Rendering Contention Channel Made Practical in Web Browsers

Shujiang Wu[†], *Jianjia Yu*[†], *Min Yang*[‡], *and Yinzhi Cao*[†][*]
[†]*Johns Hopkins University*
[‡]*Fudan University*

Abstract

Browser rendering utilizes hardware resources shared within and across browsers to display web contents, thus inevitably being vulnerable to side channel attacks. Prior works have studied rendering side channels that are caused by rendering time differences of one frame, such as URL color change. However, it still remains unclear how rendering contentions play a role in side-channel attacks and covert communications.

In this paper, we design a novel rendering contention channel. Specifically, we stress the browser's rendering resource with stable, self-adjustable pressure and measure the time taken to render a sequence of frames. The measured time sequence is further used to infer any co-rendering event of the browser.

To better understand the channel, we study its cause via a method called single variable testing. That is, we keep all variables the same but only change one to test whether the changed variable contributes to the contention. Our results show that CPU, GPU and screen buffer are all part of the contention.

To demonstrate the channel's feasibility, we design and implement a prototype, open-source framework, called SIDER, to launch four attacks using the rendering contention channel, which are (i) cross-browser, cross-mode cookie synchronization, (ii) history sniffing, (iii) website fingerprinting, and (iv) keystroke logging. Our evaluation shows the effectiveness and feasibility of all four attacks.

1 Introduction

1.1 Rendering Contention Channel

Rendering is an important component of modern web browsers, which converts raw text-based data from the Internet, e.g., HTML and images, to something displayable on the computer screen. At the high-level, the operating system (OS) provides rendering as abstract resources to web browsers via libraries like DirectX and OpenGL; at the low-level, the abstracted rendering resource are broken down into different hardware resources such as CPU, GPU, and screen buffer. No matter at high- or low-level, rendering resources are shared by all processes running on the same OS and web frames on the same browser, thus inevitably being vulnerable to side channels.

Prior works—such as Stone [54], Smith et al. [51], and Huang et al. [23]—have showed that an adversary can measure a particular, microscale rendering event, such as a link color change and an SVG filter effect, happening in just one rendering frame to infer a cross-origin secret. However, despite their success, it remains unclear how the contentions on rendering, a scarce resource provided by the OS, can be used for side-channel attacks and covert communications.

In this paper, we design a novel rendering contention channel, which stresses the rendering resources with stable, self-adjustable pressure from a browser and measures the time taken to render a sequence of frames. The measured time sequence is then used to infer any co-rendering events.

Because the channel is less known to the research community, we study it using a method, called Single Variable Testing, to better understand the cause of the channel. The method only changes the pressure on one single variable, e.g., GPU, CPU and screen buffer, during the rendering pipeline, but keeps all others unchanged to measure the Signal-to-Noise Ratio (SNR). If the SNR changes together with the tested variable, we consider that the contention on that variable contributes to the channel.

The results show that GPU, CPU and screen buffer all contributes to the channel. The breakdown is actually complicated though, which depends on different configurations. For example, hardware rendering has all three variables involved, but software rendering only has CPU and screen buffer contention because GPU is not used in the rendering process. This further demonstrates the necessity in abstracting the channel as rendering contention.

[*]Dr. Yinzhi Cao is the corresponding author.

1.2 Rendering Contention Channel Attacks

One important research question, besides the causes for the channel, is what attacks we can launch using the channel. Here we illustrate three example co-rendering events as targets and four attacks using this channel.

First, we describe a client-side covert communication between different browsers (e.g., Safari and Chrome) and modes (e.g., normal and incognito) where the co-rendering event is controlled by the sender. Specifically, the sender modulates target signal by pausing and continuing a rendering event as zeros and ones and the receiver observes the rendering workload change to de-modulate the signal. Such a convert communication can be used to synchronize cookies. Then, it can either deliver targeted ads for third-party tracking websites like DoubleClick or limit the number of free articles for news websites like NYTimes.

Second, let us consider webpage rendering as a co-rendering event. Modern browsers adopt a technique, called incremental rendering [5], to accelerate rendering and show rendered contents to users as soon as possible. At the same time, this also leads to two types of attacks, (i) history sniffing and (ii) website fingerprinting, as we discussed below:

- History sniffing attack is possible because incremental rendering groups cached contents together, making the rendering of a visited website different from unvisited. Such an attack is harder to defend against when compared with prior history sniffing attacks like those [51, 54] relying on the rendering of link color, because the slow-down of the entire page rendering will significantly hamper user experience.
- Website fingerprinting attack is possible because incremental rendering also make the renderings of different web pages unique. Such a website fingerprinting attack is complementary to many existing website fingerprinting attacks [19, 49] relying on side-channels unrelated to rendering contention.

Lastly, we consider the co-rendering event as the rendering of a small area of a webpage, such as a `div` tag. Modern web search engines like Google all support autocomplete to give users real-time suggestions during typing, or in other words, render a new `div` element. Therefore, an adversary can infer what the user types from the appearance timestamp of each new `div` element. Note that this attack is the weakest among all four because the rendering area is small—we include it for the completeness in describing the rendering channel.

1.3 Rendering Contention Framework

While all four attacks are theoretically possible on the rendering contention channel, the design and implementation of these attacks in real-world face one major challenge, i.e., the high noise level. Such noise comes from difference sources, such as browser-introduce jitters and other rendering tasks.

In this paper, we propose to build a framework, called SIDER, to launch attacks using the rendering contention channel. One important task of SIDER is to denoise the signal from the rendering contention channel. Specifically, our observation is that the over-time rendering pattern as a whole—despite a few abnormal data points—contains the semantics of the target co-rendering event. Therefore, SIDER smoothes out and normalizes the rendering pattern using a sliding window and then adopts a distance calculation considering data shifting and sudden, high-value noises.

Another important task of SIDER is to compare the denoised target signal with a reference group. This is useful because although SIDER often has one chance to run the target event, SIDER can run multiple baseline rendering events. Take the history sniffing attack for example. SIDER can only load the webpage once before it is cached, but it can load the webpage multiple times to obtain the patterns for cached webpage. Specifically, we proposed two algorithms for this: (i) a max-min algorithm designed by us and (ii) a DNN-based algorithm. The former is used online when the reference group size is small, e.g., in history sniffing. The insight is that if the minimum distance between the target and the reference group is larger than the maximum distance among samples within the group, the target is an outlier. The latter is used offline when the reference group size is large, e.g., in website history sniffing. We particularly design the DNN architecture so that it can support multiple side channels and varied length of input data.

To facilitate open science, we have made our implementation open-source at this anonymous repository (https://github.com/renderingsidechannelattacks/rendersidechannelattacks). We also release our dataset together with the open-source code in the aforementioned repository. For those who are interested in our attack, a demo can be found at this URL (http://www.renderingsidechannelattacks.com:8080/).

2 Related Work

In this section, we discuss existing attacks and defenses.

2.1 Existing Side- or Covert-channels

Side-channel attacks [25, 29, 38, 65, 66] are a well-studied problem across different platforms. Researchers have studied browser-level side channels for a long time including but not limited to lower-level caching attacks [21, 45], performance-based browser type and version inference [41, 42], document's visual content inference [31], and script and video size inference [55, 56]. We now describe them based on four attacks.

- *Cross-browser cookie synchronization.* We are unaware of existing works that can achieve direct client-side cross-browser cookie synchronization. The closest work is cross-browser fingerprinting [13], which can restore client-side cookies based on the same fingerprint. However, this restoration needs server supports and introduces many false positives due to fingerprint collision.
- *History sniffing attack.* The earliest history sniffing attack from Felten et al. [19] shows that the loading time of

Table 1: A high-level comparison of the work with a representative selection of other existing side channels.

Work	Side Channel	Adversary	Attack Type				Sampling Rate	Avg attack time
			Cross-browser Cookie Sync	History Sniffing	Website Fingerprinting	Keystroke Logging		
Lifshit et al. [36]	Power consumption	External hardware	✗	✗	outside browser	✓	1000 Hz	≈10 s
Oren et al. [45], Shusterman et al. [49]	Last-level cache	Cross-origin page	✗	✗	same-browser*	✓**	10–500 Hz	≈30 s
Felten et al. [19]	Page loading	Cross-origin page	✗	✓	✗	✗	N/A	≈3 s
Stone [54], Smith et al. [51], Huang et al. [23]	URL rendering	Cross-origin page	✗	✓̸	✗	✗	N/A	≈20 ms
Naghibijouybari et al. [43]	GPU	CUDA/OpenGL	✗	✗	outside browser	✓	>1000 Hz	≈3 s
Monaco et al. [40]	Network package	Network sniffer	✗	✗	✗	✓	N/A	N/A
Panchenko et al. [46]	Network package	Network sniffer	✗	✗	outside browser	✗	N/A	≈3 s
Rendering contention channel (this work)	Page rendering	Cross-origin page	✓	✓	same-&cross-browser	✓	10–60 Hz	≈3 s

✓: The attack is feasible using the side channel, ✓̸: the attack is feasible but fixed by some browsers [3], ✗: The attack is not feasible using the side channel.
*: Existing papers do not have evaluations on cross-browser website fingerprinting and we show that the channel's cross-browser performance is reasonably low (See Table 5).
**: Although no research and experiments have been conducted using this side channel for keystroke logging, our experiment shows that it is at least feasible.

a web page can be used to sniff browser history. Such a decade old side channel, although still being there, is less severe because the loading time depends on the slowest component, which may not be cached like many Chinese websites such as sohu, QQ and 360. Stone [54] proposes that link color change between visited and unvisited URLs can be used to infer browser history, and later on Smith et al. [51] and Huang et al. [23] also improve the attack in modern browsers with defense. Google fixed this attack [3] by adding another rendering event between two visited URLs to reduce the statistical difference.

