page resides at www.attacker.com/attack.html. The victim must browse to this page for the attack to occur.
3. **Victim is engaged.** A victim using a vulnerable browser visits www.attacker.com/attack.html, and is engaged for a short period of time.
4. **Shaders are applied.** A CSS built-in filter is rapidly re-applied to the iframe, causing the framed page to rerender without refreshing. Example code for this step is in Listing 3.
5. **Victim's data is collected.** After some time, the average framerate is determined.
6. **Reference page is timed.** The iframe loads the reference page on the attacker's server, located at www.attacker.com/reference.html, and the same process is repeated for the new page. Data must be collected from both pages since timing between devices may differ.
7. **Data is analyzed.** The total frame counts for both pages are returned to the attacker's server to be compared.

Listing 3: Toggle filter

```
<style>
  .filter{
    -webkit-filter: blur(10px);
    -moz-filter: blur(10px);
  }
</style>

<script>
  function alternate(){
    $("#attack-iframe")
      .toggleClass("filter");
  }
  setInterval(alternate, time);
</script>
```

We found that this attack also works in Firefox by using an *svg-image-blur* effect instead of *webkit-filter*, since Firefox allows a user to apply SVG filters to DOM elements. As with most web timing attacks, timing data is an approximation of the true runtime of a process. Our attack is less noisy than typical web application timing attack[4] though, since there is no network latency or packet-loss involved.

The results in Figure 3 show how the relative size of a page can be determined with CSS default filters. Notice that the page when the user is logged in produces a different average framerate than the homepage when the user is not logged in because their DOM trees are different.

3.3 Pixel Stealing with CSS filters

Next we describe a general technique that we have discovered can be used to read a field of arbitrary pixels from the user's browser window. We fill the screen with a single color and examine the way in which the browser window's average framerate changes by reading it with *requestAnimationFrame*.

