Fashion Faux Pas: Implicit Stylistic Fingerprints for Bypassing Browsers' Anti-Fingerprinting Defenses

Xu Lin*, Frederico Araujo†, Teryl Taylor†, Jiyong Jang†, Jason Polakis*
*University of Illinois Chicago, †IBM Research
*{xlin48, polakis}@uic.edu, †{frederico.araujo, terylt}@ibm.com, †jjang@us.ibm.com

***Abstract*—Browser fingerprinting remains a topic of particular interest for both the research community and the browser ecosystem, and various anti-fingerprinting countermeasures have been proposed by prior work or deployed by browsers. While preventing fingerprinting presents a challenging task, modern fingerprinting techniques heavily rely on JavaScript APIs, which creates a choke point that can be targeted by countermeasures. In this paper, we explore how browser fingerprints can be generated *without* using *any* JavaScript APIs. To that end we develop StylisticFP, a novel fingerprinting system that relies exclusively on CSS features and *implicitly* infers system characteristics, including advanced fingerprinting attributes like the list of supported fonts, through carefully constructed and arranged HTML elements. We empirically demonstrate our system's effectiveness against privacy-focused browsers (e.g., Safari, Firefox, Brave, Tor) and popular privacy-preserving extensions. We also conduct a pilot study in a research organization and find that our system is comparable to a state-of-the-art JavaScript-based fingerprinting library at distinguishing devices, while outperforming it against browsers with anti-fingerprinting defenses. Our work highlights an additional dimension of the significant challenge posed by browser fingerprinting, and reaffirms the need for more robust detection systems and countermeasures.**

1. Introduction

Online tracking is pervasive across the web ecosystem and has continued to affect users for more than two decades [1]. While many mitigations have been proposed throughout the years [2], and major browser vendors (e.g., Safari, Firefox, and Brave) have become more aggressive in deploying anti-tracking defenses [3]–[5], the underlying economy provides a strong incentive for advertisers and other entities to maintain their privacy-invasive practices. This has resulted in the public discourse around online privacy growing louder, and the U.S. Congress and Senate members introducing drafts and legislation outlining privacy protection measures [6]. Concerns about online tracking have also prompted a series of legislative initiatives that aim to curb and regulate tracking practices (e.g., GDPR [7], CCPA [8]).

Widely deployed defenses by browsers have mostly focused on restricting third-party cookie-based tracking, and the online tracking ecosystem has responded in a reactionary manner by leveraging new techniques for bypassing those restrictions (e.g., [9]). This *arms race* has motivated the development of alternative cookie-less tracking techniques; browser and device fingerprinting techniques have drawn significant attention from the research community, resulting in a plethora of insightful studies and new techniques [10]–[27]. Alarmingly, research has revealed a drastic increase in fingerprinting practices in the wild; while only 0.4% of the top 10K sites leveraged browser fingerprinting in 2013 [28], in 2021 that number climbed to 25% [29].

Popular browsers have recently adopted a series of defensive countermeasures that mitigate browser fingerprinting by blocking certain API calls (e.g., Tor blocking the Canvas API [30]), randomizing the values that certain API calls return to websites (e.g., Brave randomizing what is returned by the Canvas API [31]), or limiting what system resources are made available to websites (e.g., Firefox limiting what system fonts can be used [32]). Researchers have also proposed strategies for detecting and blocking fingerprinting based on the use of specific JavaScript APIs [29], [33]–[36].

In this paper we focus on how existing anti-tracking defenses adopted by privacy-oriented browsers and tools can be bypassed. To that end, we explore *implicit stylistic browser fingerprints* (henceforth referred to as stylistic fingerprints for simplicity), wherein we infer information about the user's environment using CSS features. Our work is motivated by the following observations: (*i*) different HTML elements have different sizes depending on aspects of the environment that they are rendered in, and (*ii*) elements' dimensions can be indirectly inferred using CSS features. Guided by our observations, we develop a novel fingerprinting technique that infers browser and system attributes *without using any* JavaScript APIs (which constitute the cornerstone of modern browser fingerprinting). Our system generates the user's stylistic fingerprint based on environmental attributes ranging from basic properties, like the browser and the operating system, to advanced fingerprints like the list of supported fonts. These attributes are implicitly inferred through the dimensional properties of carefully crafted iframe-based constructions, while also leveraging feature grouping, element placement, and ordering optimizations for achieving practical performance.

