FROST: Fingerprinting Remotely using OPFS-based SSD Timing

Hannes Weissteiner[1]✉, Tobias Weiser[1]*, Roland Czerny[1], Sudheendra Raghav Neela[1], Fabian Rauscher[1], Jonas Juffinger[2], and Daniel Gruss[1]

[1] Graz University of Technology, Graz, Austria
{first.last}@tugraz.at, *tobias.weiser@student.tugraz.at
[2] Liebherr-Transportation Systems GmbH, Korneuburg, Austria
mail@jonasjuffinger.com

Abstract. Prior work showed that variations in SSD access time can be used to leak information about user activity, e.g., the websites a user accesses, and for covert data transmission. To achieve this, SSD contention side channels require accurate high-resolution timing measurements of I/O operations, e.g., through the `io_uring` kernel API. However, the impact of these attacks is limited in their requirement for native code execution on the victim's system.

In this paper, we show that SSD contention side channels can be mounted by a remote attacker from *within the browser*, without native code execution. Our attack FROST targets the Origin Private File System (OPFS) API in JavaScript, allowing us to create and access files on the disk, within the browser's sandboxed environment. While a challenge in prior work was to evict the OS page cache, we devise an approach that instead bypasses the page cache, enabling fast SSD contention measurements from JavaScript *without any user interaction*. To evaluate the effectiveness of FROST on macOS and Linux, we build a covert channel that exfiltrates data from a native application to the malicious website with a true channel capacity of 661.63 bit/s on a Linux machine, and 891.77 bit/s on a macOS machine. To evaluate FROST in a side-channel scenario, we mount a website- and an application-fingerprinting attack on users of macOS systems. We can predict accessed websites with an F_1 score of 88.95 %, and accessed application with an F_1 score of 95.83 %, demonstrating the privacy implications our attack has on regular users.

1 Introduction

Web browsers have evolved from simple document viewers into complex platforms capable of running sophisticated applications. Companies like Google, Microsoft, and Adobe have developed full-fledged office suites, photo- and video editors, or even integrated development environments (IDEs) that run entirely within the browser. Moving applications to the web allows them to be platform-independent, accessible and always up-to-date. However, this shift also introduces new security and privacy challenges. New APIs and features are continuously added to browsers to support these complex applications, e.g., to support

device access via WebUSB [47], expose graphics capabilities via WebGPU [46], or provide persistent storage via the File System Access API [37]. While these features enhance the capabilities of web applications and allow completely novel use cases, they also increase the browser's attack surface, and some have already been shown to introduce new vulnerabilities [14, 20, 68].

The File System Access API allows web applications to request access to files and directories on the user's file system, enabling complex applications, such as IDE's, photo- or video editors, to run entirely within the browser, with a native-like access to the shared file system. Whenever a web application requests access to a file or directory, the user must explicitly grant access permissions in a browser dialog. To enforce this selective access, the browser implements various security checks to ensure that the web applications cannot access files outside of the granted scope [44]. Meanwhile, the Origin Private File System (OPFS) [44] provides a sandboxed file system for each web origin, allowing web applications to store and access files on the user's system *without requiring explicit permissions*. The OPFS is isolated from other origins and the rest of the system, making sure that web applications cannot access files outside of their own OPFS storage. Because of this isolation, accesses to OPFS files do not require as many security checks as regular file accesses, resulting in increased performance.

While SSDs are widely used in both servers and consumer systems, side-channel research targeting them is limited [19,31,41,70]. Recently, Juffinger et al. [30] discovered that it is possible to exploit contention on NVMe SSDs to build a covert channel and perform fingerprinting attacks. While their attacks are practical on a wide range of SSDs and are able to achieve high F_1 scores, they do require access to native low-overhead interfaces, e.g., `io_uring`, and suggest that restricting access to these interfaces would hinder their attacks.

In this paper, we introduce FROST, a side-channel attack from JavaScript that exploits OPFS to leak sensitive information from the browser without requiring any user interaction on both Linux and macOS. OPFS provides a direct way to perform operations on the system's disk from JavaScript in a sandboxed environment. While OPFS restricts websites to only access their own storage and not other system files, we discover that the overhead is low enough to enable SSD-contention-based side-channel attacks from within the browser. Using this low-overhead interface, we achieve a covert-channel throughput of 661.63 bit/s on Linux, comparable to the native performance reported in prior work [30]. Based on our findings, we introduce Fingerprinting Remotely using OPFS-based SSD Timing (FROST), a novel remote side-channel attack that uses SSD contention measurements from within the browser to fingerprint user activity on a system. After tricking the victim into clicking a malicious link, an attacker can monitor the victim's activity on the host system, such as website visits and application usage, without further user interaction. We demonstrate FROST on a macOS system, achieving an F_1 score of 88.95 % for top-50 closed-world website fingerprinting, and 95.83 % for application fingerprinting in a closed-world setting. In an open-world top-50 website-fingerprinting attack, we achieve a macro-averaged F_1 score of 86.95 %, demonstrating the practical impact of our attack. Finally,

we discuss the impact of modern browser features on side-channel attacks, limitations of our work, and possible defenses.

In summary, our **contributions** are as follows:

- We are the first to demonstrate that the Origin Private File System (OPFS) can be exploited remotely from JavaScript in the browser to leak sensitive information from a victim's system without any user interaction.
- We exploit the OPFS to generate and measure SSD contention on the victim's system, building a cross-application covert channel with a capacity of 661.63 bit/s on Linux, almost matching native performance from prior work [30], and 891.77 bit/s on macOS.
- We present FROST, a remote SSD contention-based side-channel attack that can be mounted from within the browser, exploiting the OPFS to monitor user activity on the victim's system without any user interaction.
- We demonstrate FROST's effectiveness by mounting website- and application fingerprinting attacks, achieving an F_1 score of 88.95 % and 95.83 %, respectively, in a closed-world setting on a macOS system.

In Section 2, we provide background on side-channel attacks, SSDs, fingerprinting attacks and browser features. In Section 3, we describe our threat model. In Section 4, we show how we can measure SSD contention from JavaScript without any user interaction. In Section 5, we build a covert channel using our SSD contention measurements, evaluating its capacity with both OPFS and the regular File System Access API. In Section 6, we present FROST, and mount website- and application-fingerprinting attacks. In Section 7, we discuss the impact of increasingly complex browser features on side-channel attacks, mitigations, and related work. We conclude in Section 8.

Responsible Disclosure. We responsibly disclosed our findings to Google (Chromium security), Mozilla (Firefox security), and Apple (Safari security). The Chromium team stated that they do not consider fingerprinting attacks to be security vulnerabilities. Apple considers the attack currently out of scope, but may implement a mitigation in the future. Mozilla acknowledged our findings but did not implement any mitigations yet.

2 Background

In this section, we provide background on side-channel attacks, solid-state drives, website fingerprinting and storage access on browsers.

2.1 Side-Channel Attacks

Side-channel attacks exploit measurable effects, e.g., electromagnetic radiation [27], power consumption [54], execution time [35], acoustic signals [5], or network metadata [7] to infer sensitive information indirectly. Modern CPUs contain a significant amount of shared hardware between cores, making them highly

susceptible to a wide range of side-channel attacks. One way to obtain metadata on CPUs to perform a side-channel attack is through contention. Competitively shared hardware, such as a shared data cache [24, 52, 55], execution ports [3,59,60], the system bus [56], the scheduler queue [16,17], and the micro-op cache [57], can be used to determine a victim's activity by measuring the throughput or availability of that hardware in an attacker application. Tan et al. [67] attacked contention on a PCIe bus that is connected to a switch to leak behavior of other PCIe devices. Jiang et al. [29] exploited a timing side channel in `fsync` operations. Liu et al. [41] exploited a cache timing side channel in the now-discontinued Intel Optane persistent memory. Recently, Juffinger et al. [30] explored contention-based side channels on NVMe SSDs, achieving a website fingerprinting attack with an F_1 score of 78 % to 96 % and also a cache side channel based on SSD's HMB cache [32].