- *Website fingerprinting.* Naghibijouybari et al. [43] and Gulmezoglu et al. [22] show that an openGL or a CUDA program can infer the website based on GPU's performance counter. Jana et al. [26] track changes in the application's memory footprint and identify the website users are visiting. Kim et al. [28] show that browser activities and statuses can be inferred by monitoring storage usages. Vila et al. [57] shows that the shared event loops as a side channel can be used for identification of websites. Shusterman et al. [49] show that the cache occupancy channel contending for last-level cache can be used to fingerprint websites. Matuyunin et al. [39] and Lifshits et al. [36] show the possibility of using magnetometer and malicious batteries as side channels in fingerprinting websites. Yang et al. [62] and Spreitzer et al. [52] exploit USB power analysis and mobile data-usage statistics for website fingerprinting. Clark et al. [17] study electrical outlets as a side channel to identify webpages.
- *Keystroke logging.* Wang et al. [59] perform keystroke logging attacks via exploiting graphic libraries. Lipp et al. [37] rely on the interrupt-timing side channel to log keystrokes using sandboxed JavaScript. The aforementioned shared event loops [57] as a side channel can also be used for keystroke logging.

Other than the aforementioned attacks, side channels, especially those in WebGL and GPU, can also be used for different purposes. Lee et al. [33] study several GPU vulnerabilities, e.g., the inference of webpage via memory size. The threat model of their attacks assumes a malicious CUDA or openGL program. Booth [9] exploit resource-based side channels and show their effectiveness.

Comparison with Rendering Contention Channel We compare existing channels in Table 1:

- Adversary. A cross-origin page refers to a webpage with an origin different from the target, which is a strong model because of its easiness to launch attacks. In the past, other attack models are also adopted, such as an openGL/CUDA program, a network sniffer and a hardware adversary.
- Attack Type. The rendering contention channel supports three side-channel attacks and one covert-channel attack. Cross-browser is a strong property of this channel, which leads to two unique attacks, i.e., cross-browser website fingerprinting and cookie synchronization.
- Attack Time and Sampling Rate. The rendering contention channel has similar low sampling rate as the state of the art, i.e., the cache occupancy channel. At the same time, the attack time is shorter, because rendering mostly happens before the `onload` event but JavaScript is still running heavily after the `onload` event.

2.2 Defense against Side Channels

Browser vendors, like Firefox, Chrome and Tor Browser [6], are reducing the resolution of its timer like `performance.now` and adding jitters as a defense. Fuzzyfox [30] introduces fuzzy time to Firefox to reduce a new clock edge attack. JavaScript Zero [48] also adds noise to performance.now via a redefinition of JavaScript APIs in Chrome extension. DeterFox [12] and JSKernel [16] enforce a deterministic time upon all the events, such as frame rendering. Wu et al. [60] show that the side channel from Cao et al. [13] is caused by floating-point operations and propose to adopt integer operations and make WebGL rendering uniform. Some new browser architectures and defenses [2, 14] are proposed to isolate third-party JavaScript but cannot defend against side-channel attacks. In addition to browser-level defenses, there also exists many defenses [8, 10, 11, 15, 20, 24, 27, 32, 34, 35, 44, 47, 50, 53, 58, 61, 63, 64] in the system level against general timing attacks.

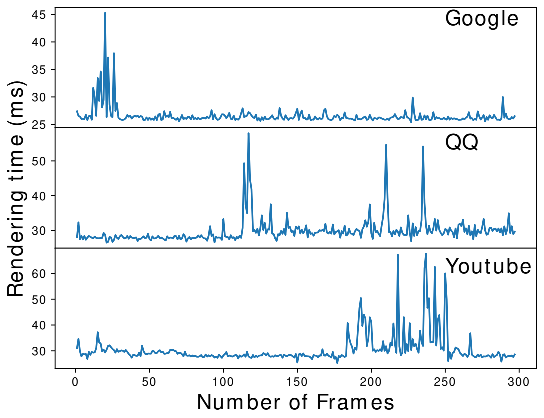


Figure 1: Rendering Patterns of QQ (www.qq.com), Google and Youtube in Tor Browser Observed from a Chrome Window through the Rendering Channel.

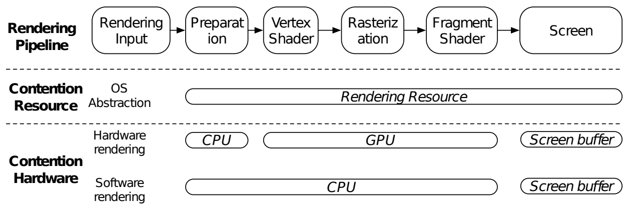


Figure 2: Rendering Pipeline and Hardware Resources.

3 Rendering Contention Channel

In this section, we answer two fundamental questions: (i) what the channel is, and (ii) what the channel's cause is (i.e., why the channel exists).

3.1 What is the rendering contention channel?

Key Take-away Answer: The rendering contention channel is that the observer, when rendering a specific workload, measures the interval between each consecutive frame and then uses the measured interval sequence as the pattern to infer another co-rendering target.

The rendering contention channel has two parties: the target and the observer. The target renders a graphics-heavy macroscale event, such as page loading; the observer measures the time to render each frame and records each frame's time as a vector to infer what the target is rendering. For example, Figure 1 shows clear, differentiable rendering patterns of three websites (QQ, Google and Youtube) visited in Tor Browser 9.0.1 but observed in Google Chrome 84.

One interesting observation here is that the channel is very noisy. There are multiple reasons. First, modern web browsers introduce a low-resolution timer and adding jitters to the timer to defend against timing channels in general. Therefore, the observed pattern fluctuates within a certain range like a background noise even if there is no target rendering events, e.g., Frame 50 and after in Figure 1 (Google) when Google finished rendering. Second, there are many events other than the target that may also be rendered at the same time to contend for the resource, e.g., the local peak at around Frame 230 in Figure 1 (Google). Third, network delays may prolong a one-frame rendering event into two or more frames, e.g., causing a half-loaded and then a fully-loaded image. The peak in Figure 1 for Youtube at around Frame 230 is such an example, which is supposed to exist in just one frame but spans over two frames in the figure.

3.2 What is the rendering contention channel's cause?

Key Take-away Answer: At the high level, the cause is a contention on the rendering resource abstracted by the operating system.
At the low level, we find three contention causes for the channel: *GPU, CPU, and screen buffer*. All three contribute to hardware rendering; only the latter two contribute to software rendering.

3.2.1 Methodology: Single Variable Testing

In this part, we describe our methodology—called Single Variable Testing—to analyze the channel's cause. The high-level idea is that we only change one single contributing factor (i.e., a single variable) of the channel but keep all others the same. Then, we observe the Signal-to-Noise Ratio (SNR) of the channel, which is defined in Equation 1.

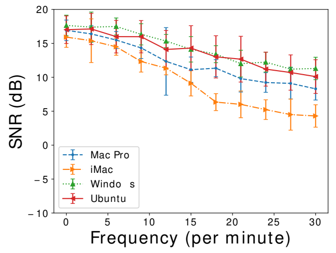
$$SNR_{dB} = 10log_{10}\frac{P_{signal}}{P_{noise}} = 10log_{10}\frac{P_{signal}}{P_{measured} - P_{signal}} \quad (1)$$

where P_{signal} is the average power of the ground truth signal and $P_{measured}$ is the average power of the measured signal from the channel. Note that if the SNR value changes with different values of the variable, the variable is considered as an influential factor—i.e., one cause—of the channel.

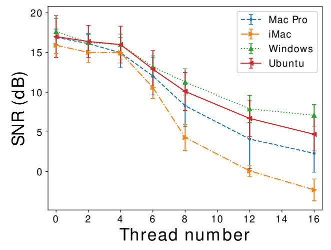
Next, we describe two things: (i) how to change each variable and (ii) what variables are considered. First, intuitively, because the rendering channel is a contention channel, we need to introduce workload for each considered variable. At the same time, we also need to change the workload constantly to introduce more noise for the channel—the more frequent the changes are, the more noise is added to the channel.

Second, we introduce different variables that are tested in the analysis. Figure 2 shows the rendering pipeline adopted by modern computers from input data to rendered images on the screen. From the high-level, rendering is abstracted by the OS as a resource; from the low-level, different elements in the pipeline are handled by different hardware resources and we describe them below.

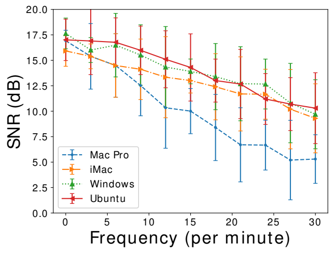
- CPU. CPU is involved in the rendering pipeline because it prepares data, e.g., matrices, for the GPU in hardware rendering or performs all the job in software rendering. We launch CPU-intensive programs and change the workload and the number of threads to test the influence of CPU on the channel.