To explore our system's robustness against anti-tracking defenses, we provide an in-depth empirical analysis against popular privacy-focused browsers (e.g., Safari, Firefox, Brave, Tor). We also evaluate our system against six popular anti-fingerprinting browser extensions and a state-of-the-art fingerprinting detection system [29]. Our experiments demonstrate our technique's effectiveness, showing that our system is able to collect highly discriminative attributes. Critically, our system infers device characteristics even when users are browsing through the Tor browser, which is notoriously proactive and aggressive in deploying anti-fingerprinting defenses by completely blocking or modifying the returned values of JavaScript APIs that leak information about the user's environment.

We conduct a large pilot study designed to stress test our system and capture the true discriminating power of our techniques, by deploying it for nine weeks within a research institution that is comprised of a highly homogeneous population of user devices. Our experiments demonstrate the effectiveness of our approach, underscoring that our system is comparable to FingerprintJS [37] (the state-of-the-art fingerprinting library which is widely used across the web ecosystem) against non privacy-oriented browsers, while *outperforming* it against browsers that have anti-fingerprinting defenses enabled *by default* (i.e., Safari and Brave). Due to its unique design characteristics and capabilities, in practice, our system can be used in conjunction with JavaScript-based fingerprinting for collecting attributes blocked by existing defenses in popular browsers, or as the sole fingerprinting system in scenarios where JavaScript-based techniques are completely ineffective (e.g., JavaScript execution is blocked).

Our research highlights the inherent privacy threat presented by browser fingerprinting, as trackers can resort to implicit techniques that are capable of inferring system characteristics that are rich sources of entropy, while remaining largely unaffected by available state-of-the-art defenses. Even privacy-preserving browsers that aggressively remove features to enhance privacy are vulnerable to more sophisticated indirect fingerprinting techniques. We hope that our work will further expose the challenges of preventing browser fingerprinting and motivate additional research.

In summary, we make the following contributions:

- We propose stylistic browser fingerprints and develop a novel fingerprinting system that implicitly infers a wide range of browser and system characteristics using CSS and carefully constructed and arranged HTML elements.
- We provide an in-depth empirical evaluation of our system against popular privacy-focused browsers, and explore how our system is effective in scenarios where JavaScript-based fingerprinting techniques falter.
- We conduct a pilot study that demonstrates the capabilities and effectiveness of our CSS-driven fingerprinting system.
- We have disclosed our findings to the browser vendors and will share our system with researchers upon request. A demonstration of our system's capabilities is available [38].

2. System Design and Implementation

We first outline the practical limitations of traditional browser fingerprinting techniques for device recognition, which motivate and guide our research. We then detail our approach for JavaScript-free device fingerprinting via stylistic fingerprints.

2.1. Browser Fingerprinting Challenges

Despite the increasing popularity of browser fingerprinting in device recognition applications, its effectiveness against modern, privacy-oriented browser environments has been hampered by reliability challenges that arise from the inherent distrust that exists between the web client and the content provider. Fundamentally, the fingerprinting features collected in the client environment can be easily altered through client API hooking techniques or completely blocked by clients that disable JavaScript. Essentially, *feature robustness* is a challenge in device fingerprinting because the client features that are typically collected by state-of-the-art fingerprinting methods are susceptible to modification through feature effacing and randomization techniques that are commonly employed by privacy-enhancing defenses.