Covert Channels. A covert channel allows two cooperating parties to transfer information covertly through methods not intended for information transfer, effectively hiding it from an unsuspecting observer. While covert channels have their own use case, *i.e.*, transferring information, they are also useful for determining the maximum leakage rate of a side-channel attack. Both the sender, equivalent to the victim during an attack, and the receiver, equivalent to an attacker who records the leaked information, cooperate, simulating a best-case scenario for an attack and therefore providing an upper bound for the leakage rate. Covert channels have been studied on various hardware and software components, e.g., CPU caches [23, 61], transient execution elements [36, 40, 62], network effects [6, 58], page cache [22, 49] or hard drive contention [38]. Trochatos et al. [70] show a thermal-based covert channel on smart SSDs. Giechaskiel et al. [19] exploit SSD contention for co-location detection of AWS instances.

2.2 Solid-State Drives

Solid-state drives store data in NAND flash chips and deliver significantly higher I/O performance than mechanical hard drives (HDDs), offering much lower latencies than HDDs. Flash memory generally operates at a page granularity. However, erasing resets entire blocks (32-128 pages) and is much slower. Each cell endures a limited number of program-erase cycles, so controllers use wear-leveling and batch erases to spread wear [8]. Because wear-leveling and other optimizations relocate data, SSDs expose a logical address space to the OS that does not match the physical layout. These optimizations also scatter data, and thus the SSD maintains a flash translation layer (FTL) that maps OS logical pages to physical locations. On SSDs that use a Host Memory Buffer (HMB) [50], this translation process is also exploitable for side-channel attacks [32].

2.3 Website Fingerprinting

In a website fingerprinting attack, an attacker's goal is to determine the website visited by a user through metadata, e.g., by employing a side channel. Extensively researched on networks, an attacker-in-the-middle observes network packet

flow and utilizes traffic analysis to determine the website visited by a user [28,53], *i.e.*, *fingerprint* the website, even over the Tor network [10, 74]. Similar styles of attacks via network traffic analysis have been used to fingerprint the type of initiated network connection [11], applications used on an end device [2, 12, 78], operating system and CPU model [1,69], or browser used to visit a webpage [33].

In recent years, side channels on computers have been shown to fingerprint website visits [34, 42, 76], applications [9, 42], and physical peripherals [65] on a user's system. Prior work mounted website-fingerprinting attacks using side channels, e.g., on the CPU cache locally [77] and remotely [51], GPU caches [13], interrupts [56, 79], android usage statistics [66], and performance counters [26]. Jiang et al. [29] showed that `fsync`-contention leakage can distinguish certain websites, while Gu et al. [25] mount a full website-fingerprinting attack using filesystem-level syncfs operations, achieving an F_1 score of 93.25 % in an open-world setting on the top 100 websites. By observing leakage with SSD caches, Juffinger et al. [30] mounted a website fingerprinting attack on the top-100 websites with different SSDs accuracy ranging between 78 % to 96 %.

2.4 Browser Storage Access

As web applications become more complex, they increasingly rely on client-side storage for data persistence. Even basic websites use cookies to locally store session information for the server. Other applications use more advanced storage mechanisms, such as LocalStorage [48] or IndexedDB [43], to store larger amounts of persistent data on the client side. However, the trend of moving full applications, such as IDEs, photo- or video editors into the web, resulted in the need for browser applications to access files on the user's system directly. The File System Access API provides functionality to request access to files and directories on the user's file system [37]. More recently, the Origin Private File System (OPFS) allows web applications to create and manage files in an origin-scoped file system on the client [44] without requiring explicit user permission.

3 Threat Model

Our threat model assumes that the attacker tricks the victim into visiting a malicious website, which hosts the attack code. We assume default configurations for the operating system and browser. The attack runs in the browser sandbox, without any special privileges or permissions, aiming to leak information about the victim's behavior on the system. Thus, the victim leaves the attacker's website open while performing other activities on the system, which is in line with browser-based side channels [15,18,33,39,51,73,77]. We show an overview of the attack model in Figure 1. The victim is not required to use the same browser, as the attack works on the storage level, across the whole system. However, the activities that the victim uses must result in storage accesses to the same disk as the file that is used for contention measurement. This is the case for many consumer devices, such as laptops, as they typically only have one internal SSD.

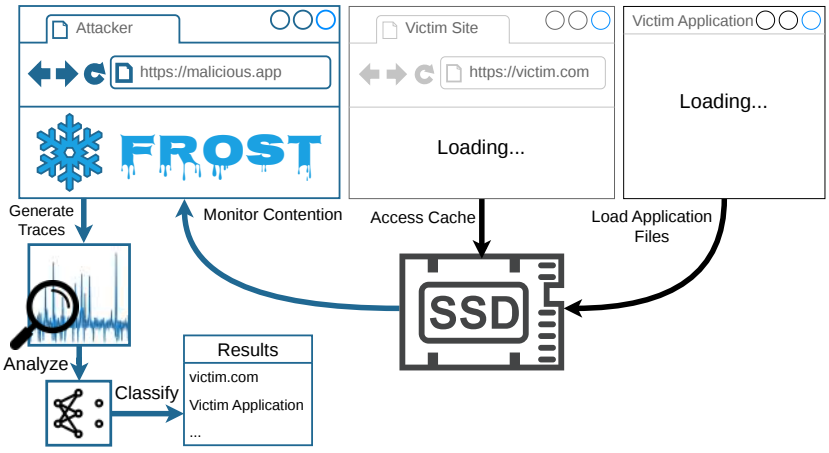


Fig. 1. High-level overview of the FROST attack. The attacker measures SSD contention caused by the victim's activity by performing random reads on a large OPFS file. The resulting timing traces are analyzed to fingerprint the activity.

4 Measuring SSD Contention from JavaScript

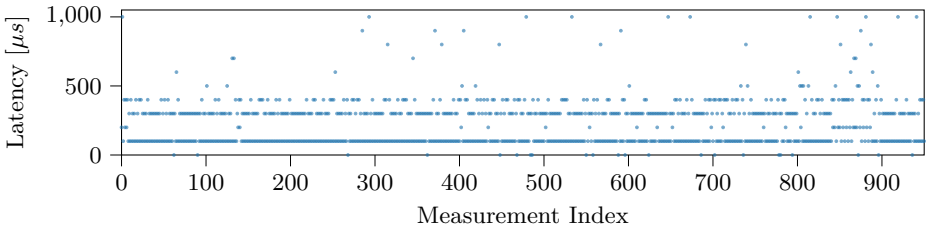
To measure SSD contention, we first require a way to force the browser to perform storage accesses. Prior work has used IndexedDB to perform storage accesses from JavaScript [71], building a HDD contention-based covert channel able to transmit 32 bits in just over 1 min. In contrast, the File System Access API [37] allows direct access to files on the user's file system. The Origin Private File System (OPFS) provides a more permissive, but isolated, file system.