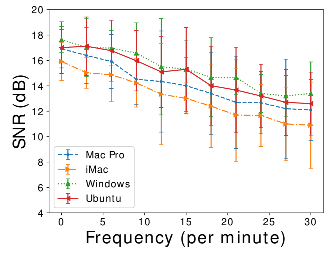
(a) CPU (varying frequencies, 8 threads, HW rendering)



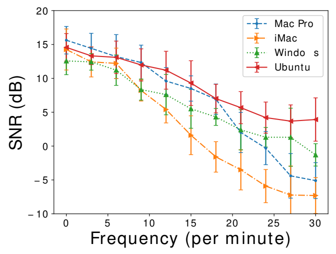
(b) CPU (varying threads, 30/min, HW rendering)



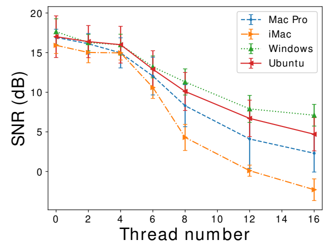
(c) GPU (varying frequencies, HW rendering)



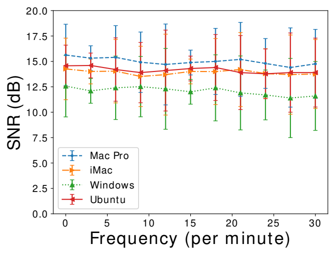
(d) Screen buffer (varying frequencies, HW rendering)



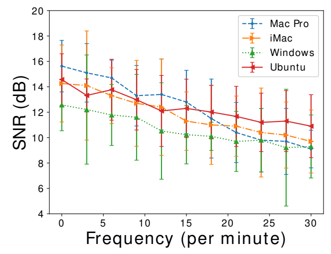
(e) CPU (varying frequencies, 8 threads, SW rendering)



(f) CPU (varying threads, 30/min, SW rendering)



(g) GPU (varying frequencies, SW rendering)



(h) Screen buffer (varying frequencies, SW rendering)

Figure 3: Signal-to-Noise Ratio (SNR) of the Rendering Contention Channel with Different Single Variables (HW: hardware and SW: software; the default number of thread is 8 and the noise frequency is 30 per minute if not otherwise indicated).

- GPU. GPU is involved in the rendering pipeline because vertex and fragment shaders are usually run in GPU to accelerate the calculation. We launch two programs: one with random matrix calculation using OpenCL in a certain frequency and the other without the calculation in the same frequency. Then, we deduct the SNR degradation caused by the latter from the former to reduce the CPU influence. Note that we choose OpenCL instead of OpenGL to remove the impacts of the screen buffer involvement.
- Screen buffer (or called Framebuffer). Screen buffer is the final stage of the rendering pipeline, which stores all the data to render in a video frame. Similar to the GPU experiment, we launch two programs: one outputting random pre-generated colors to the screen buffer in a certain frequency and the other that generates colors in the same frequency but do not draw them. Then, we deduct the SNR degradation caused by latter from the former to remove any GPU or CPU influence.

3.2.2 Experimental Setup

In this part, we describe computers and configurations used in the experiment. We have three computers: (i) iMac 4-core Intel Core i5-7600 CPU @ 3.50GHz with Radeon Pro 575 (called iMac), (ii) MacBook Pro 6-core Intel Core i7-9850H CPU @ 2.60GHz with Intel UHD Graphics 630 (called Mac Pro), (iii) Alienware Aurora R7 6-core Intel Core i7-8700k @ 3.7GHz LLC 12MB with NVIDIA GeForce GTX 1080 with Windows 10 (called Windows) and Ubuntu 20.10 (called Linux) dual Operating Systems. We use Chrome 90 for all the experiments in this section.

During the experiment, we run three programs, one as the sender, one as the observer, and the last as the noise generator. The sender runs a random workload used as the ground truth and the observer compares what been measured in the channel with the ground truth to compute the Signal-to-Noise Ratio (SNR). The noise generator changes one property, e.g., the frequency with noise on and off, and the number of threads. Each frequency or thread number is tested for 100 times with average values and standard deviations. Here are the implementation details of three types of noises.

- CPU Noise. The CPU noise is created by a WebAssembly based CPU intensive program [1] and driven by a Python code for on and off.
- GPU Noise. The GPU noise is created by two OpenCL programs (version 1.2): one that calculates random matrix multiplications and the other that does not. Each matrix in the first program ranges from 2,000×2,000 to 10,000×10,000 with random values between zero and one and the number of multiplications range from 30 to 50.
- Screen Buffer Noise. The screen buffer noise is created by two OpenGL programs (version 4.1); one that outputs random RGB colors to a 500×500 canvas and the other that does not.

3.2.3 Overall Results

Figure 3 shows the SNR of the rendering contention channel with different single variables, e.g., the number of threads and the frequency of CPU, GPU and screen buffer noises. All three factors contribute to the channel especially under hardware rendering. We now describe and analyze the detailed results.

Hardware vs. Software Rendering Figures 3a–3d show the hardware rendering results and Figures 3e–3h software rendering. GPU does not contribute to the rendering contention channel in software rendering because Figure 3g shows a flat line. Instead, the CPU's contribution in Figure 3e is very large, which can bring SNR below zero dB. As a comparison, both GPU and CPU contribute to the contention in hardware rendering. The contribution of screen buffer exists in both software and hardware rendering because both need to display contents on the screen.

Integrated vs. Dedicated GPU Figure 3c shows the SNR changes when the frequency of GPU noise increases. The channel on computers with dedicated GPU is more robust to such noises: The iMac, Windows, and Ubuntu lines (i.e., those with dedicated GPUs) are above the Mac Pro line (which only has an integrated GPU).

CPU We have two observations for CPU's contribution. First, the robustness against CPU noise depends more on the number of cores than the operating frequency. For example, both Figures 3a and 3e show that Mac Pro with more cores and lower frequency performs better than iMac. Second, processes with fewer threads have less impact on the rendering contention channel as shown in Figures 3b and 3f even when the noise frequency is 30 per minute. The reason is that some idle CPU cores are able to handle the rendering.

Windows vs. Linux The difference between the channel on Windows and Linux systems is small on hardware rendering but relatively larger on software rendering (although still being smaller those caused by different CPUs and GPUs). The reason might be that the scheduling performed by OSes on CPU is heavier than GPU.

4 SIDER: Rendering Contention Framework

In this section, we describe our general attack framework, SIDER, in reducing the noise level of the rendering contention channel. We adopt two steps, smoothing and normalization, for the denoising. The first step is to smooth the data and reduce unexpected high-value noises collected in the raw data; the second step is to normalize the raw data and mitigate diversity and noise introduced by different browsers and hardware environments.

We now present the algorithm details in Algorithm 1. The input of this denoising algorithm is the raw data collected directly from the rendering side channel and the output is the normalized sequence. The raw data is first being smoothed (Lines 6–14): Particularly, SIDER adopts a sliding window and applies smooth filter, such as an average filter, to all the data points in the window (Line 10).

Then, the smoothed data is being normalized (Lines 15–24) to standard values irrelevant to the rendering environments. The high-level idea is as follows. We find the top, say 5%, values within the smoothed data, calculate the average, and then use it as the top reference value (Line 16). Similarly, we

Algorithm 1 Denoising

```
Input: rawSeq
Output: normSeq
 1: procedure DENOISING(rawSeq)
 2:     slicedSequence ← Slice(rawSeq, startFrame, endFrame)
 3:     smoothedSeq ← Smooth(slicedSequence)
 4:     normSeq ← Normalize(smoothedSeq)
 5:     return normSeq
 6: function SMOOTH(rawSeq)
 7:     smoothedSeq ← []
 8:     frameNumber ← smoothWindow ÷ 2
 9:     repeat
10:         smoothedValue ← Filter(rawSeq, frameNumber, smoothWindow)
11:         smoothedSeq.push(smoothedValue)
12:         frameNumber++
13:     until frameNumber > (rawSeq.length - SmoothValue ÷ 2)
14:     return smoothedSeq
15: function NORMALIZE(smoothedSeq)
16:     topNormValue←smoothedSeq.top(percentage).avg()
17:     bottomNormValue ← smoothedSeq.bottom(1- percentage).avg()
18:     frameNumber ← 0, normSeq ← []
19:     repeat
20:         normValue ← (smoothedSeq[FrameNumber] -
    bottomNormalizationAverage) ÷ (topNormValue -bottomNormValue) ×
    normalizationValue
21:         normSeq.push(normValue)
22:         frameNumber++
23:     until frameNumber = smoothedSeq.length
24:     return normSeq
```

find the rest, i.e., the bottom, say 95% values, calculate the average, and then use it as the bottom reference value (Line 17). Next, the original value is normalized based on the top and bottom reference values (Lines 20–22): The bottom reference value is converted to zero and the top is *normalizationValue*, e.g., 100 (Line 20).

Implementation We implement the background stress task using a WebGL program, which renders a fixed amount of fish at random locations rotating together with a background image in a random speed. The task has many randomness, such as fish location and rotation speed, which greatly reduces caching at all levels during the rendering. Further, the task involves several rotation components, such as fish and background image, so that even if only a small amount of the rendering task is visible, the overall workload still stays stable. Note that The task has two major parts: self-adjustment and stable rendering. The former part, i.e., self-adjustment, is to change and find the number of rendered fish according to the browser. Specifically, this self-adjustment starts from a random number of fish and keeps testing the difference between the rendering interval and the target via a binary search until the frame per second (FPS) is within a target range. The latter part, i.e., stable rendering, is to render this background task constantly using the number of fish found in self-adjustment.

Result. In this part, we show SIDER's evaluation results.

- Different Background Stress Tasks. Table 2 shows the SNR values of different background stress tasks. Both the number of objects and model types have some but minimum impacts on the SNR. Random location and texture have the most impacts on SNR. The reasons are two-fold. First, random location could reduce caches from the browser,

Table 2: The Shannel's SNRs of Different Background Tasks.

