Hidden in Plain Sight: Scriptless Microarchitectural Attacks via TrueType Font Hinting

Leon Trampert[0009−0001−6891−965X] and Michael Schwarz[0000−0001−6744−3410]

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{leon.trampert,michael.schwarz}@cispa.de

Abstract. Microarchitectural attacks threaten system security and privacy, especially if they can be mounted without native code execution. Recent research has shown that such attacks are possible from within web browsers via JavaScript and WebAssembly. Moreover, recent works have demonstrated that "scriptless" attacks, using only CSS, can be leveraged for side-channel attacks, including cache contention and user fingerprinting.
In this paper, we introduce a new class of scriptless attacks that use the hinting instructions embedded within TrueType font files. We show that the hinting language is sufficiently robust to craft cache attacks, demonstrating cache-contention attacks and precise L1 Prime+Probe attacks. We demonstrate a website fingerprinting attack, as well as a method to track which page of a PDF is currently displayed. Our results demonstrate the practicality of font-based scriptless attacks in real-world scenarios. This emphasizes the need for future mitigations that go beyond traditional scripting languages.

1 Introduction

Microarchitectural attacks present a significant threat to modern computing systems, as they can compromise system security and privacy. They have been shown to steal cryptographic keys [20], or spy on user behavior [22, 36]. Particularly concerning is that these attacks can be mounted without native code execution, widening the range of adversaries and increasing their stealth.

Previous work has shown that such attacks are possible from within web browsers via JavaScript [2,25,34,38] and WebAssembly [38]. This includes cache attacks [34], interrupt-based attacks [19], Rowhammer exploits [13], and Spectre attacks [2]. Consequently, substantial efforts have focused on developing mitigations and hardening strategies to mitigate the impact of these attacks on web-based environments [31, 32].

Recent research has pushed this concept further by exploring so-called "scriptless" attacks [33]–those mounted without the use of conventional scripting or programming languages. For instance, recent works have highlighted how CSS alone can be weaponized for side-channel attacks, including exploiting cache

contention [33] and enabling user fingerprinting [18, 39], thereby increasing the attack surface. These works already suggest that existing assumptions about what constitutes "code execution" may be too narrow.

In this paper, we introduce a new class of scriptless attacks by leveraging the hinting code embedded within TrueType font files [3]. We exploit the fact that custom fonts–common in websites, emails, and PDF documents–trigger the execution of a complex hinting program on the victim system when text is simply rendered. Unlike prior techniques [33, 34], this method does not depend on JavaScript, WebAssembly, or even complex CSS, thus evading conventional detection and prevention strategies. As a result, this mechanism allows attackers to run attacker-provided code without relying on traditional scripting languages.

We demonstrate that the font hinting language provides sufficient capabilities for microarchitectural attacks. In particular, we show cache-contention attacks similar to those introduced by Shusterman et al. [33], and we further advance this approach to achieve fine-grained Prime+Probe attacks across hyperthreads.

To show the practical impact, we present a website fingerprinting attack, as well as a covert tracking method that allows an attacker to identify the current page in a PDF. Our website fingerprinting has an accuracy of 71 % to 81 %, which is on par with previous work on JavaScript-based website fingerprinting [34], and outperforms scriptless CSS-based fingerprinting [33]. Our attacks emphasize the severity and versatility of font-based scriptless attacks in real-world scenarios.

Our findings highlight that existing defenses must be reevaluated and broadened. The traditional focus on script-based code execution, such as JavaScript and WebAssembly, is no longer sufficient. Future mitigations must consider a broader range of potential triggers and attack surfaces, including the execution of hinting instructions embedded within fonts.

Contributions. We summarize our contributions as follows.

- We analyze the availability of TrueType hinting instructions in different environments, showing that they can be used for microarchitectural attacks.
- We demonstrate a website fingerprinting attack in the browser using font hinting instructions, achieving an accuracy comparable with state-of-the-art script-based attacks.
- We present a method for evicting chosen L1 cache sets, enabling a cross-hyperthread covert channel in settings such as PDFs or emails.

Availability. Our artifact is available at https://github.com/cispa/fontention.

2 Background

In this section, we provide the background required for this paper. We briefly introduce cache attacks, especially in the limited code execution setting, and briefly introduce TrueType fonts.

2.1 Cache Attacks & Defenses

Microarchitectural side-channel attacks exploit variations in execution time and resource contention at the level of CPU caches, branch predictors, and other

low-level hardware structures. Among these, cache attacks are especially well-studied, with techniques such as Prime+Probe [26, 27] allowing an attacker to measure the cache activity of specific cache sets in shared caches. By observing subtle timing differences across repeated memory accesses, an attacker can infer whether the victim accesses memory mapping to a monitored cache set. Depending on the victim, attackers can infer sensitive information, e.g., cryptographic keys [26] or user behavior [34]. Variants of these attacks have been demonstrated in environments ranging from native code execution [26, 27] to more constrained settings such as (sandboxed) JavaScript within web browsers [12, 13, 25, 30].

For web browsers, a generic defense is the reduction of high-resolution timers [23] and disabling methods for building timers [32], such as shared memory [15]. Consequently, researchers have also explored alternative vectors, such as CSS-based cache contention techniques [33]. These methods take advantage of the fact that rendering engines must still process and layout style information, even when JavaScript is unavailable. By embedding large memory-consuming CSS rules [33], an attacker can measure cache contention, i.e., the overall activity in the cache. Such methods circumvent mitigations that solely focus on JavaScript [31].

2.2 Font Terminology

Glyph. A "glyph" in typography refers to the specific shape or representation of a character or symbol in a font [8]. It is essentially the visual manifestation of a character. For example, in a particular font, the letter 'A' has a unique appearance that is the glyph for the character 'A' in that font. For TrueType, their representation is stored as vector graphics in the font file.