Evading the Page Cache. Even fast SSDs are still magnitudes slower than the computer's main memory (DRAM). To close this latency gap, operating systems cache recently accessed disk data in memory. In Linux, this cache is called the page cache; in macOS, it is called the Unified Buffer Cache (UBC). We use the term page cache for both operating systems. Typically, operating systems use all unused memory for the page cache, *i.e.*, memory not used by running programs. Due to this page cache, only the first access to a file actually reads data from the underlying disk. On subsequent accesses, the file is cached in memory and its contents served from there, until it is evicted from the page cache again. While a page of a file is cached, reading does not leak any timing information about underlying disk. Hence, prior work used page cache eviction [22] to force the OS to read from the disk again. Since we do not target specific files but the SSD itself, we do not need to rely on page cache eviction. Instead, to bypass the page cache for long-running measurements, we create a file larger than system memory. This forces the caches to evict older data whenever we access new parts of the file, and, due to the large file size, the page cache can never fully cache the file, thus forcing the OS to read from the disk on virtually every access.

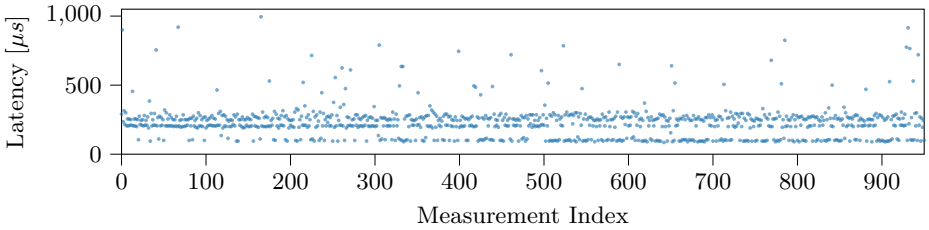
Avoiding User Interaction. The File System Access API generally requires the user to explicitly share files or directories with the web application, requiring manual confirmation, which may raise suspicion. Additionally, while users may be tricked into allowing access to a file (e.g., if the website poses as a file sharing platform), the selected file may not be large enough to achieve reliable eviction from the OS page cache, preventing the attacker from measuring SSD access times reliably. The newer Origin Private File System (OPFS) provides an isolated sandboxed file system for each origin without requiring any user interaction [44]. Files stored in the OPFS persist on the user's file system, allowing us to perform storage accesses without raising suspicion. OPFS is supported in all major browsers, including Chrome, Firefox, and Safari, making our attack widely applicable. Therefore, we can create a large file in the OPFS without any special permissions, and thus do not rely on any user interaction. On Chrome and Safari, OPFS supports very large files, up to 60 % of disk space, which is more than sufficient to avoid the page cache on most typical systems, as even a small disk size of 64 GB would allow us to create a 38.4 GB OPFS file. Thus, we can fill up the entire system memory on almost all systems, except specialized systems with very little storage and a large amount of system memory. On Firefox, the OPFS supports up to 10 % of total disk space or 10 GB (whichever is less) [45] per origin (*i.e.*, website). However, an attack using multiple independent origins can bypass this limit, as each origin has its own OPFS storage, only slightly complicating the attack. Additionally, websites can ask for permission to use persistent storage, also bypassing the OPFS storage limit.

Measuring Access Times. After obtaining a primitive that allows us to perform SSD accesses, either via OPFS or via user interaction, we can measure file access times to detect SSD contention. We read from random offsets in the large file, to avoid prefetching effects and caching. We continuously measure the disk access latency and send resulting data to the attacker's server, storing it in a trace file. The measurement process does not require significant processing power. In all of our measurements, we experience large spikes in access times at constant intervals, likely caused by scheduling interference. To prevent these spikes from interacting adversely with our further analysis, we filter the spikes by replacing every measurement greater than 1 ms with the average of the 100 surrounding values. On average, this process removes 31.88 of samples from each trace, or 0.1 % of data points. We find that this post-processing step significantly improves the reliability of further data analysis.

Increasing Measurement Resolution. To avoid timing side channels between unrelated origins, browsers limit the timer resolution based on the cross-origin policies of the website [75]. We show the effect of these low timer resolutions in Figure 2. While this reduction of accuracy may protect against attacks where websites are attacked directly by loading them inside iframes, our attack is not affected by any cross-origin restrictions, as we do not perform web requests to other origins. Thus, we can enable cross-origin isolation by configuring strict



a. SSD access latency without cross-origin isolation. The reduced timer resolution causes low measurement accuracy, with most of the measurements reading 100 µs, 200 µs, or 300 µs. Some measurements are even rounded down to 0 µs.



b. SSD access latency with cross-origin isolation. Enabling more strict cross-origin policies allows websites to access high-resolution timers, resulting in more fine-grained latency measurements.

Fig. 2. SSD latency measurements with and without cross-origin isolation. Browsers limit the timer resolutions based on the cross-origin policies of the origin. By enabling strict cross-origin policies, we gain access to high-resolution timers.

options for the Cross Origin Opener Policy (COOP) and Cross Origin Embedder Policy (COEP) headers to gain access to high-resolution timers, improving the accuracy of our measurements.

Experimental Setup. For our Linux measurements, we use a desktop PC with an AMD Ryzen 7 5800X3D CPU, 32 GB of DDR4 RAM and a 256 GB SanDisk SD8SN8U SSD running NixOS 26.05 with kernel version 6.18.5. On this machine, we perform latency measurements using Google Chrome 143.0.7499.192.

For our macOS measurements, we use a Mac mini with an Apple M2 CPU, 8 GB of RAM, and a 256 GB SSD, running macOS Sonoma 14.2.1, and running the attack in Google Chrome (version 144.0.7559.133). For long-running measurements, we assume a default configuration, but disable features interrupting the measurements, *i.e.*, automatic sleep modes, automatic logout, and suspend.

On both machines, we constantly measure the access time to a large file stored on the SSD. We perform random reads of 4 kB, spanning the entire file to avoid prefetching effects, and measure their execution time using the `performance.now()` API. We configure the COOP/COEP headers to enable high-resolution timers. During the File System Access API measurements, we

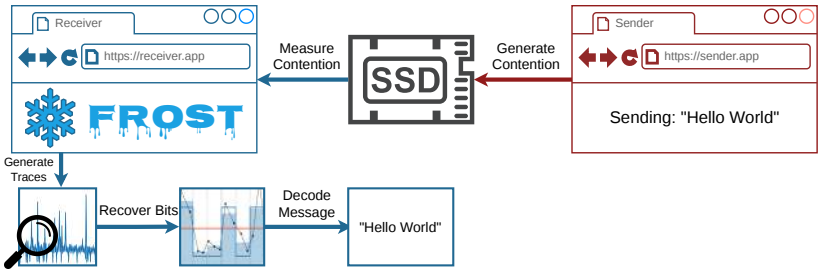


Fig. 3. Visualization of the covert channel setup. The sender can be a native application writing to a file, or a browser application writing to a user-selected file or OPFS. The receiver is a malicious website running in the browser, monitoring SSD contention via OPFS or a user-selected file.