WebGL project	# Objects	Model types	Location	Color	SNR (dB)
Rotating objects	20,000	7	random	texture	15.1±2.3
Rotating objects	20,000	1	random	texture	14.5±2.5
Rotating objects	10,000	7	random	texture	14.3±2.2
Rotating objects	20,000	7	fixed	texture	9.7±4.4
Rotating objects	20,000	7	random	random	5.3±3.2
Two triangles	2	2	fixed	random	1.2±4.2

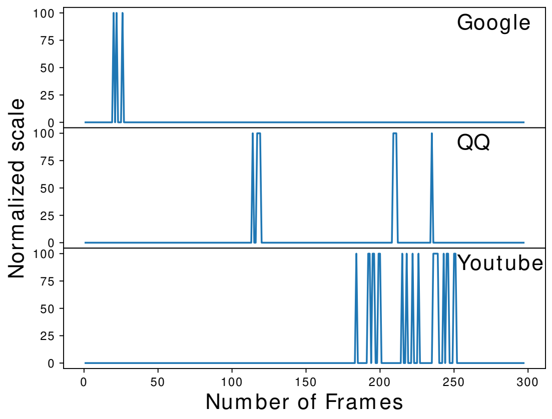


Figure 4: Denoised Rendering Patterns for Figure 1, i.e., Google, QQ, and Youtube in Tor Browser Observed from a Chrome Window through the Rendering Contention Channel. Note that we observe that each peak in the denoised curve maps to an event captured by the performance tool.

the underlying software or the hardware, which improves the task's stability. Second, texture introduces a variety of floating point operations, which could reduce the time differences from different operations [7].

- Denoising. In this part, we evaluate the denoising effectiveness of SIDER. First, we apply SIDER on Figure 1 and show the denoised rendering patterns in Figure 4. The scale of the Figure 4 is normalized to values between 0 and 100 and here we use binary values as an example. It is worth noting that we manually checked the performance tool's results and found that each peak in the figure maps to a rendering event, such as rendering of a logo or an image.

  Second, we intentionally introduce contention noises from CPU, GPU, and screen buffer and evaluate how SIDER reduce different kinds of noises. Figure 5 shows the denoising results on different machines with software and hardware rendering. The denoising results are mostly consistent across noises caused by contentions on different hardware. When the noise level increases, the denoising from SIDER also becomes more effective, i.e., the SNR difference before and after denoising increases more. In some cases, e.g., the CPU noise in Figure 5a, SIDER can double the SNR from 5 dB to 10 dB.

Table 3: A high-level summary of target events in four attacks using the rendering contention channel.

Attack	Target Event
Cross-browser cookie sync	An adversary-specified task
History sniffing	Loading of a target page by the adversary
Website fingerprinting	Loading of a target page by the user
Keystroke logging	Loading of an autocomplete textbox

5 Rendering Channel Attacks

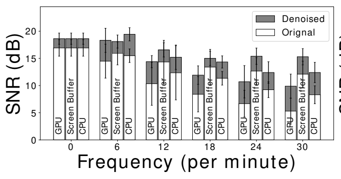
In this section, we describe how to use SIDER to launch four different attacks using the rendering contention channel. A high-level summary of different target events is shown in Table 3 and an overview of four attacks is shown in Figure 6. We now describe these attacks.

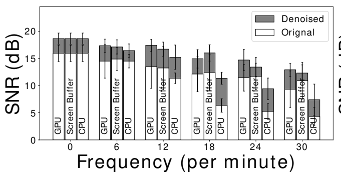
5.1 Attack One: Cross-browser cookie synchronization

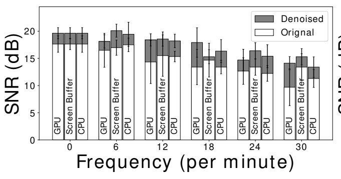
Our first attack is a covert, one-way communication channel between different browsers or modes of the same browser, e.g., normal and incognito. Such an attack can be used to synchronize tracking cookies belonging to a given domain across browser or mode. Specifically, we describe two use cases of this synchronization. First, DoubleClick, a third-party tracking website, keeps a cookie associated with user's behavior for targeted advertising on one browser. When the user opens another browser to visit webpages with DoubleClick, DoubleClick synchronizes the tracking cookie across browsers to still deliver targeted ads. Second, NYTimes uses cookies to limit the number of free articles for a user during a month. The user visits NYTimes in the incognito mode to avoid being tracked. This covert communication enables NYTimes to synchronize the cookie across modes, thus still tracking the number of free articles of the user.

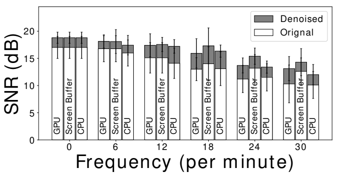
5.1.1 Attack Design

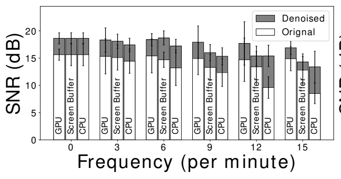
The attack design is shown in Figure 6.(a): The sender and the receiver first establish a connection based on a pre-negotiated protocol and then transmit data via the convert channel. Specifically, there are two important layers other than the raw channel and SIDER, which are (i) Connection Establishing and (ii) Encoding and Error Correction. First, SIDER establishes a connection so that both parties need to know the start time of the communication as the channel always exists. The sender renders a specific sequence of bit stream as a start and the receiver only starts to record information if the given bit stream is observed. Second, SIDER adopts error detection and correction encoding, such as Hamming code, to further reduce errors caused by noises. Specifically, the high-level idea of Hamming code is that the valid code always has a certain self-editing distance from each other and therefore some changes to a code, if being smaller than one half of the distance, can be corrected. All the communication, including the establishing pattern, are all encoded in a certain Hamming code.

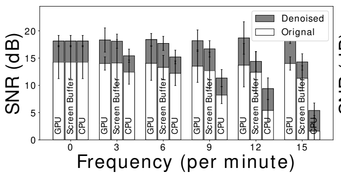

(a) Mac Pro (Hardware rendering)

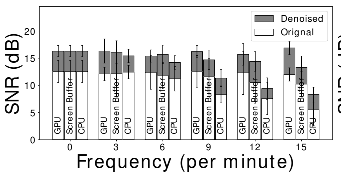

(b) iMac (Hardware rendering)


(c) Windows (Hardware rendering)


(d) Ubuntu (Hardware rendering)


(e) Mac Pro (Software rendering)


(f) iMac (Software rendering)


(g) Windows (Software rendering)

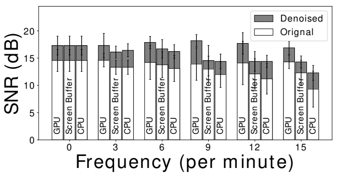

(h) Ubuntu (Software rendering)

Figure 5: Signal-to-Noise Ratio (SNR) before and after denoising on different machines.

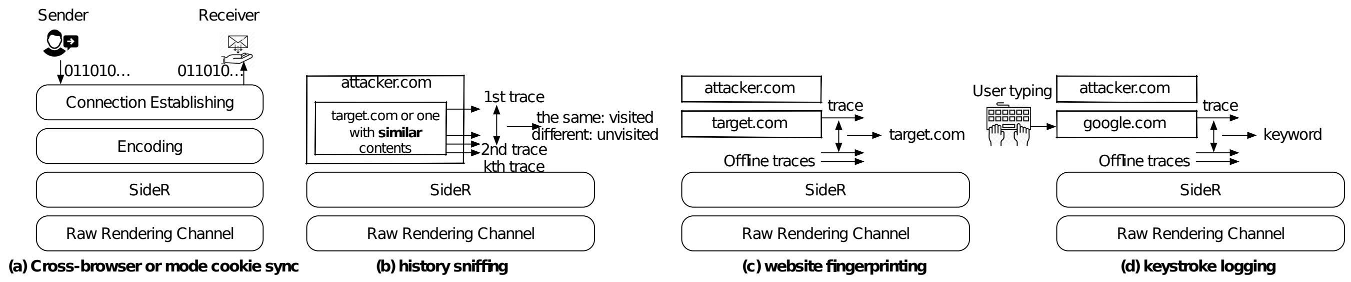


Figure 6: An illustration of four attacks using SIDER.

Theoretical Communication Bandwidth We discuss the theoretical bandwidth of this covert communication. Say we want to transmit n bits in one frame and the screen refresh rate is $Freq_{refresh}$. We show the theoretical bandwidth in Equation 2 if we assume the distribution of 0 and 1 is the same in the communication and the interval between each level (e.g., 01 and 10 in the example of two bits) is the same as the refresh interval. r is the ratio of Hamming code.

$$Bandwidth_{theory} = \frac{Freq_{refresh} \times n}{\frac{1+2+...+2^n}{2^n}} \times r = 22.9 bits/s \qquad (2)$$

where $r = 4/7$ for Hamming(7,4) code, $Freq_{refresh} = 60$ with the normal 60 Hz refresh rate, and $n = 1$ for one bit per frame.

5.1.2 Implementation and Evaluation Results

We implement a prototype of the covert communication, which pulses the rendering task on and off for a certain amount of time as the bit zero and one. We then evaluate the communication between each pair of three browsers (Google Chrome 84, Safari 13 and Firefox 79) as the sender and three plus Tor Browser 9.0.1 as the receiver. The cross-mode communication adopts one second as the interval of one pulse, the cross-browser two seconds, and any communication involving Tor Browser four seconds. Table 4 shows the experiment results of transmitting 256 random bits on an MacBook with Intel HD Graphics 515 1536 MB. All the texts are correctly transmitted without any error showing the feasibility of the attack. Note that we did not include Tor Browser as the sender due to its strong policy in deleting and isolating cookies.