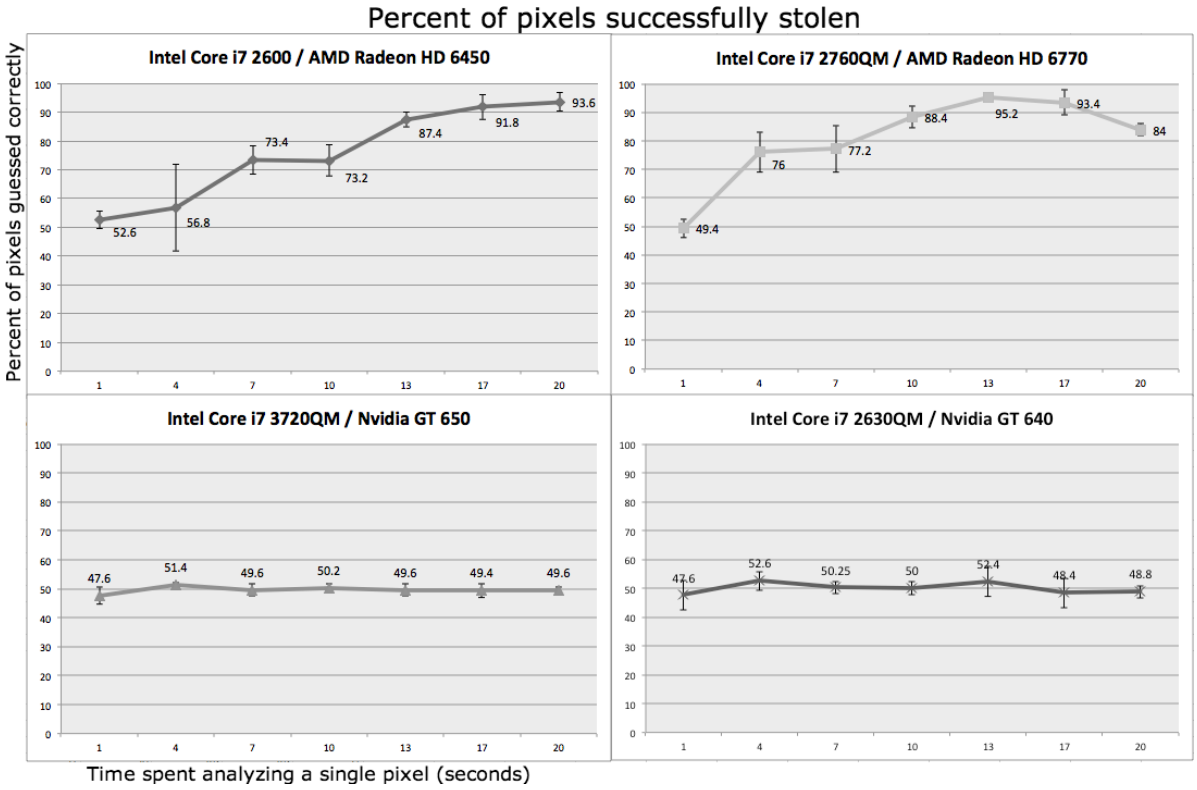

Figure 4: Using the same test that produced the bitmaps shown in Figure 5, we determined how accuracy differed across different devices.

3.3.1 *Focusing on a Single Pixel without anti-aliasing*

We found that enlarging the pixel to the size of the user's screen exaggerated timing differences. This section describes our technique.

Listing 4: Enlarging a single pixel

```
#malicious-iframe {
    overflow: hidden;
    width: 1px;
    height: 1px;
    margin-top: 0px;
    margin-left: 0px;
}

#pixel-container {
    top: calc(50% - 1px);
    left: calc(50% - 1px);
    width: 1px;
    height: 1px;
}

#pixel-container-container {
    width: 999px;
    height: 999px;
    -webkit-filter: custom(url(enlarge.vs)
         mix(url(enlarge.fs)), 4 4);
}
```

To enlarge a pixel to the size of the screen without anti-aliasing, we first apply the style rules for #malicious-iframe to a borderless iframe. The iframe contains the webpage under attack. By setting margin-left and margin-top to values less than or equal to zero, different offsets onto the victim page can be achieved.

Next we position the iframe inside a div of class #pixel-container. This div is centered inside its parent element, #pixel-container-container. This class defines a CSS filter that scales the pixel-sized iframe to a much bigger size. The width and height are odd-valued to ensure that the single-pixel iframe will be in the center.

3.3.2 *Pixel Stealing*

We use the above mechanism to traverse a bitmap of pixels by setting margin-top and margin-left in the malicious iframe. Our attack performs arbitrary transformations on the scaled pixel, determines the average framerate during the transformations, and interprets the the resulting value for each pixel. A pixel stealing attack occurs in the following way:

1. **Expand a single pixel.** The attacker builds a mechanism that can expand a pixel to the size of the user agent's screen.
2. **Victim page is framed.** The attacker frames a website that has neglected to use X-Frame-Options.
3. **Vicitm visits malicious page.** A victim user visits the attacker's malicious web page, and is tricked into

remaining on the page for the duration of the attack.

4. **Page is traversed.** In an attack, a malicious website may want to traverse each pixel in a target region of the intended website or in a target region of its source code. An arbitrary transformation is repeatedly performed on each pixel.

5. **Average framerate is captured.** *RequestAnimationFrame* determines the average framerate on the browser window for each target pixel.

6. **Data is interpreted.** An array of pixel measurements are sent to the attacker's server to be interpreted.

Collecting and interpreting pixel data turned out to be an interesting problem by itself, and we tried several methods that helped us understand the set of values returned by the attack. The most successful attempt involved writing code with a JQueryUI slider that could change the threshold that separated black and white pixels, however writing a script to interpret the pixel data for us proved to be more than sufficient. The script interpreted values by deciding what thresholds should be used to distinguish black and white pixel framerates. Using a canvas element we could visually examine the pixel values.