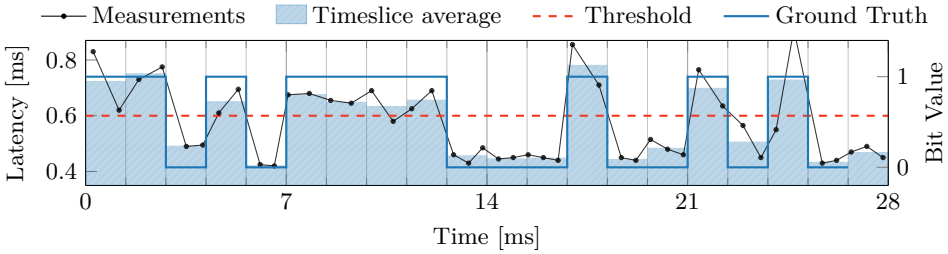


Fig. 4. Sample trace of a covert channel trace using OPFS, with timeslices of 1.4 ms. The receiver finds the threshold and phase of the signal dynamically by optimizing for the smallest possible bit error rate in the known pre- and postamble. By averaging all measurements within a timeslice, the receiver minimizes the error rate.

prompt the user to select a file whose size exceeds the available physical memory, which allows us to perform continuous measurements on it. In OPFS mode, we do not prompt the user, and instead create a suitable file in the OPFS, filling it with random data to avoid interference by page deduplication mechanisms. We perform all experiments with user-selected files and OPFS to evaluate the performance of both measurement primitives.

5 Covert Channel

To measure the capacity of the SSD contention side channel, we first build a covert channel between two cooperating parties on the same system. The sender and receiver are two separate processes, with the sender generating SSD contention to encode a message, while the receiver continuously measures the contention levels to decode the message. The receiver is running in JavaScript within the browser, while the sender is a native application, as shown in Figure 3. To transmit data, the sender encodes bits by either generating contention (bit 1) or remaining idle (bit 0). By splitting the trace into timeslices, the receiver can

Table 1. Covert channel capacity for different raw transmission rates on Linux and macOS, for both OPFS and user-selected files.

Timeslice Length	**Sender Rate**	**Linux**				**macOS**			
		User-selected		OPFS		User-selected		OPFS	
		BER[1]	TC[2]	BER	TC	BER	TC	BER	TC
[ms]	[bit/s]	%	[bit/s]	%	[bit/s]	%	[bit/s]	%	[bit/s]
1.5	666.67	4.81	481.08	6.15	444.39	5.61	458.86	4.80	481.48
1.4	714.29	4.83	514.94	6.81	458.04	5.46	495.84	5.69	489.20
1.3	769.23	5.42	535.48	6.76	494.69	6.47	503.14	9.87	411.61
1.2	833.33	6.44	546.27	6.29	550.84	6.14	555.69	8.35	487.90
1.1	909.09	7.35	564.60	6.28	601.31	5.79	619.00	7.99	543.79
1.0	1 000.00	8.97	564.66	7.38	620.01	6.15	666.68	9.37	551.18
0.9	1 111.11	11.04	554.35	9.38	612.24	8.88	630.80	10.78	562.98
0.8	1 250.00	13.04	551.79	10.05	661.63	8.44	728.13	12.16	582.57
0.7	1 428.57	17.32	478.45	14.21	586.03	8.25	841.28	10.89	719.27
0.6	1 666.67	22.21	393.44	24.55	326.66	9.87	891.77	15.21	641.34
0.5	2 000.00	37.27	94.54	25.64	357.58	21.02	516.41	25.54	360.67

[1] Bit Error Rate [2] True Capacity

decode the transmitted bits by computing the average of all samples within a timeslice and using a threshold value for the access time to decide on the value as shown in Figure 4. To synchronize the sender and receiver, the sender starts with a known preamble, followed by the transmitted data. We assume that the receiver already knows the length of the transmitted data, either via an agreed upon fixed message length, a secondary channel, or a protocol building upon the covert channel itself. The receiver continuously measures SSD latencies and saves the resulting trace for offline decoding. We evaluate the covert channel using both user-selected files and OPFS files for contention measurement.

Results. To decode the transmitted data, the receiver needs to determine the offset of the covert message within a trace and the correct threshold to distinguish ones and zeroes. We assume that sender and receiver agree on a known message size and add 11-byte preambles and postambles with known contents to the transmitted data. Thus, the receiver knows the distance between preamble and postamble before the transmission. To determine the correct offset and threshold for decoding, the receiver iterates over the trace while sweeping these parameters, detects candidate preambles and postambles, and ranks them by their bit-error count against the known data. Afterwards, the receiver decodes the payload using the found parameters. In our experiments, this strategy was very reliable and consistently found the correct markers in the trace.

The performance of the covert channel depends on the timeslice length per bit and the chunk size used to generate SSD contention. For each system, we empirically choose the latency measurement read size. We then evaluate timeslice lengths of 0.5 ms to 1.5 ms in steps of 0.1 ms.

Linux. On our Linux machine, we use a read size of 128 kB. Using user-selected files, we achieved the maximum throughput with timeslices of 1 ms, achieving a bit error rate of 8.97 % for a true capacity of 564.66 bit/s. When using OPFS, we achieve a bit-error-rate of 10.05 % for a true capacity of 661.63 bit/s with a timeslice length of 0.8 ms. Thus, on Linux, we achieve better covert channel performance using OPFS compared to user-selected files, while removing the requirement for user interaction. This makes the attack significantly more practical compared to user-selected files. In Figure 4, we show a sample trace of a covert channel transmission using OPFS with timeslices of 1.4 ms. In Table 1, we show a full overview of the results. Even with these fairly long timeslices, we only achieve a maximum of 3 measurements per timeslice when no contention is present, and 2 with contention. Consistent with this observation, the bit error rate increases sharply below timeslices of 0.7 ms, as the duration of transmitted ones falls below our measurement interval. Interestingly, for timeslices between 1.5 ms and 1.1 ms, we experienced a slight decrease in bit error rate with smaller timeslices. The observed behavior may fall within measurement error. However, it may also stem from increased scheduling interference due to the longer transmission time with larger timeslices.

macOS. On macOS, we determine the optimal read size to be 512 kB. We find that, for a user-selected file, the optimal timeslice length is 0.6 milliseconds, achieving a true capacity of 891.77 bits per second with a bit error rate of 9.87 %. When using OPFS, we achieve a true capacity of 719.27 bits per second with a bit error rate of 10.89 % for a timeslice length of 0.7 ms. The development of the error rate is similar to Linux, with a slow increase, followed by a steep rise when the timeslices become too short. In contrast to our Linux results, user-selected files achieve a higher maximum throughput than OPFS files on macOS. We assume that this difference in behavior between the two operating systems may be caused by different prioritization of file reads using the File System Access API, compared to the OPFS API. Still, we achieve a higher true capacity on macOS than on Linux when using OPFS, demonstrating the accuracy of this contention measurement.

Cross-Browser Comparison. For our fingerprinting attack, described in Section 6.2, we run the attack cross-browser to simplify measurement automation, *i.e.*, the malicious website runs in a different browser than the victim. To demonstrate that cross-browser attacks and same-browser cross-tab attacks have similar leakage rate, we re-implemented the covert channel sender in JavaScript to measure covert channel speeds in both scenarios. Using the JavaScript-based sender, our covert channel performance drops significantly, as we cannot generate as much contention from JavaScript as from a native application. When testing our covert channel between two tabs of Chrome, we achieve a true capacity of 65.06 bit/s. Meanwhile, when sending data from Safari to Chrome using the same covert channel parameters, we achieve 67.26 bit/s, *i.e.*, a difference of 3.38 %, which is within the margin of error. Thus, cross-browser attacks and cross-tab attacks have comparable leakage rates.

Performance comparison. Gu et al. [25] report a raw capacity of 940 bit/s with an error rate of 0.01 % by timing syncfs operations, resulting in a true capacity of 938.61 bit/s using the `sync` operation on Linux, which is significantly faster than our covert channel. However, their approach requires access to the `sync` command, which is not available from JavaScript. Lipinski et al. [38] use HDD contention to build a covert channel with 0.1 bit/s. Giechaskiel et al. [19] exploit SSD contention on AWS to build a 0.125 bit/s covert channel between AWS instances. Trochatos et al. [70] showed a covert channel using SmartSSDs with an integrated FPGA achieving 0.066 bit/s with a 25 % error rate.