Rasterization & Hinting. Rasterization is the step that converts the vector outline of a glyph into actual pixels (i.e., a bitmap). Here, hinting is the process of optimizing the outlines so that they appear clear and legible on screens with various resolutions and pixel densities [3, 9]. It involves adjusting the outlines of glyphs to align with the pixel grid of the screen. Note that hinting is optional, and fonts can be rasterized without hinting, especially on high-resolution displays where the effects of hinting are less noticeable [9]. Further, some systems employ automatic hinting, ignoring the hinting instructions in the font file [40]. In the following, we use the terms "rendering" and "rasterization" interchangeably, as rendering a glyph involves rasterizing it.

2.3 TrueType

TrueType is an outline font standard initially developed by Apple in the late 1980s and later adopted by Microsoft, becoming widely used on both Macintosh and Windows platforms [7]. It is universally supported across operating systems.

Hinting. TrueType employs a set of instructions and algorithms that work at the glyph level, utilizing a custom instruction set to fine-tune how each glyph is rendered on various screens and resolutions [3]. These instructions, often referred to as "hints," are embedded within the font file as bytecode per glyph and dictate

Table 1: Hinting support for selected applications on major operating systems.

	Windows	**Linux**	**macOS**	**Android**
Chromium	✓	✓†	✓‡	✓*
Firefox	✓	✗	✗	✗
Acrobat Reader	✓	N/A	✓	✓
Thunderbird	✓	✗	✗	✗
Outlook	✗	N/A	✓	✗

† with zoom or when applying the CSS scale transformation * on overscroll or when applying the CSS `font-stretch` property ‡ with `-webkit-font-smoothing: none` at font sizes less than `36px`

how the outlines of glyphs should be adjusted to align with the pixel grid of the display. The font rasterizer executes the bytecode with an interpreter.

3 Font Environments

In this section, we analyze the support of TrueType hinting instructions on different operating systems and applications. While the operating system typically does font rendering, specific applications relying heavily on font rendering, such as browsers or PDF viewers, can ship their own rendering engine. As an example, the Adobe Acrobat Reader ships with its own rendering engine that supports hinting [1]. Thus, we analyze both the operating system support and the support in selected applications. Table 1 provides an overview of the results.

3.1 Native Applications

Many native applications rely on the operating system's default text rendering engine. For instance, on Windows, the DirectWrite API typically processes TrueType hints automatically [7]. Our tests on Windows 11 show that hinting is generally active in third-party applications such as Chromium, Firefox, and Outlook, but also in Microsoft applications such as Outlook or Office. On macOS, the Core Text framework handles font rendering [7]. However, macOS ignores hinting by default. Note that an application can still perform hinting explicitly. On Linux, most applications rely on the FreeType2 library [7, 16]. However, the configuration of FreeType and related components depends on the distribution. Our tests on Ubuntu 24.04 show that hinting is used conditionally, depending on, e.g., font size, transformations, and zoom levels. For attacker-controlled content, it is generally possible to force hinting.

3.2 Browser

The browser can leverage system fonts or Web fonts that are supplied via Cascading Style Sheets (CSS). Here, the `@font-face` directive allows supplying a font

that is loaded from a URL. Further, hinting can be controlled via CSS properties explicitly using `-webkit-font-smoothing`, or implicitly using `font-stretch`. Chromium and Firefox generally use the rasterizer provided by the operating system [7]. On Windows, hinting is generally active in both Chromium and Firefox. In the default configuration of Ubuntu 24.04 LTS, Chromium applies font hinting if specific conditions are fulfilled or if the system configuration enforces it. We see font hinting in Chromium when using a `transform:scale` CSS effect on the font to stretch it, as well as when setting the CSS `zoom` factor to a value larger than '1' and smaller or equal to '2'. According to our testing, the Chromium browser performs hinting on macOS only when applying the CSS property `-webkit-font-smoothing: none` at font sizes less than `36px`. On Android, Chrome also supports hinting, but it is only applied when overscrolling text, i.e., when the user tries to scroll past the beginning or end of a page. However, we can also force this behavior by applying the `font-stretch` property in CSS. When applying a stretching factor, the hinting is always applied.

3.3 PDF

PDF files can utilize either system fonts or embedded fonts. Embedding fonts directly into the PDF ensures consistent presentation across all devices but increases the file size. Some PDF viewers, such as Adobe Acrobat, ship custom rendering engines that perform advanced font processing, including partial or full TrueType hinting [1]. Our tests show that Adobe Acrobat Reader (Version 2024.005.20320 64-bit) uses font hinting on Windows 10 and macOS Sonoma 14.2.1, independent of the system settings. The reader ships with its own rendering engine that supports hinting.

3.4 Email

HTML emails can use custom fonts by specifying them with CSS. However, compatibility varies across email clients, with some clients (e.g., the Gmail web client) not supporting custom fonts. Further, content from the Web, including Web fonts, may not be loaded until explicitly allowed. Note that inline CSS with inlined fonts does not require explicit permission in many cases. Prior research has shown that about half of email clients support custom fonts [39].

4 Threat Model

We assume a victim who uses an attacker-provided font for rendering text. The attacker can control the content of the font, including the hinting instructions. Note that this font does not have to be installed by the victim. Many applications, such as browsers, mail clients, or PDF viewers, can use embedded fonts without requiring explicit permission or user interaction. Thus, it is sufficient that a victim visits a website, opens an email, or PDF provided by the attacker. Further, we assume the hinting instructions are executed by a rendering engine,

either by the operating system or an application. As fonts do not provide direct means to communicate, e.g., via network requests or shared memory, we also rely on a colluding party that relays the results of the attack to the attacker. This colluding party can, e.g., be a simultaneously-opened website. The text rendered with the attacker-provided font can either be attacker-controlled or even secret information that the attacker wants to extract, depending on the scenario.