3.3.3 Our Results

We initially tested our pixel stealing idea by running black and white pixels through shaders with arbitrary transformation matrices. Our intuition was if there existed two colors that could create distinct timing channels, black and white would be the most likely candidates. Our intuition turned out to be correct.

A proof-of-concept attack involved recreating a 10×10 bitmap of black and white pixels to determine the timing consistencies of a user's device. A higher percent of correctly-guessed pixels implies a more vulnerable user agent. Results from this initial test indicated that stealing cross-origin pixels could be achieved with high enough accuracy to make pixel stealing attacks practical. We conducted tests that measured each pixel's color for 4, 12, and 20 seconds, and the results are shown in Figure 5. Compare our results with the actual bitmap in (d).

In this test, we counted the number of pixels that were guessed correctly out of 100. If our attack was returning random values, we would expect about 50 pixels to be correct regardless of time spent per pixel. The chance that exactly half of the pixels are guessed correctly is 50%. The chance that more than 60% of a bitmap's pixels are chosen correctly on any given test is just 6%. Figure 4 shows accuracy for various devices. Half of the setups we tried were vulnerable.

3.3.4 Possible Attack Scenarios in Practice

Once a bitmap of pixel colors can be determined over https, attack possibilities are endless. The most interesting attack we implemented involves stealing tokens of cross-origin text. We were able to read a fake token that we retrieved across origins. This token was stolen from a machine with an AMD Radeon HD 6770 and Intel Core i7:

We also used our pixel stealing attack to implement history sniffing. Until recently, a user's history could be determined

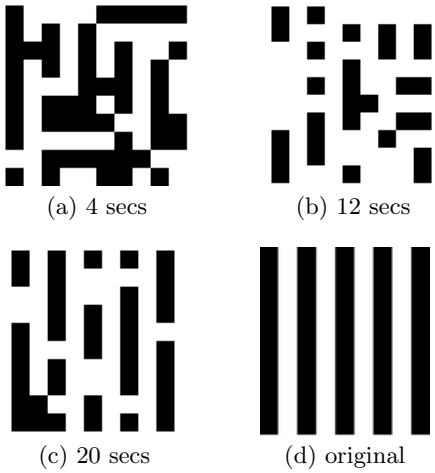

Figure 5: The original pixel stealing tests (a), (b), and (c) show how accuracy improves as the amount of time (4, 12, and 20 seconds respectively) spent per pixel is increased. (d) shows the original bitmap.

by simply adding a link to a page and determining its color by calling *getComputedStyle()*[13] since visited links have different colors than non-visited links. Browsers will now lie [21] if this function is called on a link.

The following attack can be used to determine whether or not a user has visited a particular website:

1. **Victim visits malicious page.** The malicious page initializes with a block of ascii text (say, ascii value 219) that hrefs to a website that is known to not exist.

2. **Link is expanded.** A pixel from the link is expanded to the size of the user's screen.

3. **Data is collected.** The malicious web page allows a sufficient amount of time to pass while measuring the page framerate using *requestAnimationFrame*.

4. **Process is repeated.** Steps 2-4 are repeated two more times. The first is for this page, which will return results for a URL that we know the user has visited. The second is for a URL we are curious about.

5. **Data is analyzed.** We can determine if the user visited the victim URL by comparing it with the framerate of our two test URLs.

By using the pixel stealing technique described above, it is possible to determine whether or not a user has visited a website by analyzing a single pixel.

3.3.5 Complications and Solutions

We have found that the best results can be obtained when the input pixel field is restricted to black and white only. We combined several filter effects to achieve a close estimate of a black and white transform. We ended up using the following filter combination:

(a) Original Text

(b) Text after applying filters

(c) Stolen Text

Figure 6: (a) shows the original bitmap of anti-aliased pixels in Google Chrome for a text token. We applied filters to approximate a black and white version of the token, shown in (b). Finally, (c) shows the cross-origin token of text that we stole with our timing attack.