Another obstacle is *fingerprinting detection*, which is facilitated by scripted fingerprinting approaches that reuse common JavaScript APIs and libraries. Such feature reuse patterns enable browser anti-fingerprinting mechanisms to recognize and disarm fingerprinting behavior [29], [39]. Finally, the *performance overhead* incurred by any newly-proposed fingerprinting technique or system needs to be accounted for, as it can pose an obstacle to real-world deployment.

2.2. Implicit Stylistic Browser Fingerprints

We tackle these challenges by introducing *stylistic fingerprints*, a novel strategy that dispenses the use of JavaScript and provides discriminating fingerprints comparable to current state-of-the-art approaches. Stylistic fingerprints are built from visual attributes generated by web renderers, which depend on a device's configuration. Our technique bypasses existing anti-fingerprinting defenses by relying solely on CSS and HTML elements, without the need for JavaScript API calls that can be blocked or manipulated. These elements are also instrumental in the correct rendering of a webpage, making it difficult to block them without breaking functionality.

However, there are important challenges that arise when creating fingerprints from stylistic web elements. First, we must be able to obtain the fingerprints dynamically without using JavaScript once the browser renders the page. Second, we must select HTML elements that possess discriminatory capabilities, and those elements need to be arranged strategically on the screen to maintain a stable fingerprint, and to ensure that pages' performance does not suffer. Moreover, relying solely on HTML and CSS features mandates an implicit approach to inferring device characteristics, which can lead to an insurmountable number of network requests; this necessitates a precise construction for achieving practical performance. Finally, an effective approach is required to encode usable information from the HTML elements so that the server can actually create the fingerprints.

2.3. Fingerprinting Techniques

We observe that browsers render HTML elements differently in diverse environments, as their dimensions are not solely determined by the browser rendering engine but are also affected by the operating system (OS) and other environmental factors. For example, native HTML elements such as checkboxes and drop-downs are rendered differently across operating systems. Other environmental factors, such as available fonts, user preferences, and browser settings, also have an impact on

Listing 1: Probe the iframe's width in iframe.html.

```
/*Only last matched query sends out request. */
@media (min-width: 300px) {
  #probe {background: url(/iframe-width-300);}}
@media (min-width: 301px) {
  #probe {background: url(/iframe-width-301);}}
...
@media (min-width: 600px) {
  #probe {background: url(/iframe-width-600);}}
```

Listing 2: A simple example.html document showing a stylistic feature using a `<textarea>` element.

```
1 <div class="container">
2   <textarea id="story" rows="5.3" cols="33.99">
3       It was a dark and stormy night...
4   </textarea>
5   <div>
6       <iframe src="iframe.html"></iframe>
7   </div>
8 </div>
```

the rendered dimensions of certain elements. While such rendering differences may be small, dimensional data is sufficiently distinct to differentiate devices. This key observation informs our design: *if we deploy and properly arrange HTML elements in a web page, we can infer device characteristics by observing their dimensions.* Appendix A provides an indicative example.

We aim to obtain multiple elements' dimensions for inferring device information. To collect dimensions without JavaScript, we utilize CSS *media queries*. A CSS media query enables websites to test or retrieve characteristics of the device irrespective of the webpage being rendered on the client. CSS media features' `width` and `height` can be used to test the dimensions of a web page's viewport (the section of the page that is visible in the browser window). However, they cannot directly query HTML elements' dimensions, since media queries are designed to work with devices or media types (e.g., print, screen, speech). `Width`, `height`, and other dimension-based media features all refer to the dimensions of either the viewport or the device's screen in screen-based media—they cannot refer to a specific HTML element. As such, we trick media queries into measuring the dimensions of elements by introducing iframes (inline frames), which are used to embed other web pages into the current page.

To use media queries on HTML elements, we first make an iframe's dimensions adapt to the elements' dimensions. For example, to measure a single HTML element's dimensions, we align the element vertically with an iframe in a container of a fixed height, as shown in Figure 3a (Appendix B). We set the iframe's width and height to 100% so that it takes up all space available in the container. We make the container's width fit the element's width so that the element's width equals the iframe's width. The element's height is equal to the container's height minus the iframe's height.