5 TrueType Hinting for Cache Attacks

In this section, we present a novel approach to constructing cache attacks with TrueType font hinting instructions. We provide an overview in Section 5.1 and analyze the capabilities of TrueType hinting instructions in Section 5.2. In addition, we also investigate the practical limitations imposed by the TrueType standard [3] and the open-source FreeType2 implementation [16]. Note that our approach is not limited to FreeType2 and can be applied to other rasterizers. According to our analysis, FreeType2 is generally the most restrictive regarding the number of memory accesses and executed instructions such that our findings also directly translate to other rasterizers. We describe how to build a custom attack font in Section 5.3 and detail our cache-attack implementation in Section 5.4.

5.1 Overview

TrueType hinting relies on a stack-based interpreter that processes a specialized instruction set, allowing fine-grained control of glyph rendering [3]. Although the primary purpose of this instruction set is to position font outlines with pixel-level precision, its flexible memory model and control-flow capabilities enable attackers to abuse fonts for microarchitectural attacks.

Our approach leverages the TrueType interpreter's storage area and the ability to access it at arbitrary indices. By relying on the `WS` (write storage) and `RS` (read storage) instructions, we can create chains of pointer-chasing sequences stored as hinting instructions. By placing indices referencing subsequent storage locations, the interpreter's repeated access to these entries forms a dynamic traversal through memory. At each step, one storage value determines the following address, i.e., storage location, to read, effectively preventing CPU prefetchers from predicting future accesses and interfering with the eviction [38].

More concretely, we begin by writing a series of indices into consecutive storage slots. Each entry within the storage points to the next, creating a linked sequence of memory references. For instance, one might choose a chain such as $64 \rightarrow 128 \rightarrow 192 \rightarrow 256 \rightarrow 64$. By starting at `storage[64]` and repeatedly using the `RS` (read storage) instruction, the interpreter follows the chain, as `RS` pops the index to read from the stack and pushes the read value onto the stack. Listing 2 in Appendix A shows the corresponding hinting code for this illustrative example. In practice, this loop is much longer. Additionally, we rely on control-flow instructions (conditional jumps and backward branches) to repeat the access pattern multiple times, leading to reliable eviction.

While implementing cache attacks purely through stack operations is theoretically also possible, this is more cumbersome. The storage area already offers a more direct means for crafting predictable memory access patterns.

5.2 Capability Analysis

In this section, we introduce the features of TrueType that enable cache attacks and analyze the limitations imposed by the TrueType standard [3] and the implementation in the FreeType2 library [16, 41].

Storage Size. The most important factor is the available memory that an attacker can use during hinting. We focus on the explicitly provided storage area that is accessed via the `RS` (read storage) and `WS` (write storage) instructions.

Standard The interpreter provides two different types of memory: a stack, and an indexed storage area. The storage area is implemented as a fixed-size array as the maximum index is also announced in the `maxp` table of the font [3]. The limit of both the maximum stack and storage area index are $2^{16} - 1$ [3]. Per TrueType standard, each stack or storage cell must be capable of holding a 32-bit value [3]. As such, the required maximum size of each the stack and storage area, per standard, is thus about 262 kB (262 140 B).

FreeType2 Implementation The FreeType2 library implements the storage area as a `long` array, which increases its maximum size to about 524 kB (524 280 B) on 64-bit systems [41]. After parsing the next instruction from the instruction stream, the interpreter calls the corresponding subroutine for the instruction. As such, the overhead between TrueType instructions is just a small number of lookups, function calls, and range checks. Generally, storage operations are performed by first conducting a bounds check and afterward accessing the storage area array at the corresponding index.

Instruction Limit. The number of instructions that can be executed per glyph is limited. Some interpreters even enforce additional restrictions to detect endless loops and misbehaving bytecode.

Standard The maximum length of the instruction stream for a glyph must be announced in the `maxp` table of the font [3]. Here, the maximum value is $2^{16} - 1$ bytes [3]. Note that this limit is not the number of executed instructions but the length of the bytecode, which can contain loops and subroutine calls. The standard does not specify a maximum number of executed instructions.

FreeType2 Implementation The interpreter adheres to the standard and parses the instruction stream until the value announced in the `maxp` table is reached. During execution, the interpreter keeps track of a few values to detect infinite loops in the bytecode of glyphs. This includes the number of executed instructions, as well as the number of jumps and subroutine calls. By default, the maximum number of executed instructions per glyph is 1 000 000 [41]. In addition, the interpreter limits the number of subroutine calls to 16. Further, heuristics are employed as an upper bound on the number of jump instructions with negative values and the `LOOPCALL` instruction. The bounds are determined dynamically by various factors, such as the number of points in the glyph.

5.3 Custom Font Creation

We choose to build an entirely custom TrueType font with integrated hinting instructions to demonstrate our technique. Note that it is possible to take an existing font and embed our hinting code into its source. Modifying an existing font allows for a more stealthy attack, as the font is less likely to be detected as malicious, by both the user and security software. Existing fonts can be modified using tools such as, e.g., FontForge[1] or `fontTools` [28].

Instead, we generate a font from scratch, including dummy glyphs and hinting instructions, as this approach is more straightforward and flexible. For our attack, the visual appearance of glyphs is irrelevant; only the hinting instructions matter. By building our own font, we ensure full control over the storage and stack configuration required for our pointer-chasing implementation.

We rely on the `fontTools` library for Python [28] to define glyphs, font metadata, and the hinting bytecode. After setting basic font parameters and creating rudimentary glyph outlines, we insert the custom hinting instructions as TrueType assembly instructions. This code sets up the storage indices, writes the required pointers, and then triggers a looped sequence of reads, ensuring that the interpreter continuously accesses memory. Finally, we assemble the entire font and produce a fully functional and standard-compliant `.ttf` file that, when rasterized with hinting enabled, executes the integrated pointer-chasing logic.