Table 4: Cross-browser or cross-mode cookie synchronization of 256-byte random texts between different browser pairs. Note that (i) the diagonal line means synchronization from normal to incognito mode, and (ii) we did not include Tor Browser as a sender because it isolates all third-party cookies.

sender\ receiver	Google Chrome	Safari	Firefox	Tor Browser
Google Chrome	1.12 bits/s	0.56 bits/s	0.56 bits/s	0.28 bits/s
Safari	0.56 bits/s	1.12 bits/s	0.56 bits/s	0.28 bits/s
Firefox	0.56 bits/s	0.56 bits/s	1.12 bits/s	0.28 bits/s

We would like to point out that the actual bandwidth in practice is much smaller than the theoretical one. There are several reasons. First, it is because we cannot perfectly align the received signal with the sending signal. When the bandwidth is lower, even if the alignment has some small errors, we can still correctly infer the signal. Second, the theoretical bandwidth assumes that there exists no noise. In practice, the existence of noise will make actual bandwidth lower according to the Shannon limit.

5.2 Attack Two: History Sniffing

In this subsection, we describe our second attack, history sniffing. The key insight here is that the rendering of a visited website is different from unvisited ones. The reason is

Algorithm 2 Max-min Outlier Detection

```
Input: targetSequence, referencePool
Output: True or False
 1: function OUTLIERDETECTION(targetSequence, referencePool)
 2:     max ← maximum(referencePool.calcPairDistance(DTW-M))
 3:     newPool ← referencePool ∪ {targetSequence}
 4:     min ← min(newPool.calcPairDistance(targetSequence, DTW-M))
 5:     if max<min then
 6:         return True
 7:     else
 8:         return False
 9: procedure DTW-M(sequenceQ, sequenceC)
10:     m ← Length(sequenceQ), n ← Length(sequenceC)
11:     distanceMap ← [][]
12:     DTWMdistanceMap ← [][]
13:     for i in 0...m do
14:         for j in 0...n do
15:             distanceMap[i][j] ← i ≥ j ? |sequenceQ[i] - sequenceC[j]| : maxValue
16:             if i = 0 || j = 0 then
17:                 DTWMdistanceMap[i][j] ← distanceMap[i][j]
18:             else
19:                 DTWMdistanceMap[i][j] ← MinDistance(i, j, distanceMap, DTWMdistanceMap)
20:     return DTWMdistanceMap[m][n]
```

that modern web browsers cache contents, such as images and scripts, in memory and disk, for a visited website. Then, when the browser visits and renders the website again, these contents are immediately fetched from the cache and become available so that incremental rendering groups them together for rendering, making the rendering pattern different from an unvisited one.

5.2.1 Attack Design

Figure 6.(b) shows a high-level overview of the history sniffing attack. The attacker embeds either the target website or a website with almost exactly the same contents (i.e., the URLs of all the images, videos, and other contents are preserved) but from a different domain as an iframe. We have the second option because some websites, like Google, disallow itself to be embedded as an iframe. From a high level, the attacker loads the target repeatedly for several times (say n) and compares the first unknown loading with the rest (i.e., the loading of a cached page) using an outlier detection algorithm. If the first load is different from the rest, the attacker will consider that the target has been unvisited; otherwise, visited.

Max-min Outlier Detection Algorithm Algorithm 2 shows the algorithm. First, SIDER calculates the maximum pairwise distance (called max) among the reference group (Line 2), and then the minimum pairwise distance (called min) between the target sequence and the reference group (Line 4). If the max is less than the min (Line 5), it means that the data samples within the reference group are significantly similar to each other, but the target is an outlier (Line 6); otherwise, SIDER cannot differentiate the target from the reference group and will not consider the target as an outlier (Line 8).

SIDER adopts a modified version of Dynamic Time Warping [4], defined as DTW-M, for computing pair-wise distance between two data sequence Q and C with lengths as n and m. Specifically, SIDER first creates a matrix with dimensions of $n \times m$, in which the value of each element (i, j) is the distance between Q_i and C_j (Line 15). Then, SIDER finds a path, $W = w_1, w_2, ..., w_k$, in this matrix from $(1,1)$ to (n,m) (Lines 16–19) that satisfies the following properties:

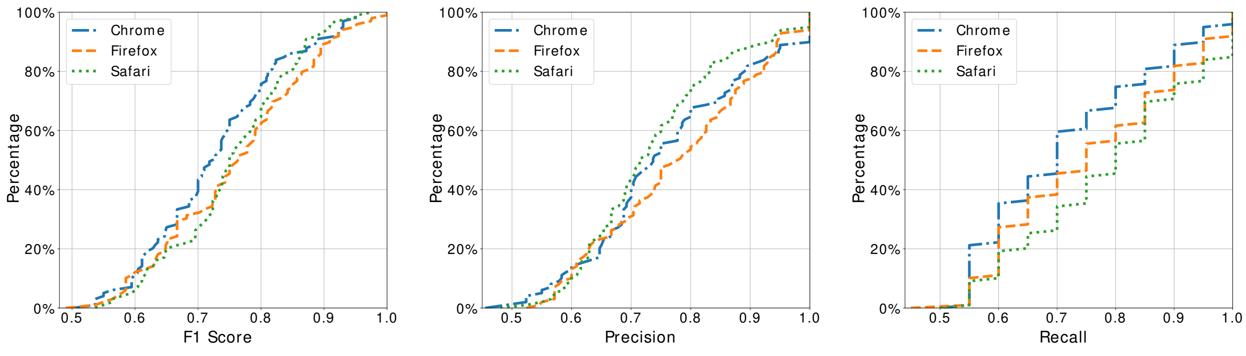
- The path starts from $(1,1)$ to (m,n).
- Continuity and monotonic. If $w_{k-1} = (a', b')$, the next step w_k can only be $(a'+1, b')$, $(a', b'+1)$ or $(a'+1, b'+1)$.
- Lagging (our modification). Each element (i, j) in the path must follow $i \geq j$.
- Minimum summation value (our modification). The summation of values of the selected path is the minimum among all possible paths.

Finally, SIDER adopts a dynamic programming algorithm to calculate the minimum distance at each step (Line 19) and selects the minimum path in the end (Line 20).

A website rerouting target contents Because some websites, such as Google and Youtube, disallow another website to embed itself as an iframe and prevent frame busting, we need to build another third-party website with a different domain name but being similar to the target. Specifically, we rely on a proxy to remove such protections, e.g., the `X-Frame-Options` header and the Content Security Policy (CSP) header, and relay all the contents without any modification to the client in another domain name. Note that such a proxied rendering has a similar effect as the original website: The index page, e.g., those in HTML, is not cached, but other contents, such as scripts and images, still are.

5.2.2 Experiment Setup

We ask real-world users from Amazon Mechanical Turk to visit our website for the history sniffing attack. The specific steps are as follows. First, we ask them to enter incognito mode for the experiments due to our IRB requirement (See Section 6). Then, we ask them to install an add-on in the incognito mode for the verification purpose. Next, we ask them to visit a selected list of websites from Alexa Top 100. Lastly, we ask them to visit our attack website for the history sniffing attack: All the data including the history sniffing result and intermediate rendering data will be then transferred back to our server for analysis via the client-side code of our attack website. It is worth noting that the add-on has two tasks. It will monitor that (i) the participant to ensure that the participant has visited the website in our instruction and also (ii) the participant is in private browsing mode so that browser histories are cleared. In practice, we do observe participants who do not install our add-on and we abandoned such data, but they all follow our instructions if the add-on is installed. We did not collect browser versions during the experiment



(a) F1-Score (b) Precision (c) Recall

Figure 7: The CDF graph of F1-Score, Precision and Recall of History Sniffing Attack against Top 100 Alexa Websites (broken down by Chrome, Firefox, and Safari).

and adopt the number (n) of loading the target webpage as four in practice.

5.2.3 Evaluation Results

In the experiment, we collected data from 60 browser instances (20 Firefox, 20 Chrome, and 20 Safari) from Amazon Mechanical Turk after filtering those who do not install our add-on.

F1-score, Precision, and Recall Figure 7 shows the Cumulative Distribution Function of F1-score, precision and recall of the history sniffing attack against Top 100 Alexa Websites on three browsers. The median F1-score is 0.723 on Chrome, 0.763 on Firefox, and 0.750 on Safari. The best performing website is Baidu, the largest search engine in China, due to its clear rendering pattern with and without cache. The worst one is the login page of TMall, because the page is too simple without much rendering to perform.

There are two things worth noting. First, the performance of websites that are directly embedded as an iframe is generally better than those that are rerouted from a third-party due to, for example, the `X-Frame-Options` protection because a third-party website indeed loses some cached contents due to a different domain name. Second, the performance of websites may differ from browser to browser. For example, the attack on Amazon, when rerouted via a third-party domain, is very high on Firefox with 0.947 F1-score, but relatively low on Chrome with 0.739 F1-score. The reason is that Firefox tends to group more contents during rendering a cached Amazon, but Chrome group less. This is supposed to be a good performance feature of Firefox, but somehow also makes it more vulnerable to the history sniffing attack.