Next, we place the queries within the iframe. This tricks the queries into believing the iframe is a viewport and causes them to respond with the iframe's dimensions, allowing us to indirectly infer the elements' dimensions. Listing 1 shows the CSS syntax of a media query. The query is analogous to an `if/switch` statement in programming whereby each `media` block represents a different branch in an `if/case` statement. A block is triggered if the condition is met in the `media` block. In our example, if the iframe's `min-width` is 301px, the second block is triggered, and the client browser makes a callback request to the server for the corresponding background image with the crafted url, notifying the server that the iframe's width is 301px. If the dimension does not match any values listed in the query, then no callback request occurs. For each iframe deployed, we make a list of `media` blocks of queries with candidate widths and heights to probe into the iframe's dimensions, and each query requests a unique background image that does not exist on the server, allowing us to obtain the iframe's dimension without any user interaction. In this way, we can obtain and communicate the specific element's dimensions to the fingerprinting service without using JavaScript.

To further illustrate this, in Listing 2 we place a `<textarea>` element (lines 2–4) and an `<iframe>` (line 6) in a `<div>` container. The container's width depends on the `<textarea>` element's width, and the height is 1000px. Suppose we determine that the iframe has a width of 430px and a height of 850px through media queries. Then, we can learn that the `<textarea>` has a width of 430px and a height of 150px. Note that an iframe's dimensions are not always integers, but can also be decimals, as some browsers do not round numbers for media queries (e.g., Firefox). However, it is obviously impractical to generate a media query with all possible decimal numbers in a range of dimensions. Therefore, we use minimum dimension values (`min-width` and `min-height`) instead of the exact values (`width` and `height`). Importantly, conditions from multiple media queries can be satisfied as long as the minimum values are not greater than the actual value, but only the last matched block can be triggered; therefore, `media` blocks must be sorted in ascending order. For example, assume candidate widths range from 70px to 90px, with the iframe's actual width being 80.5px. Then, only the `min-width` of 80px is returned due to sorting.

2.4. Fingerprinting Features

Our framework derives fingerprints from a diverse set of HTML elements and CSS media features to discern different device characteristics. Table 1 details the stylistic fingerprinting attributes and the HTML elements associated with them. Our system has a total of 30 fingerprinting features using 25 iframes and 339 HTML elements. These elements are grouped into *four* categories, according to the types of features they fingerprint. Table 2 summarizes these fingerprinting attributes, which include traditional features typically detected by existing fingerprinting approaches, such as browser vendor and operating system, as well as new features, such as the system language. Our feature selection was guided by prior work as well as an exploratory study wherein we identified new features specific or relevant to styles. We reference the AmIUnique [40] and FingerprintJS [37] frameworks as representative and popular state-of-the-art fingerprinting systems. While we do not aim to comprehensively compare feature set support with prior art since our novelty lies largely in our approach to feature construction

TABLE 1: StylisticFP features and the HTML elements associated with them.