5.4 Cache Attack Implementation

With our custom font, we can implement eviction and, thus, eviction-based cache attacks. Since different rendering contexts are isolated into different processes, we cannot leverage shared memory. Consequently, we opt for cache contention and Prime+Probe on the L1 data cache.

Cache Contention. We implement general cache contention using a pointer chase over multiple glyphs. As the amount of memory per glyph is limited, we generate a font where each glyph accesses a different part of the storage area to work around the limitations imposed by FreeType2. This additionally circumvents the limitations of instructions per glyph and the heuristics to detect endless loops. Our custom font contains 52 glyphs, which feature hinting instructions. They correspond to the ASCII letters from ‘A’ to ‘Z’ and from ‘a’ to ‘z’. In total, the font features 8 distinct pointer-chasing sequences, each with 4000 indices. Each sequence is stored in a separate glyph, and the sequences are chosen to access different parts of the storage area. This results in a total of 32 000 indices, which corresponds to 256 kB of memory on 64-bit systems.

L1 Prime+Probe. For L1 Prime+Probe, we construct an eviction set–essentially a set of addresses that all map to the same cache set in the L1 cache. Because many modern CPUs use a virtually indexed L1 cache, we only need to determine virtual addresses that map to the target cache set and do not require

[1] https://fontforge.org/

knowledge of physical addresses. We accomplish this by carefully choosing offsets and strides so that all the resulting storage indices correspond to memory locations that collide within a single set.

Our analysis shows that the storage area is page aligned, i.e., starts at a virtual address that is a multiple of 4096. Moreover, as every storage area entry has 64-bit, we can directly convert a page offset to a storage-area index by dividing it by 8 [41]. As the L1 cache set is defined by bits 6 to 11 of the virtual address, we choose the base index as $\frac{64}{8} \times set$ to ensure that it maps to the set number *set* (given that L1 cache lines are typically 64 bytes). By incrementing the base address in steps of 512 ($= \frac{4096}{8}$), we create additional indices to the storage area where the underlying address maps to the same L1 cache set. For reliable eviction, we choose 64 addresses for the eviction set, ensuring that there are definitely more addresses than cache ways. All calculated indices are combined into the pointer-chase loop.

The hinting code uses a loop to repeat the pointer chase 500 times, ensuring that the memory accesses consistently evict the target cache set. With each iteration, the interpreter follows the same chain of 64 storage accesses, creating a reliable eviction set for the L1 Prime+Probe attack. For this L1 Prime+Probe, one glyph is sufficient for building an eviction set that evicts one cache set. Thus, with a font containing 64 glyphs, we can have one glyph for every cache set.

6 Website Fingerprinting Attack in the Browser

In this section, we present a novel browser-based cache contention attack for website fingerprinting [29, 34]. Website fingerprinting is a side-channel attack that allows an attacker to infer the websites a user visits without directly observing the network traffic [29, 34]. The attack leverages unique patterns in the timing, magnitude, and number of network requests, which are influenced by the structure and content of the website. Prior work has established that website fingerprinting can be performed by monitoring the cache activity of the system, as the cache activity is influenced by the browser's network requests [34].

Our attack implementation consists of two phases: a data collection phase in the browser, and a classification phase that leverages ML-based techniques to identify the visited website. Our data collection phase uses the custom font introduced in Section 5.4 to induce cache contention on the victim's system. The cache contention is measured by timing the rendering of glyphs in the custom font. The timing is influenced by the cache activity, which in turn is influenced by the network requests made by the browser. The classification phase is implemented in Python and uses a time series forest classifier [5, 35] to recover the visited website from the collected data.

6.1 Data Collection

The data collection of our attack is implemented in a single HTML document. A script acts as the colluding party that communicates the attack's results to

```
for (let size of sizes) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.font = `${size}px CustomFontFamily`;
  let start = performance.now();
  ctx.fillText(string, 10, canvas.height / 2);
  let diff = performance.now() - start;
  measurements.push(diff);
}
```

Listing 1: The data collection phase of the Website Fingerprinting attack that uses the custom font to induce cache contention. JavaScript is acting as the colluding party for time measurements.

the attacker. We leverage the Canvas API and its 2D rendering context, which allows measuring the time it takes to render a glyph synchronously. We first load our custom font that contains the TrueType bytecode for our attack. We prepare an HTML `canvas` element, which is manipulated during the following phases. Finally, we wait 10 seconds to ensure our font is loaded before we start the actual data collection.

In the following, we render a string consisting of 8 characters, each corresponding to one of the 8 pointer-chasing sequences in our custom font, repeatedly. In each iteration, we clear the canvas element, set the font to our custom font, and the font size to the current iteration number. This ensures that the hinting instructions are executed again, as the rendering engine has to rasterize the glyphs at a new size, preventing the renderer from using a cached version of the glyph. We render the string on the canvas using the `fillText` method and measure the time it takes to render it using `performance.now()`. The implementation is shown in Listing 1. This is repeated for 120 iterations, so the exploitation phase takes about 5 seconds.

In our setup, we trigger the opening of an additional tab in the browser using `window.open()` in the sixth iteration. This tab contains the target website that we want to fingerprint. It emulates a user visiting a website while the attack runs in an attacker-controlled tab.

6.2 Classification

The data collected in the data collection phase is a time series of 59 measurements. Each measurement corresponds to the time it took to render the string of 8 characters, which are designed to induce cache contention. We use a time series forest classifier to classify the visited websites based on the collected data. The time series forest classifier is an adaptation of the random forest classifier designed explicitly for time series data [5]. For this, we use the `sktime` library, which provides an easy-to-use implementation [35].