Stealthiness Testing In this part, we evaluate the stealthiness of our history sniffing attack on local machines. Specifically, we perform three tests: (i) changing the frame size of the target website from 5%, 20%, 90% to 100% of the screen width, (ii) changing the frame size of the attack website, and (iii) partially occluding, making transparent, and introducing an overlay to the attack website. The results in this evaluation show that the F1-scores of our history sniffing attack of Top 100 Alexa websites under all stealthy settings are with 1.4% of the ones under the fully-visible iframe setting.

Over-time Attack Performance In this part, we evaluate the over-time F1-score of Top 100 Alexa websites on a given machine with five months difference. The results show that the attack F1-Score of each website may vary a little, but stay within 5% range of increase or decrease. Note that we expect that the performance of the history sniffing attack is unrelated to time, because we obtain all the traces in real-time instead of offline. The F1-score differences are mainly caused by content changes rather than any performance degradation over time.

Performance vs. the Number of Frames In this part, we evaluate the attack performance vs. the number of collected frames for three websites. The result shows a strong correlation. Particularly, we show the F1-Score of the history sniffing attack in Figure 8 as the number of frames increases of Baidu, JD and 360. The performance of Baidu is high even if there are just a small number of frames due to two explicit, early rendering events. The performance of JD and 360 is low in the beginning, but jumps at certain number of frames, because of a differentiable event in the middle of the rendering.

Attacks on Mobile Browsers In this part, we further evaluate the history sniffing attack on mobile browsers. Specifically, we choose two mobile devices: one Samsung Galaxy Note 9 with Qualcomm AArch 64 Processor rev 13 CPU, Adreno (TM) 630 GPU and the other iPhone X with Hexa-core 2.39 GHz CPU, Apple GPU. We test the history sniffing attack on `qq.com` for ten times with the default browser on both devices, i.e, Samsung Internet 15 and Safari 14. The attack succeeds on Samsung Internet (i.e., all ten inferences are correct), but fails on Safari (i.e., all ten inference results are the

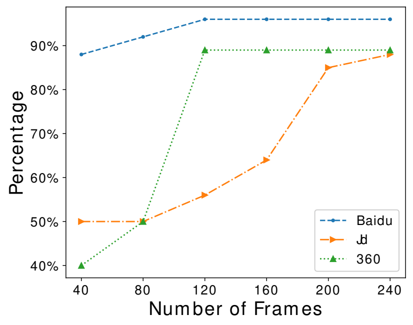


Figure 8: F1-Score vs. The number of frames for Three Websites, Baidu, JD and 360.

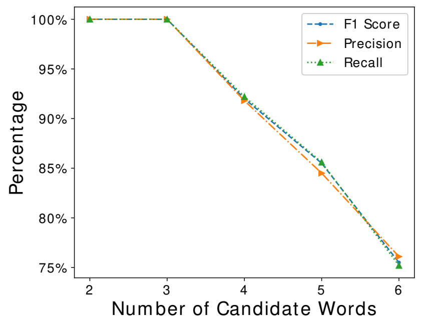


Figure 9: Precision, Recall and F1-Score of Keystroke Attacks using SIDER vs. the Number of Candidate Words

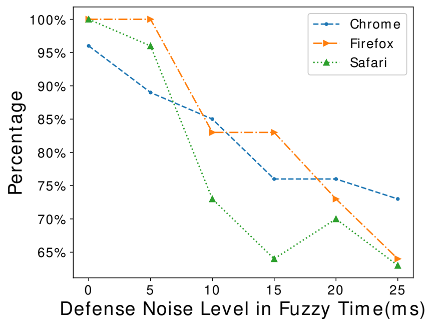


Figure 10: F1-Score of History Sniffing Attack against Baidu vs. the Defense Noise Level in Fuzzy Time.

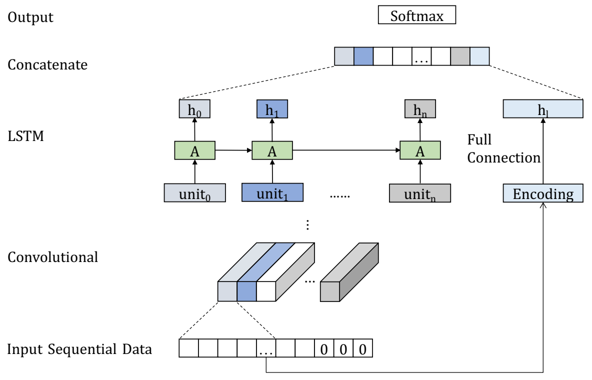


Figure 11: DNN Architecture for Website Fingerprinting.

same no matter qq.com is visited or not). The reason is that Safari separates the iframe cache from the top frame cache similar to Tor Browser. Note that the rendering contention channel still exists, but the specific attack on iPhone's Safari does not work because of the caching policy. We believe that such cache separation is a good strategy in defending against history sniffing attacks in general.

5.3 Attack Three: Website Fingerprinting

In this subsection, we describe our third attack, website fingerprinting, from design, experiment setup and evaluation.

5.3.1 Attack Design

Figure 6.(c) shows the overall attack design of our website fingerprinting attack. The attacker's website locates in a separate window from the target, which collects data using SIDER from the rendering channel and then compares the collected data with several offline traces at the server side to decide the target. We now describe our DNN-based outlier detection at the server side for fingerprinting.

DNN-based Outlier Detection We design the architecture of our DNN for two important properties: (i) support of sequential data with varied length, and (ii) support of combination of multiple side-channels. Our detailed DNN architecture is shown in Figure 11, which accepts denoised data as input and outputs a classification result of a website name. SIDER provides multiple convolutional layers with max pooling and then an LSTM layer for each side channel, and then uses one flatten and one concatenate layer to combine outputs from all the channels. Then, SIDER adopts one dropout and one dense layer after the concatenate layer to output the final, combined result. Note that the concatenate layer also supports simple channels like loading time: For example, SIDER adopts two dense layers to incorporate loading time to concatenate with other channels.

5.3.2 Experiment Setup

We now describe how we collect our offline traces and how to conduct the website fingerprinting during runtime. Note that we use Chrome 84, Firefox 79, and Safari 13 in this experiment.

Offline Traces We collect our dataset following the state-of-the-art methodology [49] with closed- and open-world settings. All the data are collected from a MacBook Pro i5-7360U LLC 4 MB with Intel Iris Plus Graphics 640.

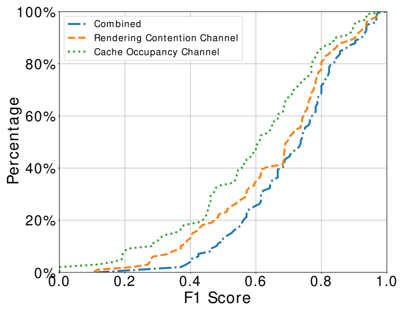
- Closed-world setting. Datasets in this setting consist of 100 traces each for 100 websites.
- Open-world setting. Datasets in this setting consist of the closed-world dataset plus 4,675 other webpages, leaving more possibilities than the closed-world setting. Note that the original code from Shusterman et al. [49] only collects 4,675 other pages rather than 5,000 as stated in the paper.

Online Attack Setting We run the website fingerprinting attack on an Alienware Aurora R7 Intel Core i7-8700k LLC 12MB with NVIDIA GeForce GTX 1080 and Windows 10. The attack website collects top 100 Alexa website data and then sends the data back to a server for the attack.

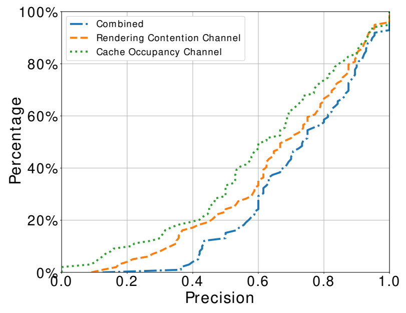
5.3.3 Evaluation Results

We evaluate same-browser, cross-browser and over-time performance of website fingerprinting in this part.

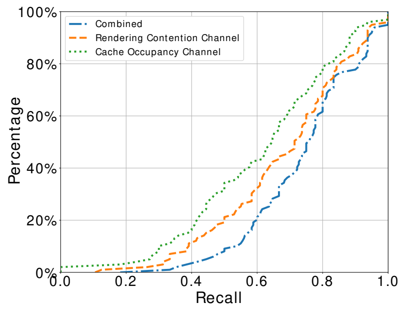
F1-Score, Precision and Recall of Same-browser Attack We evaluate the F1-Score, Precision and Recall of (i) rendering contention channel, (ii) cache occupancy [49], and (iii)



(a) F1-Score

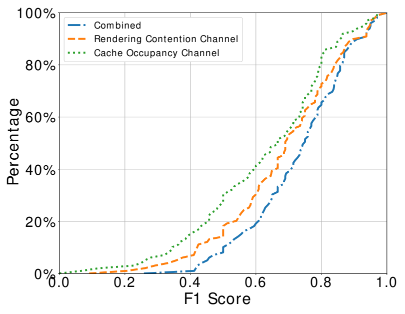


(b) Precision

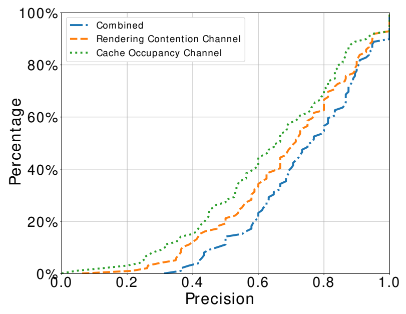


(c) Recall

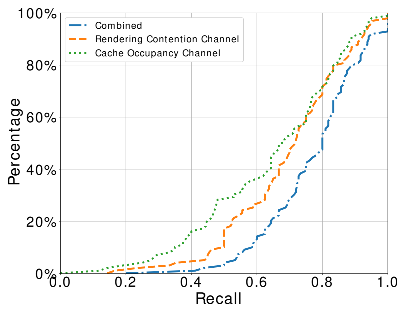
Figure 12: The CDF graph of F1-Score, Precision and Recall of Website Fingerprinting Attack against 100 Websites in a Closed-world Setting (the 100 website list and the setting configuration are from Shusterman et al. [49]).