	HTML Elements				HTML Elements		
Feature	**Type**	**Number**	**Entropy**	**Feature**	**Type**	**Number**	**Entropy**
Env-1	acronym, applet, article, aside, pre, form, strike, tt	8	0.42	Env-2	h1, h2, h3, h4, h5, h6, picture, time, del, details, figure, img	12	0.44
Env-3	address	1	0.39	Env-4	canvas	1	0.29
Env-5	audio, video, svg	3	0.36	Env-6	textarea	1	0.44
Env-7	bdi, bdo, bgsound, big, blink, blockqoute, button, input-button, center, rtc, hgroup, keygen, spacer, q, small, p	16	0.46	Env-8	cite, code, data, input-color, content, em, image, progress, meter, portal, ins, dfn, p, marquee, u, wbr, s, mark	18	0.43
Env-9	input-date, input-file, input-month, input-week	4	0.48	Env-10	input-number, input-range, input-time, select, embed	5	0.53
Env-11	input-datetime, input-datetime-local, input-tel, input-radio, input-reset, input-submit, input-image, input-text, input-email, input-search, input-url, input-checkbox	12	0.46	Env-12	span elements of ISO-8859-1 characters, ISO-8859-1 symbols, Greek letters, Math symbols, Miscellaneous HTML entities	5	0.46
Env-13	span elements of non-printable and control characters, ruby, rb	4	0.50	Env-14	main, nav, menu, section, math, fieldset, footer, hr, table	9	0.45
JS-block ext.	noscript	1	0.01	JS-block config.	canvas	1	0.00
Font-pref-1	span elements of test font sizes	20	0.34	Font-pref-2	span elements of system fonts	3	0.44
Font-pref-3	span elements of generic font families	3	0.46	Font-1	span elements of test font families	19	0.52
Font-2	span elements of test font families	19	0.56	Font-3	span elements of test font families	15	0.47
Shadow-font-1	span elements of test shadow font families	19	0.51	Shadow-font-2	span elements of test shadow font families	19	0.56
Shadow-font-3	span elements of test shadow font families	15	0.45	Screen res.	div	1	0.38
Ad-block	ad1	1	0.05	Ad-block ident.	ad2, ad3, ad4, ad5, ad6	5	0.08
Media-1	div	23	0.42	Media-2	div	76	0.58

TABLE 2: Fingerprinting attributes captured by our approach.

Category	**Fingerprint attributes**	**AIU**	**FPJS**
Environment	browser	●	●
	browser major version	●	●
	operating system	●	●
	platform	◐	◐
	operating system language		
	scrollbar settings		
	JS disabled		
Fonts	font preferences		●
	supported fonts	●	●
	supported shadow fonts		
Ad blocker	presence of ad blocker	●	
	ad blocker identification		
Media properties	screen resolution	●	●
	supported media features		◐
	media features' values		◐

AIU: captured by AmIUnique [40] **FPJS**: captured by FingerprintJS [37]
◐: partial feature support ●: full feature support

and the ability to bypass existing anti-fingerprinting defenses, our system incorporates both known and novel attributes.

Environment. The first category contains elements of 101 different types from the HTML elements reference guide [41]. These elements are good candidates for fingerprinting because their sizes vary depending on the environment in which they are rendered. For example, in macOS Monterey 12.4, the width/height of the `<input>` element of type `color` in Chrome v101 is 50px/27px, yet evaluates to 64px/32px and 48px/23px in Firefox v100 and Safari v15, respectively. These values also change with the system and browser versions. We exclude elements that are no longer supported by major browsers, as well as elements that can cause problems (e.g., during our experiments we discovered that the `<object>` element impacts our system's performance in Safari). We detect if JavaScript is disabled by wrapping an HTML element inside the `<noscript>` tag, and we use the `<canvas>` element to determine whether the disabling is due to browser settings. We also include nine elements with special characters in the element's text, due to such characters' rendering being affected by the computing environment. Specifically, we place Greek letters, math symbols, ISO-8859-1 characters and symbols, non-printing characters, and other miscellaneous HTML entities in `<span>` elements, and place East Asian characters with annotations in the `<ruby>` element, which is typically used to demonstrate the pronunciation of East Asian characters. We provide an example of how certain elements allow us to detect the OS language in Appendix C. Elements in this category make use of 14 iframes, ranging from features Env-1 to Js-block config. in Table 1.

Fonts. These are one of the most popular fingerprinting mechanisms due to their discriminating power [42]. We utilize two types of font features: *font preferences* and *supported fonts*. The font preference attributes refer to a browser's font preferences such as font sizes (e.g., minimum font size), generic font families, and system fonts. In total, we embed text in 26 `<span>` elements using various font configurations, and record the element size. The next set of collected attributes provides information about supported browser fonts. In both JavaScript and CSS, websites assign fonts to elements using *font family*. Note that fonts are not the same as font families. A font family is a collection of related fonts. For example, the Arial family is made up of multiple fonts, including Arial Regular, Arial Italic, Arial Bold, Arial Bold Italic, etc. We check for 52 different font families in the browser, derived from the list used by FingerprintJS [37]. Moreover, we define our own set of font families that mirror the existing set of font families using `@font-face`, and divide them into three *shadow* groups. Since we use dimensional data, none of the font families need to be written in the media queries. This category uses nine iframes, ranging from features Font-pref-1 to Shadow-font-3 in Table 1.