6.3 Evaluation

We evaluate our attack on a system running Ubuntu 24.04 LTS with an Intel Core i7-1260P CPU that is connected to a fast and reliable 1 GbE network. As a dataset, we use the Alexa top 100 websites collected on Jan 31, 2022. We opted for this dataset, as it has been used in prior work [42] and also does not contain domains that do not host websites. We restrict ourselves to a closed-world scenario, where the attacker knows the set of possible websites, as this is a common assumption in website fingerprinting attacks [34], and our attack merely serves as a proof-of-concept. We use the Google Chrome browser (Version 120) for our evaluation, as it is the most widely used browser [37]. For the data collection, we instrument the browser in headful mode using the Playwright library. The automated browser visits the attacker-controlled document, which triggers the loading of a target website in a new tab. The target website is chosen from the dataset. This is repeated for all websites in the dataset over 30 iterations. Ultimately, we collect 30 samples for 100 sites, resulting in 3000 samples in total.

Classification Results. The classifier is trained on 75 % of the samples and tested on the remaining 25 %. We employ a hyperparameter search using a grid search with 5-fold cross-validation that is repeated 5 times. This effectively results in 25 training and testing runs, each with a different split of the training data, for the hyperparameter search. After training the classifier, we evaluate the classification performance of the optimal classifier using the accuracy, macro-averaged F1 score, and the top-5 accuracy. The top-5 accuracy is the percentage of samples where the correct website is among the top-5 predictions of the classifier and was used in prior work [33]. The performance is evaluated on the held-out test set of 25 % of the samples.

This evaluation is performed for the top 100 and the top 15 websites in the dataset. On the top 15 websites, the attack achieves an accuracy of approximately 81 %, a macro-averaged F1 score of 81 %, and a top-5 accuracy of 98 % (cf. Table 2). On the entire dataset, we find that our attack achieves an accuracy of 71 %, a macro-averaged F1 score of 69 %, and a top-5 accuracy of 92 % (cf. Table 2). We compare our trained classifier to a dummy classifier that predicts uniformly at random and thus presents a baseline for the classification performance. Since the dataset is balanced, the dummy classifier achieves an accuracy of about 5 % and 1 %, respectively. As our classifier significantly outperforms the dummy classifier, our attack effectively classifies the visited websites.

For the top 15 websites, we also provide a confusion matrix in Figure 1. The diagonal represents the correctly classified websites, while the off-diagonal elements represent misclassifications. The classifier performs well for most sites, with only a few misclassifications. Interestingly, the classifier struggles primarily with websites for Asian countries, such as `taobao.com` and `tmall.com`. This is likely due to our evaluation setup, which uses a system in Europe, and the data collection time window, which is set to 5 seconds.

Figure 3 in the appendix shows the confusion matrix for the top 100 websites. The observation regarding the misclassification of websites for Asian countries

Table 2: The classification results of the Website Fingerprinting attack on the top 15 and top 100 websites of the Alexa dataset.

	Accuracy	macro-F_1	Top-5 Acc.	No. of sites
`DummyClassifier`	0.053	0.007	0.177	15
`TimeSeriesForestClassifier`	0.812	0.813	0.984	
`DummyClassifier`	0.007	0.000	0.040	100
`TimeSeriesForestClassifier`	0.705	0.688	0.917	

also holds for the top 100 websites. In addition, the figure shows that the classifier occasionally struggles with differentiating between sites that offer regionalized domains, such as `google.com` and `google.fr`. This is no surprise, as the content and server infrastructure of these sites are the same, and the only difference is the top-level domain. For example, `google.fr` still serves content in English when accessed with a browser that sends an English language header.

7 Emitting Unique Signals

In this section, we show that our fonts can be useful as sending ends of a covert channel in scenarios where no traditional code execution is possible. This includes PDFs or emails. We combine two properties of hinting: our building blocks to evict a specific cache set (Section 5.4) and that hinting code is only executed when the text is rendered, i.e., visible. Consequently, we can create a specific cache pattern as soon as a text is rendered. Finally, we discuss two use cases: a PDF reading tracker and an email read notification.

7.1 Setup

The sending end of our covert channel is our custom-crafted font that targets an attacker-chosen cache set. We ensure that text is rendered with this font, triggering the eviction as soon as the text is displayed. As the receiving end of our covert channel, we use a program that measures the activity in a target cache set. This can be implemented in native code or as a website, e.g., in JavaScript. The only requirement is that the sending and receiving end of the covert channel run on the same physical CPU core, i.e., on hyperthreads.

We verify that our hinting code indeed evicts the targeted cache set by measuring the activity of all 64 cache sets in the L1 cache when rasterized by the FreeType2 library (Version 2.13.3) on a system running Ubuntu 24.04 LTS with an Intel Core i7-1260P CPU. The receiver is implemented as a native program that repeatedly accesses data in the L1 cache and measures the access times. Figure 2 shows the activity of all 64 cache sets in the L1 cache while rendering text that evicts cache set '21' or cache set '50'. As can be seen in the figure, the

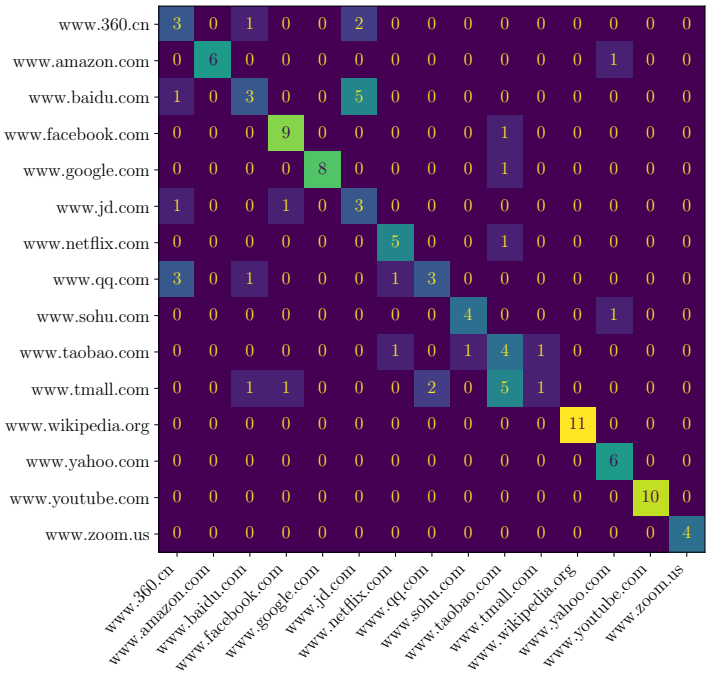

Fig. 1: Confusion matrix for the top 15 websites.