(a) F1-Score



(b) Precision



(c) Recall

Figure 13: The CDF graph of F1-Score, Precision and Recall of Website Fingerprinting Attack against 100 Websites in an Open-world Setting (the 100 website list and the setting configuration are from Shusterman et al. [49]).

the combined with two channels running simultaneously. Figure 12 shows the closed-world result and Figure 13 the open-world. In the closed-world setting, the medium F1-Score is 0.703 for the combined channel, 0.683 for the rendering contention and 0.609 for the cache occupancy; in the open-world setting, the medium F1-Score is 0.746 for the combined, 0.690 for the rendering contention, and 0.667 for cache occupancy. The combination of two channels improves the performance of website fingerprinting.

It is worth noting that the capabilities of rendering contention and cache occupancy channels are different. The rendering contention channel is good at fingerprinting websites with high rendering load, such as video websites and those with abundant visual contents, while the cache occupancy is good at those websites with high computation tasks, e.g., JavaScript calculations. For example, rendering contention channel (R) outperforms cache occupancy (C) in `yandex.com` (R: 96.3%, C: 88.2%) and `ltn.com.tw` (R: 96.8%, C: 87.0%); by contrast, cache occupancy outperforms rendering contention in `askcom.me` (R: 82.8%, C: 88.9%) and `wittyfeed.tv` (R: 80.0%, C: 96.3%). Note that all numbers in the previous sentence are in the open-world setting.

Table 5: Performance of SIDER and cache occupancy in cross-browser website fingerprinting of 100 sites in the closed-world setting.

Cross-browser	Channel	Accuracy	F1-Score	Precision	Recall
Chrome→Firefox	Rendering contention	82.0%	66.0%	78.6%	56.9%
	Cache occupancy	52.0%	47.2%	52.0%	43.0%
Chrome→Tor Browser	Rendering contention	74.1%	57.8%	69.4%	49.5%
	Cache occupancy	42.8%	40.4%	49.5%	34.1%
Chrome→Safari	Rendering contention	80.2%	64.6%	79.6%	54.5%
	Cache occupancy	57.9%	54.8%	81.6%	41.3%

Performance of Cross-browser Attack We evaluate the performance of cross-browser website fingerprinting with two settings: (i) an adversary website located in Chrome launching the attack against visited website in Firefox, and (ii) an adversary website located in Chrome launching the attack against visited website in Tor Browser. Table 5 shows the performance of the cross-browser website fingerprinting (Chrome→Firefox, Chrome→Tor Browser, and Chrome→Safari). Note that the cross-browser attack performance against Tor Browser is on par with the same-browser attack on commercial browsers. That is, even if users adopt

Table 6: Overtime F1-score of SIDER and cache occupancy in website fingerprinting of 20 sites in the closed-world setting.

Channel	Day #1	Day #7	Day #64
Rendering contention	88.2%	82.2%	67.4%
Cache occupancy	89.0%	83.4%	60.3%

Tor Browser with a high security level, the behaviors on Tor Browser can still be inferred as long as that the user keeps another browser open in the background.

Over-time Performance We evaluate the overtime performance of both our rendering and the cache occupancy channels in terms of F1-score. That is, we collect offline traces at Day #1 and test the performance with newly crawled data at Day #X. Table 6 shows the evaluation results. The performance degradation of the rendering contention channel is similar to the cache occupancy. In the beginning at Day #7, the performance of the rendering channel degrades a little bit more than the cache occupancy. Then, at Day #64, i.e., two months later, the performance of the rendering channel is actually 7% better than the one of the cache occupancy. The reason could be that the rendering channel is more sensitive to visual content changes, but the cache occupancy is more sensitive to computational heavy task changes. In a short term, visual contents may change, but in a long term, website layouts are preserved.

5.4 Attack Four: Keystroke Logging

In this subsection, we describe the details of our key stroke logging attack.

5.4.1 Attack Design

Figure 6 illustrates the keystroke logging attack. When a user types in a search word in a search engine, such as Google, the attack will collect the runtime data and send it back to a server. Then, the server compares the data with precollected data to infer the keyword following Monaco [40].

5.4.2 Experiment Setup

We adopt a keystroke dataset collected by a research group [18] and adopted by other research papers [40]. The dataset shows over 100k users typing excerpts from the Enron email corpus and English gigaword newswire corpus. We adopt the Github repository provided by Monaco to preprocess the data, e.g., separating words, and choose popular keywords typed by different people as our dataset. We then simulate the typing with an add-on that inputs keywords following the interval specified in the dataset. SIDER is running on another window to collect performance data. All the experiments of the keystroke logging attacks are performed on a Dell machine installed with Windows 10 and Chrome 84.

5.4.3 Evaluation Results

Figure 9 shows the precision, recall and F1-Score of this keystroke logging attack when the number of candidate keywords increases. As expected, when the number of keywords is small, e.g., two and three, and those keywords different from each other, the attack's F1-Score is very high. However, when the number of keywords increases and some keywords are similar to each other, e.g., with similar length, the attack's F1-Score drops significantly to around 70%. We would like to point out that keystroke logging is the weakest attack among three because the rendering event is relatively short and the number of collected frames is relatively small.

6 Discussion

In this section, we describe several commonly-raised issues.

Ethics. We have obtained IRB approval before conducting the research. The communication between our group and IRB committee mainly focuses on two things: (i) whether our experiment will obtain private information, and (ii) whether the user is aware of our attack. First, one IRB reviewer is concerned that if a user is logged into his Facebook or Google Account, the information on his or her page may contain private information. We explained to the reviewer that our experiment is performed in private browsing mode and all cookies are cleared by default. Second, one IRB reviewer is concerned that we may conceal our data collection and therefore we explicitly show all the iframes in the attack without occlusion or transparency. In the end, we have obtained an "Exempt" decision for this project.

Limitations. We discuss several limitations of the rendering contention channel when it is used for four attacks. First, while cookie synchronization, history sniffing, and keystroke logging are unrelated to the time, the performance of website fingerprinting degrades over time because website contents may change. Our evaluation in Section 5.3 shows that the performance can at least last for a week. Second, while Tor Browser is vulnerable to cross-browser website fingerprinting and covert communication as a receiver, the performance for other attacks, e.g., history sniffing, is limited because Tor Browser deletes caches and cookies during every start and does not share them between third-party domains. The "Safer" security level further limits the attack types, because it makes WebGL click-to-play; at the same time, the aforementioned two attacks also work in the "Safer" security level. Third, our rendering contention channel requires that web browser renders contents on the screen. That is, although a user can switch to another window like incognito mode, another browser, or even another application like Word, the user cannot switch to another tab too quickly for the same-browser attack scenario. The reason is that modern browsers optimize performance and stop rendering for an inactive tab.

Other Influential Factors and Factor Breakdowns. It is worth noting that although we identify that CPU, GPU and screen buffers are contributing factors of the rendering contentions channel, it might still be some other factors that we did not test using our single variable testing. We will leave it for the future work to explore all other possible factors. Similarly, we will leave the breakdown of CPU and GPU factors, e.g., ALU, CPU cache, GPU core, and GPU cache, as our future work.

7 Possible Defenses

In this section, we discuss possible defenses against the rendering side channel and corresponding framework, SIDER. There are two traditional methods in defending against timing attacks: fuzzy and deterministic time. The former, like Tor Browser and Fuzzyfox [30], reduces timer resolution and adds jitters, while the latter, like DeterFox [12], makes the timer tick based on a deterministic event.

Fuzzy Time. We first discuss and evaluate the effectiveness of fuzzy time in defending against SIDER. Although modern browsers have already reduced their timer precision, the precision especially on commercial browsers is still relatively high, e.g., 1 ms. In this part, we mimic the behaviors of existing defenses by introducing a larger noise and reducing the resolution to the similar level. Then, we evaluate F1-Score of history sniffing attack of existing websites, e.g., Baidu, and show the results of three commercial browsers in Figure 10.

There are three things worth noting here. (i) It requires a relatively high-level defense noise, e.g., 10 ms in Figure 10, in order to influence the performance of SIDER in conducting history sniffing attack. The reason is that the useful information of this rendering side channel is the pattern across different frames instead of the performance values of each frame. (ii) The robustness of three commercial browsers are similar: Chrome needs a high noise level for defense, and Safari and Firefox are similar. (iii) Theoretically, the background rendering task can increase the workload to overcome defense noise, but in practice, SIDER cannot degrade the performance of browser rendering too much to influence user experience.

Deterministic Time. We now discuss the effectiveness of deterministic time in defending against SIDER. Since a deterministic timer normalizes the interval between consecutive frames, SIDER cannot observe any patterns to launch the attack. Therefore, deterministic timer, if implemented correctly, can defend against rendering contention channel. Specifically, we evaluated SIDER in DeterFox [12], a research prototype browser modified from Firefox, with deterministic time. The workload adjustment of our background rendering task fails to find an appropriate FPS because the FPS is always a determined value in DeterFox.