Ad blocker presence. We use this set of attributes to detect the presence of an ad blocker and identify it from a list of popular options (e.g., AdBlock, AdGuard). To do so, we use six elements (three `<img>` elements and three `<div>` elements) as ad elements, which bait the ad blocker into removing the element if an ad blocker exists. Two of the elements request a remote resource, thus triggering two requests. While this feature can provide useful information in certain cases, it is not as robust as the other features (e.g., due to ad blockers changing their heuristics, or extensions being disabled when the user is browsing in incognito mode). This category is associated with the features Ad-block and Ad-block ident. in Table 1. HTML elements in this category do not use any dedicated iframes because they share the iframes with elements from other groups.

CSS media properties. We obtain the screen resolution using the CSS media features `device-width` and `device-height`, which do not require the device to be in full-screen mode. Our framework further probes for 23 CSS media features. These include: (*i*) device features, like the number of bits per color component and the number of device pixels used to represent each CSS pixel, (*ii*) browser preferences, such as a light color theme and reduced motion, and (*iii*) browser support of recent CSS media features and their configurations. In this category, we test 23 media features from media queries levels 3 [43] to 5 [44] using 99 media feature expressions. Each expression uses a `<div>` element. Table 6 (Appendix D) summarizes these media features. This category is associated with the features Screen res., Media-1, and Media-2 from Table 1, and uses two iframes.

2.5. Performance Optimizations

To reduce the overhead of stylistic fingerprinting, we implement several arrangement optimization techniques that minimize the number of media query requests while preserving the entropy of the data used to compute the fingerprints, as we detail next.

HTML Element Arrangement. Numerous possible element arrangement strategies exist. In Appendix B, we present different strategies and discuss the information loss that affects certain design choices. Here we present the element arrangement strategy that guided our system's design. We adopt the strategy depicted in Figure 1 and arrange elements into diagonal groups, thereby drastically reducing the number of iframes while preserving fingerprinting entropy. Specifically, we strategically divide specific types of elements into groups and sum the dimensions together, thus avoiding the loss of information. Overall, our system uses the dimensions of 25 groups of HTML elements as fingerprinting attributes.

We place all elements in an 800px by 1000px iframe (hereafter, the *main* iframe) to ensure that the dimensions of the elements remain consistent across different screen resolutions. In the main iframe, we create a `div` container with a grid layout using `display: grid`. We place a group of elements in each column of the container along with an iframe, so that the number of iframes corresponds to the number of groups rather than the number of elements. To obtain multiple elements' widths

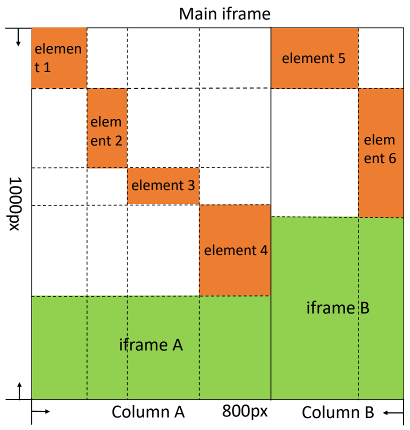


Figure 1: Example HTML element arrangement. The main iframe is divided into two columns. Column A has four elements, while column B has only two. Each element is placed in a specific sub-row and sub-column within the column. Iframe A is in the fifth row, spanning four sub-columns in column A, and iframe B is in the third row, spanning two sub-columns in column B.

and heights using the iframe, we further split the column into a grid layout and arrange the HTML elements along the diagonal of the grid. The number of sub-columns equals the number of elements in this group, and the number of sub-rows equals the number of elements plus one. The first element is in the first sub-row and first sub-column, the second element is in the second sub-row and second sub-column, the third element goes to the third sub-column and sub-row, and so on. The iframe in this column is in the last sub-row and spans all sub-columns.