targeted cache set is the one with the highest activity. We measure the activity by averaging the average access time to 300 addresses falling into each set. The higher the activity, the higher the probability that our accessed address was already evicted, leading to a slow cache miss.

7.2 Use Case: PDF Reading Tracker

An interesting use case of this technique is embedding multiple versions of the custom-crafted font–each targeting a distinct L1 cache set or cache-set combination–across different pages of a PDF. When a user opens the PDF in a viewer that supports font hinting, such as Adobe Acrobat Reader or Chrome, rendering the text automatically triggers the eviction patterns. By assigning a unique cache set (or combination thereof) to each page, we can observe spikes in the receiving end's measurements that correspond exactly to the page a user is currently viewing. As long as the attacker's measuring code, which is implemented, e.g., in a website that remains open in the background, is running on the same physical CPU core, it can continuously probe cache sets to detect which page in the PDF is currently displayed. This approach allows for real-time tracking of a user's reading progress. In practice, attackers could use this to stealthily monitor when

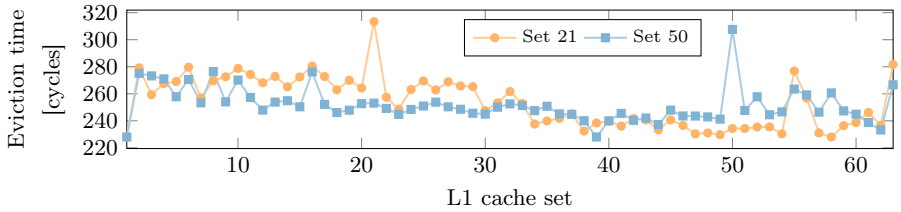

Fig. 2: L1 cache-set activity while rendering a glyph containing an eviction set for a specific cache set. The spike shows the targeted cache set.

sensitive sections of a PDF document are accessed, or the other way round, important parts of the PDF are skipped.

7.3 Use Case: Email Read Notification

A related scenario can be constructed with HTML emails viewed in a mail client supporting font hinting, such as Microsoft Outlook. Just as with PDF files, a font that triggers evictions on a specific cache set can be embedded in the email. This is possible via remote content but also directly via data URLs [39]. When the user opens the email, the hinting instructions activate and evict the designated cache lines. Meanwhile, an attacker-controlled website running in the browser (again on the same physical CPU core) can detect that eviction event. While the website has no control over the core scheduling, it is possible to spawn multiple workers until co-location is achieved [6]. This setup enables a stealthy "read receipt" mechanism, notifying the attacker the moment the user views the email. Additionally, this technique can also be used for benign use cases. For instance, enterprise or security-focused organizations might embed such covert signaling to confirm that a user indeed opened the message on the current device. This technique demonstrates how font-based evictions function as a powerful channel for relaying state changes or user actions, all without requiring explicit code execution in traditional programming languages like JavaScript.

8 Mitigations

In this section, we discuss potential mitigations for our presented attack. We discuss mitigations on the application and the system level.

Application Level. The most straightforward and most restrictive approach to mitigating font-based cache attacks is to prevent the loading of custom fonts. This is, for example, done by email clients such as the webmail interface of Google Gmail. Especially in security-sensitive contexts, such as encrypted emails, this may be a sensible approach. Another restrictive approach is to disable using the bytecode interpreter for untrusted fonts, e.g., for fonts that are not system fonts. This could either be implemented by the application calling the text rendering library or in the text rendering library. Simply removing or reducing the storage

area is not sufficient. An attack can work around a reduced storage area by distributing the attack over multiple glyphs. If there is no storage area, an attacker can still craft an attack via the code or the stack.

System Level. The most restrictive but also most effective mitigation is to disable hinting at the system level. This is, e.g., already the case on macOS, where hinting is not enabled by default for most applications. However, this assumes that applications rely on the operating system for font rendering and do not come with their rendering engine, as is, e.g., the case with Adobe Acrobat Reader on macOS. A mitigation specific to cache attacks would be to randomize the allocation of the memory used by the bytecode interpreter and ensure that the memory is not contiguous. Disruptive memory accesses [10, 31] might also be a simple but effective strategy to make attacks more difficult.

9 Discussion

In this section, we discuss the applicability of our attack to other font formats, theoretical improvements in the data collection, and related work.

9.1 Other Font Formats

Hinting is not exclusive to TrueType fonts. In OpenType, hinting can be provided via embedded TrueType bytecode (for OpenType fonts with a `.ttf` extension) or through Compact Font Format (CFF) instructions (for OpenType fonts with a `.otf` extension) [16, 21]. CFF hinting is more declarative and less procedural than TrueType hinting, which makes it less suitable for our approach.

Other web-oriented formats, such as WOFF (Web Open Font Format) or WOFF2, typically contain a TrueType or OpenType/CFF core. When decompressed by the browser or a rendering engine, the same rasterization pipelines (including TrueType or CFF hinting) can be applied [17].