Compared to Juffinger et al. [30], we achieve a *higher throughput* than *any* of their SSDs when using a 1 000 bit/s raw transmission rate. However, we achieve a slower maximum throughput than most of their SSDs. While the results are not directly comparable because we use a different SSD model, our results indicate that SSD contention measurements from the browser are highly reliable, but less fine-grained than those from native code.

In conclusion, our covert channel experiments show that browser-based SSD contention measurements achieve a similar reliability to native measurements, albeit with a lower resolution. We also observe that OPFS not only eliminates the need for user interaction, but it also achieves a higher throughput compared to user-selected files on Linux. While user-selected files achieve a higher throughput on macOS, OPFS still outperforms the maximum true capacity on Linux.

6 FROST Fingerprinting Attacks

In this section, we present *FROST*, a remote side-channel attack that uses SSD contention from within the browser to fingerprint user activity on the host system. We describe the attack setup and present two practical fingerprinting attacks: website fingerprinting and application fingerprinting.

6.1 Attack Setup

To mount FROST, the victim first visits an attacker-controlled website, e.g., through a seemingly benign website, advertisements, or spam. When using OPFS, no further user interaction is required, and the attack code runs in the background while the victim performs other activities on the system. The attacker continuously measures SSD contention by performing random reads from a large OPFS file. SSD contention caused by user activity causes measurable latency differences for these read operations. By training a convolutional neural network (CNN) on these traces, the attacker can fingerprint user activity on the host system by classifying new traces using the trained model.

Our attacker server is implemented in Python using Flask while the client-side code is implemented in JavaScript. We collect data on a Mac mini with an M2 chip, running macOS Sonoma 14.2.1. Automation of browser- and application actions is performed using macOS's built-in automation framework. We report the fingerprinting results focusing on macOS due to easier automation but emphasize that the attack works identically on Linux, see also Section 5.

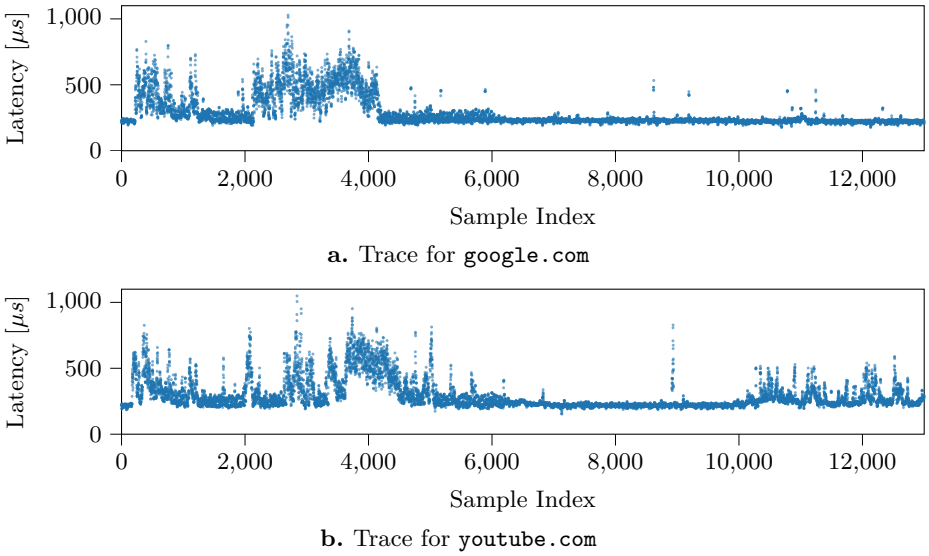


Fig. 5. Traces for `google.com` and `youtube.com` recorded with FROST using an OPFS file. Both are clearly distinguishable, with `google.com` only having a small spike around index 11 000, while `youtube.com` causes higher latency spikes for a longer amount of time. Both traces start with a similar pattern, which is caused by Safari starting up.

6.2 Website Fingerprinting

For our FROST website-fingerprinting attack, we first generate training data for our CNN model. We collect traces while visiting the top 50 websites from the Alexa Top Million list [4], generating 100 traces per website, resulting in a labeled dataset of 5 000 traces. Additionally, we collect 1 trace each for the next 300 websites in the list as an open-world set. Our attack setup is symmetric, *i.e.*, the attacker and victim can use the same or different browsers, as the attack works on the storage level, independent of the browser. Hence, we focus on one setup that we evaluate in depth, namely a cross-browser attack from an attacker page in Google Chrome, targeting websites in Safari (version 17.2.1). This approach also shows that FROST works across different browsers; however, the attack does not depend on two different browsers being used, as the contention occurs on the storage level, irrespective of the browser, as we also show in Section 5.

We show example traces for the websites `google.com` and `youtube.com` in Figure 5. They are clearly distinguishable by eye, with `google.com` only having a short latency spike, while `youtube.com` has higher latencies for longer periods of time. The traces also show a similar shape in the beginning, caused by Safari starting up. In Section 6.3, we fingerprint different applications by classifying their SSD latency patterns during application startup.

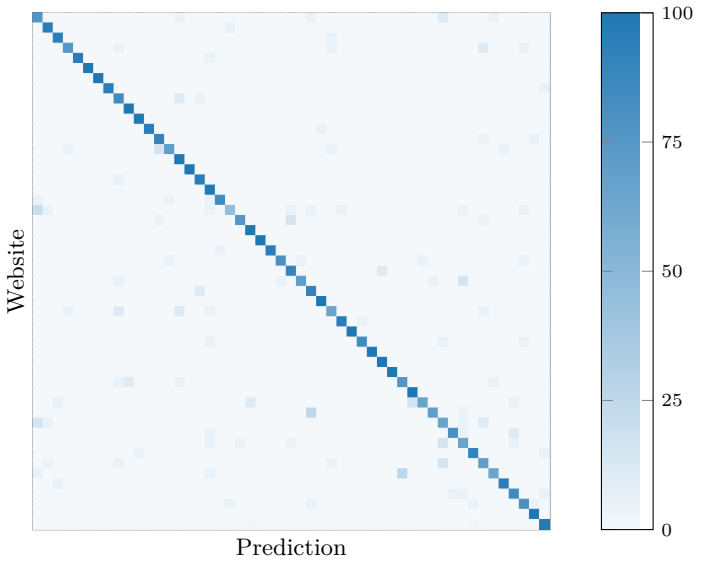


Fig. 6. The confusion matrix for website fingerprinting using FROST using OPFS, achieving a macro-averaged F_1 score of 86.95 % with an open-world accuracy of 96.72 %.

Evaluation. We measure traces for 10 s each, opening the website 2 s after starting the measurement. After traces have been collected, we preprocess them, by filtering out spikes as described in Section 4 and truncating them to equal lengths. To classify the traces, we use a convolutional neural network (CNN) with four convolutional blocks (64, 128, 256, 512 filters), batch normalization and dropout, followed by two dense layers (512, 256 units). We train the model using 60 % of the traces for training, 20 % for validation, and 20 % as a hold-out test set. Section A provides the details of an attack with user-selected files, achieving an open-world F_1 score of 85.60 %. When using OPFS for contention measurements, we achieve a macro-averaged F_1 score of 86.95 % in an open-world setting and 88.95 % in a closed-world setting.