Listing 5: Filter effects to achieve a close estimate of a black and white transform

```
.black-and-white {
-webkit-filter: saturate(0%)
                grayscale(100%)
                brightness(69%)
                contrast(100%);
}
```

Notice in Figure 7 that without the use of these filter effects, non-black pixels are usually interpreted as white. This sample of cross-origin HTML text is easily recognized as the letter "h" in Figure 7, but without default filters, many letters with curves such as "S" and "R" were nearly impossible to read.

A final challenge we faced with this attack was finding a reasonable amount of time to spend on each pixel. It is important that all pixels are read before the user closes his or her browser window. To get a better understanding of the accuracy achieved as a result of various amounts of time spent per pixel on several graphics cards, see Figure 4. While history sniffing or stealing small text tokens is possible in practice, stealing medium-sized images or large tokens may not be.

Under what conditions do the attacks fail? We determined that running shaders in background tabs produces undesirable performance, and will cause an attack to fail. Running the attack in a background window will have the same performance results and an attack in the foreground. Running the attack on a div or iframe element whose opacity equals *0.0* will not work, but covering it with another DOM does work.

Figure 7: Attempts to read text without converting to black and white made letter curvature difficult to capture.

4. CONCLUSIONS AND FUTURE WORK

In this paper we show that timing attacks using CSS filters can reveal sensitive information such as text tokens. Our work uses both default and custom filters to exploit timing channels in rendering engines of various browsers. A paper that cites the original version of this paper shows that the timing attacks can also be performed with SVG filters [17].

An easy solution would be to simply place shaders and filters under the restrictions of the Same-Origin Policy, but this would entirely defeat the purpose of these features. This would also likely be covering up a larger problem that is present across multiple browsers. Creating awareness in the security community seems like the best way to proceed.

5. ACKNOWLEDGEMENTS

We would like to thank Eric Chen and Lin-Shung Huang for their guidance and support to our research.

6. REFERENCES

[1] Adobe. Css shaders. http://www.adobe.com/devnet/html5/articles/css-shaders.html.

[2] A. Barth. Adam barth's proposal. http://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html.

[3] A. Barth, C. Jackson, C. Reis, and T. Team. The security architecture of the chromium browser, 2008.

[4] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

[5] Chromium. Gpu command buffer. http://www.chromium.org/developers/design-documents/gpu-command-buffer.

[6] Chromium. Graphics and skia. http://www.chromium.org/developers/design-documents/graphics-and-skia.

[7] CMU. Spatial data structures. http://www.cs.cmu.edu/afs/cs/academic/class/15462-f12/www/lec_slides/lec13.pdf.

[8] R. Crawfis. Mozilla window.requestanimationframe. https://developer.mozilla.org/en-US/docs/DOM/window.requestAnimationFrame.

[9] A. Deveria. Can i use css filter effects? http://caniuse.com/css-filters.

[10] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32. ACM, 2000.

[11] HTML5Rocks. Catch-all for html5 rocks website. http://updates.html5rocks.com.

[12] R. Hudea, R. Cabanier, and V. Hardy. enriching the web with css filters.

[13] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 737–744. ACM, 2006.

[14] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[15] V. Kokkevis. Gpu accelerated compositing in chrome. http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.

[16] I. LiTH. Painter's algorithm. http://www.computer-graphics.se/TSBK07-files/PDF12/6b.pdf.

[17] P. Stone. Pixel perfect timing attacks with html5. http://www.contextis.com/files/Browser_Timing_Attacks.pdf.

[18] W3. Css shader proposal. https://dvcs.w3.org/hg/FXTF/raw-file/tip/custom/index.html.

[19] W3. Shader security. http://www.w3.org/Graphics/fx/wiki/CSS_Shaders_Security.

[20] Webkit. Accelerated rendering and compositing. http://trac.webkit.org/wiki/Accelerated%20rendering%20and%20compositing.

[21] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 147–161. IEEE, 2011.

[22] S. White. Accelerated css filters landed in chromium. http://blog.chromium.org/2012/06/accelerated-css-filters-landed-in.html.