9.2 Scriptless Data Collection

Similar to the technique described for the CSS-based cache-contention attack [33], we could hypothetically remove the need for a timing solution in JavaScript by leveraging the in-order rendering of HTML documents. Here, timing information is collected by an attacker-controlled DNS server. The rendering time of the HTML is calculated as the difference between the arrival times of DNS requests at the DNS server. These DNS requests are induced by having one image tag at the top and one image tag at the bottom of the document, which is rendered top-down. The approach does, however, not work with hinting, as it is performed asynchronously. The initial layout of the DOM simply uses the *advance width* of each glyph, such that the bottom HTML element in the HTML document is reached before hinting is completed. Note that this is not a limitation of the technique itself but rather a limitation imposed by the current implementation of the rendering pipeline.

9.3 Related Work

Website Fingerprinting. Prior work has achieved 70 % to 90 % accuracy in classifying websites based on cache activity in a closed-world scenario of 100 sites [29, 34]. Notably, existing attacks have been trained on a greater number of samples, which may lead to higher accuracy, and used JavaScript to induce cache contention. Furthermore, our evaluation setup has not been optimized for performance, and we have not conducted extensive hyperparameter tuning, as our work is primarily a proof-of-concept.

Scriptless Cache Attacks. Shusterman et al. [33] presented a scriptless cache contention attack that leverages CSS to induce cache contention (cf. Section 9.2). While their attack does not require JavaScript, the primitive is limited to contention and does not provide fine-grained control over cache patterns. Their attack achieves an accuracy of 50 % on the top 100 websites while the evaluation uses a larger dataset and a more complex classifier.

Font-based Attacks. While font-based attacks are not a new concept, they have primarily focused on exploiting vulnerabilities in font parsers or rasterizers. For example, CVE-2011-3402 describes a vulnerability in the Windows TrueType font parser that could be exploited to execute arbitrary code [4]. As a response, font parsers have received increased scrutiny, with fuzzing tools being developed to identify vulnerabilities [11, 14]. Further, unprotected web fonts have been shown as a vector for misrepresenting a website's content [24].

10 Conclusion

In conclusion, our work demonstrates that microarchitectural attacks can be mounted through hinting instructions embedded in TrueType fonts that are executed during text rendering. We presented a novel cache contention attack that leverages a custom font to induce cache contention for Website Fingerprinting in the browser. Our attack achieves accuracies of 81 % and 71 % on the top 15 and top 100 websites, respectively, demonstrating the feasibility of our approach. We further showed that our fonts can be used to emit unique signals in scenarios where traditional code execution is not possible, such as PDFs or emails, by performing targeted L1 cache-set evictions. These signals can be used for tracking user behavior or as a read receipt mechanism. Our font-based cache attacks bypass existing script-based defenses and highlight the need for further research into scriptless attacks and corresponding mitigations.

Acknowledgment

We want to thank our anonymous reviewers for their comments and suggestions. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. We thank Ruiyi Zhang, Daniel Weber, and Lukas Gerlach for their valuable feedback and discussions on the experiments. We further thank the Saarbrücken Graduate School of Computer Science for their funding and support.

References

1. Adobe: Adobe PDF Library Overview (2004), https://web.archive.org/web/20150923212758/http://www.datalogics.com/pdf/doc/Version6.1/PDFLOverview.pdf
2. Agarwal, A., O'Connell, S., Kim, J., Yehezkel, S., Genkin, D., Ronen, E., Yarom, Y.: Spook.js: Attacking chrome strict site isolation via speculative execution (2022)
3. Apple: TrueType Reference Manual (2024), https://developer.apple.com/fonts/TrueType-Reference-Manual/, retrieved 2024-04-24
4. Bencsáth, B., Pék, G., Buttyán, L., Félegyházi, M.: Duqu: A stuxnet-like malware found in the wild. CrySyS Lab Technical Report (2011)
5. Deng, H., Runger, G., Tuv, E., Vladimir, M.: A time series forest for classification and feature extraction. Information Sciences **239** (2013)
6. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid Prototyping for Microarchitectural Attacks. In: USENIX Security (2022)
7. Esfahbod, B.: State of Text Rendering 2024 (2024), https://behdad.org/text2024/
8. Fonts, G.: Glossary - Glyph (2025), https://fonts.google.com/knowledge/glossary/glyph
9. Fonts, G.: Glossary - Hinting (2025), https://fonts.google.com/knowledge/glossary/hinting
10. Fuchs, A., Lee, R.B.: Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In: Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15) (2015)
11. Google Project Zero: BrokenType (2025), https://github.com/googleprojectzero/BrokenType
12. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
13. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA (2016)
14. (j00ru) Jurczyk, M.: Reverse engineering and exploiting font rasterizers (2015), https://j00ru.vexillium.org/talks/44con-reverse-engineering-and-exploiting-font-rasterizers/
15. van Kesteren, A.: Safely reviving shared memory (2020), https://hacks.mozilla.org/2020/07/safely-reviving-shared-memory/
16. Lemberg, W.: The FreeType Auto-Hinter (2025), https://freetype.org/index.html
17. Levantovsky, V.: Woff file format 2.0 (2024), https://www.w3.org/TR/2024/REC-WOFF2-20240808/
18. Lin, X., Araujo, F., Taylor, T., Jang, J., Polakis, J.: Fashion faux pas: Implicit stylistic fingerprints for bypassing browsers' anti-fingerprinting defenses. In: IEEE S&P (2023)
19. 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)
20. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. ACM CSUR (2021)
21. Microsoft: OpenType Font Specification (2024), https://learn.microsoft.com/en-us/typography/opentype/spec/, retrieved 2024-04-24
22. Monaco, J.: SoK: Keylogging Side Channels. In: S&P (2018)
23. Mozilla: performance.now resolution (2019), https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