In comparison, prior fingerprinting attacks from the browser achieve *accuracies* of 70.5 % to 97.3 % using the Quota Management API (storage space) [33], 82.1 % to 88.6 % using CPU cache attacks [51], 78.89 % to 82.86 % by timing the DNS cache [76], and 76.7 % accuracy using shared browser event loops [73]. Thus, FROST performs similarly to prior browser-based fingerprinting attacks while using a novel OPFS-based approach to obtain SSD contention measurements without user interaction. Closest to our work, Juffinger et al. [30] use native code for SSD contention measurements and achieve F_1 scores of 78 % to 96 % (open-world), and 80.2 % to 97 % (closed-world) on the top 100 websites, depending on the specific SSD used. While our F_1 score is lower compared to native SSD-based attacks on most SSDs [30], the data was recorded from within a browser, without features such as `io_uring`. Our results show that SSD-based

Table 2. Applications selected for fingerprinting attack.

Applications	
Contacts	Music
Calculator	App Store
Keynote	TV
Weather	System Settings
Maps	Safari

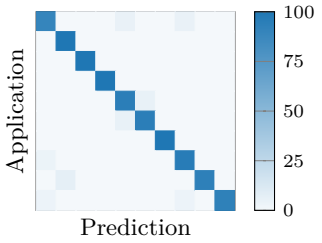


Fig. 7. The confusion matrix for application fingerprinting using FROST, using OPFS to avoid user interaction. We reach a F_1 score of 95.83 % in an closed-world setting.

leakage can be effectively exploited even within a sandboxed environment, without the need for low-overhead primitives, as long as the sandbox has access to a sufficiently large file on the SSD, e.g., via OPFS. We conclude that FROST performs similarly to prior browser-based attacks, while using a novel OPFS-based approach to achieve SSD contention measurements without any user interaction.

6.3 Application Fingerprinting

In this section, we mount an application-fingerprinting attack using FROST. We select 10 native, pre-installed applications on macOS, listed in Table 2. We chose applications that do not require login or specific projects to avoid biasing the fingerprinting process with personalized settings or data. For this measurement, we use only OPFS, as it reflects a broader threat model. To generate traces, we measure SSD contention for 10 seconds, launching the target applications 2 seconds into the measurement. We then stop the measurement, close the application, and for the repeated measurements, restore the previous system state for the page cache to avoid interference between measurements. We repeat this process 100 times for each application, resulting in a total of 1 000 traces.

Evaluation. We measure traces for 10 seconds each, which covers the startup time of all tested applications. We perform the same preprocessing and training procedure as in Section 6.2. After training the model, we reach an F_1 score of 95.83 % in our closed-world experiment. In Figure 7, we show the confusion matrix of our results using an OPFS file. This experiment shows that FROST can not only detect website accesses, but general system activity as well.

7 Discussion

The File System Access API is a powerful feature that allows web applications to interact with the user's file system, enabling a wide range of applications, such as image- and video editors or IDEs, to run within the browser. Modern browsers are heavily sandboxed environments, designed to prevent malicious code from accessing sensitive system resources. Thus, remote full-chain exploits, which gain code execution from a website visit, are fairly rare in the wild, and often require exploiting multiple vulnerabilities in the browser or operating system. In contrast, remote side-channel attacks, such as FROST, can be mounted without bypassing any security mechanisms, as they exploit unintentional information leakage from legitimate operations. Additionally, side-channel attacks are often hard to defend against, as they exploit side effects of normal system behavior. Modern web browsers are complex pieces of software, continuously evolving to support new features and use cases. However, as we have shown in this paper, such powerful features can also allow attackers to port previous native side-channel attacks to the browser, enabling remote side-channel attacks. Many new features are designed to provide near-native application behavior. Without careful consideration of the security implications, these features can inadvertently also introduce near-native side-channel leakage, allowing new remote exploits. Thus, while typical Internet users may not be targeted by sophisticated sandbox escapes to achieve native remote code execution, their browsers leak much more information about their activities than intended, allowing attackers to spy on them remotely, *i.e.*, for advertising or blackmail purposes.

Limitations. To allow FROST to measure SSD contention, the victim must perform activities that result in storage accesses to the same disk as the file used for contention measurement. When using OPFS, the file is stored in the browser's default storage location, allowing attackers to always perform website fingerprinting attacks on the same browser. However, application fingerprinting attacks require the victim to use applications that access the same disk as the browser. On most consumer devices, this is not a significant limitation, as they typically only have one internal SSD. In contrast, large and powerful workstations may have multiple disks, with heavy-duty tasks like video editing or software development using different disks than the browser. However, many applications still access the user's home directory for configuration- or project files, which are accessed during startup and generate measurable contention.

Because the page cache prevents repeated accesses to the same data from hitting the SSD, and we do not have the system-level features to bypass the page cache (e.g., the `O_DIRECT` flag for the `open` syscall), long-running measurements require a large file to evade the page cache. Short, single-shot measurements are possible with smaller files. However, in the case of OPFS, the file is created by the attacker script when it is first run, which implicitly loads the file into the page cache. Thus, it is not possible to run a short one-shot measurement using OPFS, without first evicting the file from the page cache. While it is possible

that the victim re-visits the attacker's page after clearing the page cache, e.g., by rebooting, this scenario is significantly less likely than our threat model.

On browsers without OPFS support, FROST requires the user to select a large file for contention measurement. According to documentation [44], OPFS is supported in all major desktop browsers since 2023. While Chromium-based browsers and Safari allow websites to use up to 60 % of total disk space for OPFS storage, Firefox only allows up to 10 GB per origin (*i.e.*, website). However, a more advanced attack using multiple origins can bypass this limit, as each origin has its own OPFS storage. Furthermore, the attacker can request persistent storage permissions, requiring user intervention, to bypass the limit. Although browsers do not prominently indicate OPFS disk usage, users monitoring their available disk space could notice the attack's high storage consumption.

Mitigations. Mitigating side-channel attacks is often challenging without restricting legitimate functionality. A possible mitigation for FROST is to restrict the total used storage space for OPFS files without explicit persistent storage permissions. By limiting the maximum file size to e.g., 1 GB, the file fits into system memory, preventing SSD accesses and thus eliminating the contention effects. However, an advanced attacker could use multiple origins to bypass this limitation by creating multiple maximum-size OPFS files. To prevent this, browsers could implement tracking for OPFS behavior, notifying users when large amounts of data are saved to the OPFS across multiple origins, within a short time frame. Another mitigation is to consider file system access a cross-origin resource, and to restrict access to high-resolution timers when OPFS is used. Ultimately, the most effective mitigation would be to enable OPFS only after explicit user permission, which would significantly harm the usability of OPFS for legitimate applications and cause disruptions to user workflows. Additionally, when users are trained to automatically accept permission requests for any web-based application, they will likely accept attacker's requests as well, making this mitigation less effective in practice [72].

During our research, we encountered one system running `profile-sync-daemon` (PSD). PSD moves browser profiles into a tmpfs directory, syncing them back to the hard drive only occasionally to improve performance of the browser. Thus, on systems with PSD enabled, activity on OPFS files does not result in any SSD accesses, since they are not stored on the SSD. However, PSD only prevents the zero-interaction OPFS attack; an attack using a user-selected file is still possible if the attacker can trick the victim into selecting a file on the SSD.