However, deterministic time brings compatibility and functionality issues. Say a WebGL program wants to adjust its workload dynamically based on the FPS. The FPS measured using a deterministic timer is always a constant value. That is, deterministic time sacrifices important WebGL functionalities for FPS measurement in defending against rendering contention channel.

8 Conclusion

In this paper, we propose a rendering contention side channel that stresses the rendering resource abstracted by operating systems, measures the time taken to render a sequence of frames, and then infers any co-rendering event of the browser. We then perform single variable testing and deduce that the rendering contention channel is caused by a combination of CPU, GPU, and screen buffer although the detailed breakdown depends on different configurations, e.g., software vs. hardware rendering. We further designed and implemented an attack framework, called SIDER, and launched four types of attacks: cross-browser/mode cookie synchronization, history sniffing, website fingerprinting and keystroke logging. Our evaluation shows that all four attacks are feasible in practical settings.

Acknowledgment

We would like to thank our shepherd, Dr. Sangho Lee, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] Cpu stress test online. https://cpux.net/cpu-stress-test-online.

[2] Google code home page of configurable origin policy. http://code.google.com/p/configurableoriginpolicy/.

[3] Issue 835589: Security: Css paint api leaks visited status of links (up to 3k/sec). https://bugs.chromium.org/p/chromium/issues/detail?id=835589.

[4] [wikipedia] dynamic time warping. https://en.wikipedia.org/wiki/Dynamic_time_warping.

[5] [wikipedia] incremental rendering. https://en.wikipedia.org/wiki/Incremental_rendering.

[6] Tor browser, 2017. https://www.torproject.org/projects/torbrowser.html.en.

[7] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy* (2015), pp. 623–639.

[8] Aviram, A., Hu, S., Ford, B., and Gummadi, R. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2010), CCSW '10, ACM, pp. 103–108.

[9] Booth, J. *Not so incognito: Exploiting resource-based side channels in JavaScript engines*. PhD thesis, 2015.

[10] Buiras, P., Levy, A., Stefan, D., Russo, A., and Mazieres, D. A library for removing cache-based attacks in concurrent information flow systems. In *International Symposium on Trustworthy Global Computing* (2013), Springer, pp. 199–216.

[11] Cabuk, S., Brodley, C. E., and Shields, C. Ip covert timing channels: Design and detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 178–187.

[12] Cao, Y., Chen, Z., Li, S., and Wu, S. Deterministic browser. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 163–178.

[13] Cao, Y., Li, S., Wijmans, E., et al. (cross-)browser fingerprinting via os and hardware level features. In *NDSS* (2017).

[14] Cao, Y., Li, Z., Rastogi, V., Chen, Y., and Wen, X. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2012), ASIACCS, ACM, pp. 8–9.

[15] Chen, A., Moore, W. B., Xiao, H., Haeberlen, A., Phan, L. T. X., Sherr, M., and Zhou, W. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 541–554.

[16] Chen, Z., and Cao, Y. Jskernel: Fortifying javascript against web concurrency attacks via a kernel-like structure. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2020), pp. 64–75.

[17] Clark, S. S., Mustafa, H., Ransford, B., Sorber, J., Fu, K., and Xu, W. Current events: Identifying webpages by tapping the electrical outlet. In *European Symposium on Research in Computer Security* (2013), Springer, pp. 700–717.

[18] Dhakal, V., Feit, A. M., Kristensson, P. O., and Oulasvirta, A. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2018), CHI '18, Association for Computing Machinery.

[19] Felten, E. W., and Schneider, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2000), CCS '00, ACM, pp. 25–32.

[20] Gianvecchio, S., and Wang, H. Detecting covert timing channels: an entropy-based approach. In *ACM Conference on Computer and Communications Security* (2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, pp. 307–316.

[21] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. Aslr on the line: Practical cache attacks on the mmu. In *Annual Network and Distributed System Security Symposium* (2017), NDSS.

[22] Gulmezoglu, B., Zankl, A., Eisenbarth, T., and Sunar, B. Perfweb: How to violate web privacy with hardware performance events. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 80–97.

[23] Huang, A., Zhu, C., Wu, D., Xie, Y., and Luo, X. Cross-platform improvement: an adaptive method of browser history sniffing. In *Measurements, Attacks, and Defenses for the Web (MADWeb) Workshop* (2020).

[24] Huisman, M., Worah, P., and Sunesen, K. A temporal logic characterisation of observational determinism. In *CSFW* (2006), IEEE Computer Society, p. 3.

[25] Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.

[26] Jana, S., and Shmatikov, V. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 143–157.

[27] Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.

[28] Kim, H., Lee, S., and Kim, J. Inferring browser activity and status through remote monitoring of storage usage. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), pp. 410–421.

[29] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In

Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 104–113.

[30] KOHLBRENNER, D., AND SHACHAM, H. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 463–480.

[31] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: Timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 1055–1062.

[32] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[33] LEE, S., KIM, Y., KIM, J., AND KIM, J. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 19–33.

[34] LI, P., GAO, D., AND REITER, M. K. Mitigating access-driven timing channels in clouds using stopwatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013* (2013), pp. 1–12.

[35] LI, P., GAO, D., AND REITER, M. K. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur. 17*, 2 (Nov. 2014), 8:1–8:28.

[36] LIFSHITS, P., FORTE, R., HOSHEN, Y., HALPERN, M., PHILIPOSE, M., TIWARI, M., AND SILBERSTEIN, M. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies 2018*, 4 (2018), 141–158.

[37] LIPP, M., GRUSS, D., SCHWARZ, M., BIDNER, D., MAURICE, C., AND MANGARD, S. Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 191–209.

[38] LIU, Y., GHOSAL, D., ARMKNECHT, F., SADEGHI, A.-R., SCHULZ, S., AND KATZENBEISSER, S. Hide and seek in time - robust covert timing channels. In *ESORICS* (2009), M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 120–135.

[39] MATYUNIN, N., WANG, Y., ARUL, T., KULLMANN, K., SZEFER, J., AND KATZENBEISSER, S. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society* (2019), pp. 135–149.

[40] MONACO, J. V. What are you searching for? a remote keylogging attack on search engine autocomplete. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 959–976.

[41] MOWERY, K., BOGENREIF, D., YILEK, S., AND SHACHAM, H. Fingerprinting information in javascript implementations. In *WEB 2.0 SECURITY & PRIVACY (W2SP)* (2011).

[42] MULAZZANI, M., RESCHL, P., HUBER, M., LEITHNER, M., SCHRITTWIESER, S., WEIPPL, E., AND WIEN, F. Fast and reliable browser identification with javascript engine fingerprinting. In *WEB 2.0 SECURITY & PRIVACY (W2SP)* (2013).

[43] NAGHIBIJOUYBARI, H., NEUPANE, A., QIAN, Z., AND ABU-GHAZALEH, N. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 2139–2153.

[44] NING, P., REEVES, D. S., AND PENG, P. On the secrecy of timing-based active watermarking trace-back techniques. *IEEE Symposium on Security and Privacy* (2006).

[45] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1406–1418.

[46] PANCHENKO, A., LANZE, F., PENNEKAMP, J., ENGEL, T., ZINNEN, A., HENZE, M., AND WEHRLE, K. Website fingerprinting at internet scale. In *NDSS* (2016).

[47] SABELFELD, A., AND SANDS, D. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE* (2000), IEEE, pp. 200–214.

[48] SCHWARZ, M., LIPP, M., AND GRUSS, D. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS* (2018).

[49] SHUSTERMAN, A., KANG, L., HASKAL, Y., MELTSER, Y., MITTAL, P., OREN, Y., AND YAROM, Y. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 639–656.

[50] SMITH, G., AND VOLPANO, D. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 355–364.

[51] SMITH, M., DISSELKOEN, C., NARAYAN, S., BROWN, F., AND STEFAN, D. Browser history re: visited. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (2018).

[52] SPREITZER, R., GRIESMAYR, S., KORAK, T., AND MANGARD, S. Exploiting data-usage statistics for website fingerprinting attacks on android. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2016), pp. 49–60.

[53] STEFAN, D., BUIRAS, P., YANG, E. Z., LEVY, A., TEREI, D., RUSSO, A., AND MAZIÈRES, D. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security* (2013), Springer, pp. 718–735.

[54] STONE, P. Pixel perfect timing attacks with html5 (white paper).

[55] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1382–1393.

[56] VAN GOETHEM, T., VANHOEF, M., PIESSENS, F., AND JOOSEN, W. Request and conquer: Exposing cross-origin resource size. In *Proceedings of the 21st USENIX Conference on Security Symposium* (2016), Security.

[57] VILA, P., AND KÖPF, B. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 849–864.

[58] VOLPANO, D., AND SMITH, G. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th* (1997), IEEE, pp. 156–168.

[59] WANG, D., NEUPANE, A., QIAN, Z., ABU-GHAZALEH, N. B., KRISHNAMURTHY, S. V., COLBERT, E. J., AND YU, P. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS* (2019).

[60] WU, S., LI, S., CAO, Y., AND WANG, N. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1645–1660.

[61] WU, W., AND FORD, B. Deterministically deterring timing attacks in deterland. In *Conference on Timely Results in Operating Systems (TRIOS)* (2015).

[62] YANG, Q., GASTI, P., ZHOU, G., FARAJIDAVAR, A., AND BALAGANI, K. S. On inferring browsing activity on smartphones via usb power analysis side-channel. *IEEE Transactions on Information Forensics and Security 12*, 5 (2016), 1056–1066.

[63] ZDANCEWIC, S., AND MYERS, A. C. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA* (2003), p. 29.

[64] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* (2012), pp. 99–110.

[65] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 313–328.

[66] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.