24. Mueller, T., Klotzsche, D., Herrmann, D., Federrath, H.: Dangers and prevalence of unprotected web fonts. In: International Conference on Software, Telecommunications and Computer Networks (SoftCOM) (2019)
25. 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)
26. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
27. Percival, C.: Cache Missing for Fun and Profit. In: BSDCan (2005)
28. Python Package Index (pypi): fonttools (2024), https://pypi.org/project/fonttools/
29. Rimmer, V., Preuveneers, D., Juarez, M., Van Goethem, T., Joosen, W.: Automated Website Fingerprinting through Deep Learning. In: NDSS (2018)
30. Schwarz, M., Canella, C., Giner, L., Gruss, D.: Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. arXiv:1905.05725 (2019)
31. Schwarz, M., Lipp, M., Gruss, D.: JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS (2018)
32. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)
33. 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 Symposium (2021)
34. 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 Symposium (2019)
35. sktime: Api reference - timeseriesforestclassifier (2025), https://www.sktime.net/en/stable/api_reference/auto_generated/sktime.classification.interval_based.TimeSeriesForestClassifier.html
36. Spreitzer, R., Moonsamy, V., Korak, T., Mangard, S.: Systematic classification of side-channel attacks: a case study for mobile devices. IEEE Communications Surveys & Tutorials **20**(1), 465–488 (2017)
37. Stats, S.G.: Desktop Browser Market Share Worldwide (2023), https://gs.statcounter.com/browser-market-share/desktop/worldwide
38. Trampert, L., Rossow, C., Schwarz, M.: Browser-based CPU Fingerprinting. In: ESORICS (2022)
39. Trampert, L., Weber, D., Gerlach, L., Rossow, C., Schwarz, M.: Cascading Spy Sheets: Exploiting the Complexity of Modern CSS for Email and Browser Fingerprinting. In: NDSS (2025)
40. Turner, D.: The FreeType Auto-Hinter (2025), https://freetype.org/autohinting/hinter.html
41. Turner, D., Wilhelm, R., Lemberg, W.: Truetype bytecode interpreter - version 2.13.3 (2025), https://gitlab.freedesktop.org/freetype/freetype/-/blob/VER-2-13-3/src/truetype/ttinterp.c
42. 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)

```
PUSHW[ ] 128
PUSHW[ ] 64
WS[ ]        ; storage[ 64] = 128
PUSHW[ ] 192
PUSHW[ ] 128
WS[ ]        ; storage[128] = 192
PUSHW[ ] 256
PUSHW[ ] 192
WS[ ]        ; storage[192] = 256
PUSHW[ ] 64
PUSHW[ ] 256
WS[ ]        ; storage[256] =  64
PUSHW[ ] 64  ; Start at storage[64]
RS[ ]        ; Reads storage[ 64] (which returns 128)
RS[ ]        ; Reads storage[128] (which returns 192)
RS[ ]        ; Reads storage[192] (which returns 256)
RS[ ]        ; Reads storage[256] (which returns 64, forming a loop)
```

Listing 2: Hinting instructions for the pointer-chase example in Section 5.1.

```
timeseries_forest_params = {
  "n_estimators": [50, 100, 150, 200],
  "inner_series_length": [10, 25, 50, 100],
  "min_interval": [2,3,4],
}
```

Listing 3: The hyperparameters used for the optimization of the classifier.

A Sample Pointer Chase with Hinting Instructions

Listing 2 shows a simple example of a pointer chase implemented in TrueType hinting instructions. The first 12 lines set up the storage area, writing the values 128, 192, 256, and 64 to storage indices 64, 128, 192, and 256, respectively. Next, the chase starts at storage index 64 and then performs four reads, following the pointers. The last read returns to the starting point, forming a loop.

B Hyperparameters

Listing 3 shows the hyperparameters used for the optimization of the time series forest classifier. They are used in a grid search to find the best combination for the classifier. Listing 4 shows the best parameters found for the time series forest classifier for the top 15 and top 100 websites as described in Section 6.

```
# best performing hyperparameters, top 15
{'inner_series_length': 50, 'min_interval': 3, 'n_estimators': 100}
# best performing hyperparameters, top 100
{'inner_series_length': 50, 'min_interval': 3, 'n_estimators': 200}
```

Listing 4: The best hyperparameters found for the time series forest classifier.

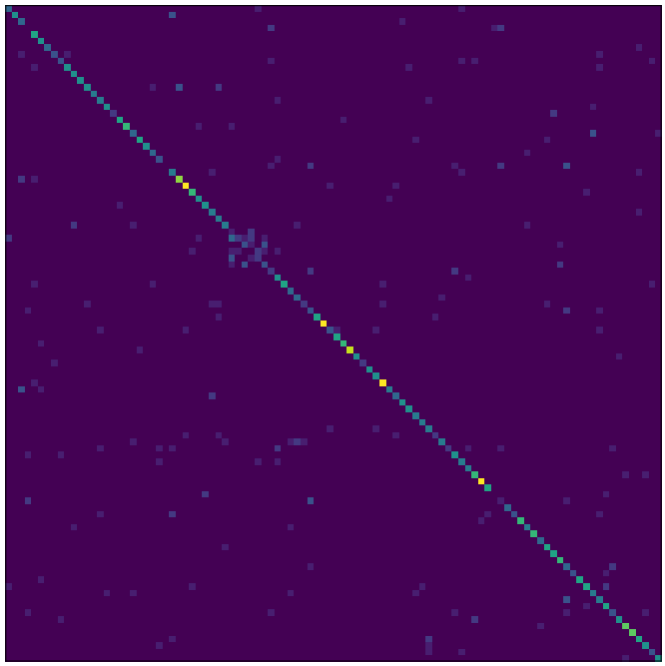

Fig. 3: The confusion matrix for the top 100 websites without labels.

C Confusion Matrix

Figure 1 shows the confusion matrix for the top 100 websites without labels. The diagonal represents the correctly classified websites. Misclassifications correspond to off-diagonal elements. A notable cluster of misclassifications can be observed for regionalized domains by Google, such as `google.com` and `google.fr`.