Related Work. Website fingerprinting is a common attack target to showcase the effectiveness of a side channel. Oren et al. [51] used Prime+Probe to build a covert channel and website fingerprinting attack from the browser, achieving an accuracy of 82.1 % when detecting website accesses out of 8 candidates on Safari. As timer precision in the JavaScript engine was reduced to hinder side-channel attacks, Shusterman et al. [63] showed that Prime+Probe was still possible using different timing sources, demonstrating website fingerprinting in the TOR

browser. Cache occupancy has also been exploited for fingerprinting from the browser, on the CPU for an open-world accuracy of 87.7 % (100 websites) [64], and on the GPU for an open-world F_1 score of up to 89.8 % (100 websites) [13].

Others exploit side channels originating from implementation details of software, such as the browser, operating system, or system services. Kim et al. [33] track the available storage quota from JavaScript, to identify visited websites with an accuracy of 97.3 % in a closed-world setting (100 websites). Vila et al. [73] detect website accesses by measuring timing behavior caused by the shared event loop in Chrome, for an accuracy of up to 76.7 % (closed-world, 500 websites). While not strictly fingerprinting, Gruss et al. [21] infer opened websites by exploiting page deduplication mechanisms in Linux, achieving a false-positive rate of 0.3 % to 1.1 %. Similarly, Weissteiner et al. [76] determine whether a website was recently accessed by measuring whether or not it's domain is in the DNS cache, achieving an F_1 score of up to 82.86 %.

8 Conclusion

In this paper, we presented FROST, a novel remote side-channel attack that leaks sensitive information from within the browser using SSD contention on both macOS and Linux systems. Unlike prior work, which requires native code execution, FROST can be performed as a drive-by attack through JavaScript embedded on a website. We discovered that the private file system API OPFS, included in most major browsers, allows an attacker to perform precise SSD latency measurements, enabling a wide range of attacks. By exploiting OPFS, we built a high-speed covert channel with a capacity of 661.63 bit/s based on SSD contention, comparable to native performance. We used FROST to perform website and application fingerprinting based on this contention channel from JavaScript with F_1 scores of 88.95 %, and 95.83 % respectively, demonstrating the security implications of our new attack. Our results show that direct low-latency filesystem access from the browser can pose a significant threat to a user's privacy, and that additional protections, e.g., a pop-up requesting permission to use OPFS, are required to mitigate our attacks.

Acknowledgments

This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM 10.55776/I6054 and FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Red Hat, Intel, and Google. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Aksoy, A., Louis, S., Gunes, M.H.: Operating System Fingerprinting via Automated Network Traffic Analysis. In: Congress on Evolutionary Computation (2017)
2. Al-Naami, K., Chandra, S., Mustafa, A., Khan, L., Lin, Z., Hamlen, K., Thuraisingham, B.: Adaptive Encrypted Traffic Fingerprinting With Bi-Directional Dependence. In: ACSAC (2016)
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port Contention for Fun and Profit. In: S&P (2019)
4. Alexa Internet, Inc.: The top 1 million sites on the web (5 2023), `https://www.alexa.com/topsites`
5. Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., Sporleder, C.: Acoustic Side-Channel Attacks on Printers. In: USENIX Security (2010)
6. Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., Butler, K.: On detecting co-resident cloud instances using network flow watermarking techniques. International Journal of Information Security **13**, 171–189 (2014)
7. Bissias, G.D., Liberatore, M., Jensen, D., Levine, B.N.: Privacy Vulnerabilities in Encrypted HTTP Streams. In: PETS (2006)
8. Boukhobza, J., Olivier, P.: Flash Memory Integration: Performance and Energy Issues. Elsevier (2017)
9. Chen, Y., Jin, X., Sun, J., Zhang, R., Zhang, Y.: POWERFUL: Mobile app fingerprinting via power analysis. In: INFOCOM (2017)
10. Cherubin, G., Jansen, R., Troncoso, C.: Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World. In: USENIX Security (2022)
11. Dusi, M., Crotti, M., Gringoli, F., Salgarelli, L.: Tunnel Hunter: Detecting Application-Layer Tunnels with Statistical Fingerprinting. Computer Networks **53**(1), 81–97 (2009)
12. van Ede, T., Bortolameotti, R., Continella, A., Ren, J., Dubois, D., Lindorfer, M., Choffnes, D., van Steen, M., Peter, A.: FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In: NDSS (2020)
13. Ferguson, E., Wilson, A., Naghibijouybari, H.: WebGPU-SPY: Finding Fingerprints in the Sandbox through GPU Cache Attacks. In: AsiaCCS (2024)
14. Frigo, P., Giuffrida, C., Bos, H., Razavi, K.: Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P (2018)
15. Gast, S., Czerny, R., Juffinger, J., Rauscher, F., Franza, S., Gruss, D.: SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript. In: USENIX Security (2024)
16. Gast, S., Juffinger, J., Maar, L., Royer, C., Kogler, A., Gruss, D.: Remote Scheduler Contention Attacks (Extended Version). arXiv:2404.07042 (2024)
17. Gast, S., Juffinger, J., Schwarzl, M., Saileshwar, G., Kogler, A., Franza, S., Köstl, M., Gruss, D.: SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P (2023)
18. Genkin, D., Pachmanov, L., Tromer, E., Yarom, Y.: Drive-by Key-Extraction Cache Attacks from Portable Code. In: ACNS (2018)
19. Giechaskiel, I., Tian, S., Szefer, J.: Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. ACM Transactions on Reconfigurable Technology and Systems (2022)
20. Giner, L., Czerny, R., Gruber, C., Rauscher, F., Kogler, A., De Almeida Braga, D., Gruss, D.: Generic and Automated Drive-by GPU Cache Attacks from the Browser. In: AsiaCCS (2024)

21. Gruss, D., Bidner, D., Mangard, S.: Practical Memory Deduplication Attacks in Sandboxed JavaScript. In: ESORICS (2015)
22. Gruss, D., Kraft, E., Tiwari, T., Schwarz, M., Trachtenberg, A., Hennessey, J., Ionescu, A., Fogh, A.: Page Cache Attacks. In: CCS (2019)
23. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA (2016)
24. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security (2015)
25. Gu, C., Zhang, Y., Abu-Ghazaleh, N.: I know What You Sync: Covert and Side Channel Attacks on File Systems via syncfs. In: S&P (2025)
26. Gulmezoglu, B., Zankl, A., Eisenbarth, T., Sunar, B.: PerfWeb: How to violate web privacy with hardware performance events. In: ESORICS (2017)
27. Haas, G., Aysu, A.: Apple vs. EMA: Electromagnetic Side Channel Attacks on Apple CoreCrypto. In: Design Automation Conference (2022)
28. Hintz, A.: Fingerprinting Websites Using Traffic Analysis. In: PETS (2003)
29. Jiang, Q., Wang, C.: Sync+Sync: A Covert Channel Built on fsync with Storage. In: USENIX Security (2024)
30. Juffinger, J., Rauscher, F., La Manna, G., Gruss, D.: Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS (2025)
31. Juffinger, J., Weissteiner, H., Steinbauer, T., Gruss, D.: The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer. In: DIMVA (2025)
32. Juffinger, J., Weissteiner, H., Steinbauer, T., Gruss, D.: The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer. In: DIMVA (2025)
33. Kim, H., Lee, S., Kim, J.: Inferring Browser Activity and Status Through Remote Monitoring of Storage Usage. In: ACSAC (2016)
34. Kim, S., Na, S.H., Kim, J., Shin, S., Choi, H.: AVXProbe: Enhancing Website Fingerprinting with Side-Channel-Assisted Kernel-Level Traces. In: AsiaCCS (2025)
35. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)
36. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
37. LePage, Pete and Steiner, Thomas: The File System Access API: simplifying access to local files (2024), `https://developer.chrome.com/docs/capabilities/web-apis/file-system-access`
38. Lipinski, B., Mazurczyk, W., Szczypiorski, K.: Improving Hard Disk Contention-based Covert Channel in Cloud Computing Environment. In: S&P Workshops (2014)
39. Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C.m.t.n., Mangard, S.: Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS (2017)
40. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security (2018)
41. Liu, S., Kanniwadi, S., Schwarzl, M., Kogler, A., Gruss, D., Khan, S.: Side-Channel Attacks on Optane Persistent Memory. In: USENIX Security (2023)
42. Matyunin, N., Wang, Y., Arul, T., Kullmann, K., Szefer, J., Katzenbeisser, S.: MagneticSpy: Exploiting Magnetometer in Mobile Devices for Website and Application Fingerprinting. In: ACM Workshop on Privacy in the Electronic Society (2019)
43. Mozilla: IndexedDB API - Web APIs (2025), `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API`

44. Mozilla: Origin Private File System - Web APIs (2025), https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system
45. Mozilla: Storage quotas and eviction criteria (2025), https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria
46. Mozilla: WebGPU - Web APIs (2025), https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API
47. Mozilla: WebUSB API - Web APIs (2025), https://developer.mozilla.org/en-US/docs/Web/API/WebUSB_API
48. Mozilla: Window.localStorage - Web APIs (2025), https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
49. Neela, S.R., Juffinger, J., Maar, L., Gruss, D.: Eviction Notice: Reviving and Advancing Page Cache Attacks. In: NDSS (2026)
50. NVM Express, Inc: NVM Express, rev 1.2.1 (2016)
51. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)
52. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
53. Panchenko, A., Lanze, F., Pennekamp, J., Engel, T., Zinnen, A., Henze, M., Wehrle, K.: Website Fingerprinting at Internet Scale. In: NDSS (2016)
54. Randolph, M., Diehl, W.: Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. Cryptography **4**(2) (2020)
55. Rauscher, F., Fiedler, C., Kogler, A., Gruss, D.: A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS (2025)
56. Rauscher, F., Gruss, D.: Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: CCS (2024)
57. Ren, X., Moody, L., Taram, M., Jordan, M., Tullsen, D.M., Venkat, A.: I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In: ISCA (2021)
58. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS (2009)
59. Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y.: Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS (2022)
60. Rokicki, T., Maurice, C., Schwarz, M.: CPU Port Contention Without SMT. In: ESORICS: European Symposium on Research in Computer Security (2022)
61. Saileshwar, G., Fletcher, C.W., Qureshi, M.: Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS (2021)
62. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)
63. Shusterman, A., Agarwal, A., O'Connell, S., Genkin, D., Oren, Y., Yarom, Y.: Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In: USENIX Security (2021)
64. Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security (2019)
65. Spolaor, R., Liu, H., Turrin, F., Conti, M., Cheng, X.: Plug and Power: Fingerprinting USB Powered Peripherals via Power Side-channel. In: IEEE INFOCOM (2023)

66. Spreitzer, R., Griesmayr, S., Korak, T., Mangard, S.: Exploiting data-usage statistics for website fingerprinting attacks on Android. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks (2016)
67. Tan, M., Wan, J., Zhou, Z., Li, Z.: Invisible Probe: Timing Attacks with PCIe Congestion Side-Channel. In: S&P (2021)
68. Trampert, L., Hetterich, L., Gerlach, L., Schappert, M., Rossow, C., Schwarz, M.: Peripheral Instinct: How External Devices Breach Browser Sandboxes. In: WWW (2025)
69. Trampert, L., Rossow, C., Schwarz, M.: Browser-Based CPU Fingerprinting. In: ESORICS (2022)
70. Trochatos, T., Etim, A., Szefer, J.: Covert-channels in FPGA-enabled SmartSSDs. ACM Transactions on Reconfigurable Technology and Systems (2023)
71. Van Goethem, T., Joosen, W.: One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In: WOOT (2017)
72. Vance, A., Jenkins, J.L., Anderson, B.B., Bjornn, D.K., Kirwan, C.B.: Tuning out Security Warnings: A Longitudinal Examination of Habituation Through fMRI, Eye Tracking, and Field Experiments. MIS Quarterly **42**(2), 355–380 (2018)
73. Vila, P., Köpf, B.: Loophole: Timing Attacks on Shared Event Loops in Chrome. In: USENIX Security (2017)
74. Wang, T., Goldberg, I.: Improved Website Fingerprinting on Tor. In: WPES (2013)
75. WebDev: Why you need "cross-origin isolated" for powerful features (2020), `https://web.dev/articles/why-coop-coep`
76. Weissteiner, H., Czerny, R., Franza, S., Gast, S., Ullrich, J., Gruss, D.: Continuous User Behavior Monitoring using DNS Cache Timing Attacks. In: NDSS (2026)
77. Zaheri, M., Oren, Y., Curtmola, R.: Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses. In: USENIX Security (2022)
78. Zhang, K., Li, Z., Wang, R., Wang, X., Chen, S.: Sidebuster: Automated Detection and Quantification of Side-Channel Leaks in Web Application Development. In: CCS (2010)
79. Zhang, R., Kim, T., Weber, D., Schwarz, M.: (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security (2023)

A Attacks with user-selected files

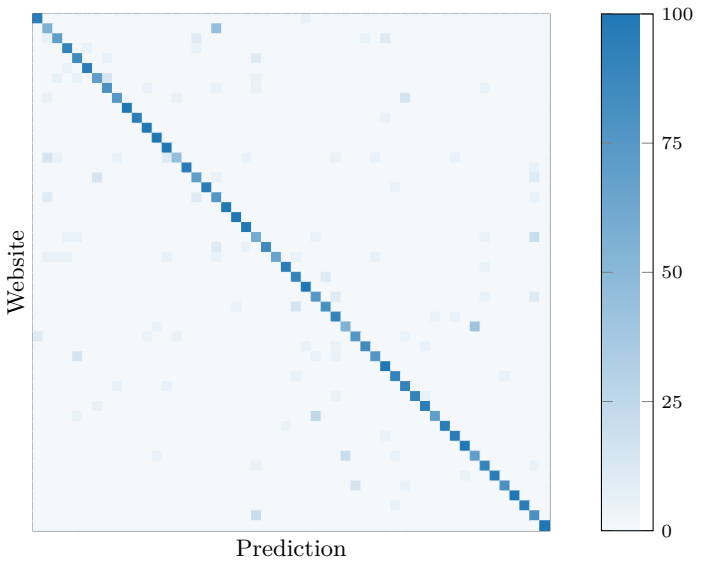


Fig. 8. The confusion matrix for website fingerprinting using FROST, when using user-selected files selected by the user. We achieve a macro-averaged F_1 score of 85.60 % in an open-world setting.

In this appendix, we present the results of website fingerprinting using user-selected files for SSD contention measurement. We consider this scenario weaker, as it requires user interaction. However, it is still relevant for older browsers that do not support OPFS, and to show the impact of using OPFS on the attack performance. Using user-selected files, our trained model achieves a macro-averaged F_1 score of 85.60 % in an open-world setting, demonstrating the effectiveness of FROST for website fingerprinting from the browser. In a closed-world setting, we reach an F_1 score of 87.27 %. This is very similar to the results on OPFS, which had a macro-averaged F_1 score of 86.95 % in an open-world setting and 88.95 % in a closed-world setting. Thus, the results are in line with our covert channel results, where the performance with OPFS and user-selected files was comparable. We show the confusion matrix for user-selected files in Figure 8.