We obtain the iframe's dimensions using media queries. Within each column, the sum of elements' widths equals the width of the iframe, and the sum of elements' heights equals 1000px minus the iframe's height. In Figure 1, the sums of elements' dimensions in column A and column B define our fingerprinting attributes, which can be obtained with four requests using two iframes . Contrast this to a total of 12 requests for six elements had we employed a single-element-per-container approach. In our implementation, the number of elements in each group varies. We further discuss this in the following section. Note that the main iframe is set to 1000px in height, and nested iframes have a default height of 150px. Therefore, the sum of elements' heights in each column cannot exceed 850px, otherwise it increases the height of the main iframe, making it impossible to use our schema to calculate the sum of the elements' heights.

HTML element grouping. The HTML elements used for fingerprinting are grouped based on the attributes they discriminate. These groups of elements are arranged together in containers aiming to maximize the entropy of that container in discerning a specific environmental feature and meet the height limit of the main iframe.

While we try to group elements that detect a specific environmental attribute (e.g., JS-block, font preferences), the elements in multiple groups can be sensitive to a single feature (e.g., OS language), and the rendering of all groups is

Listing 3: CSS code that probes Media properties' capabilities.

```
/* If the property is supported
    the element has a factor of 2 width. */
@media(prefers-reduced-motion)
    {#element_1{width: 1px; height: 0}}
@media(prefers-contrast)
    {#element_2{width: 2px; height: 0}}
@media(scripting)
    {#element_3{width: 4px; height: 0;}}
@media(environment-blending)
    {#element_4 {width: 8px; height: 0;}}
```

based on some common characteristics (e.g., system, browser). Moreover, the number of elements in each group varies. For example, Env-13 (Table 1) uses four elements to render special characters, while Media-1 feature uses 23 elements for testing media property capabilities.

Font fingerprinting. A naïve algorithm for CSS-based font fingerprinting would check for each font family using the `@font-face` rule directly, resulting in up to 52 CSS media query requests (the number of font families checked by our framework). To reduce this performance overhead, we develop a novel font fingerprinting approach based on elements' dimensions that do not rely on `@font-face` requests. Specifically, we assign a font family and two fallback fonts to a `<span>` element. We use Arial Black and Arial as the fallback fonts, since Arial Black is typically larger than other font families and is available on most systems. When Arial Black is not available, it falls back to Arial, another safe font. If the test font family is available, the element does not use the fallback font and is rendered with a different size. This approach prevents a large number of requests and is not affected by font family name collision, particularly for non-system fonts. Such collisions can occur in scenarios where users have downloaded and installed a custom implementation of a given font family.

Media properties. We deploy two groups of elements to test media properties. In the first group, we probe into the browser's support of 23 media features using 23 `<div>` elements, such as `@media (update)` and `@media (scripting)`. These features are relatively new and some of them may not be supported by a particular browser. All of the elements are grouped with a single iframe in a container, sending two requests to the server, of which one contains the iframe's width and the other contains the iframe's height. We can learn which media features are supported by setting each element size to be a factor of 2 (e.g., 2^0, 2^1,...), as shown in Listing 3. The elements have a size of 0 by default. If a media property is not supported, the size of the corresponding element will remain 0; otherwise, the styles will be applied, and the element's width or height will be some number 2^i, where i represents the position of the element within the group. As a result, the sum of elements' widths or heights will be: $\sum_{i=0}^{n-1} b_i * 2^i$, where $b_i = 0$ if media property is not supported using element i, and $b_i = 1$ otherwise. Given that the result will always be some summation of 2^i values, we will always get a distinct sum for any combination of elements with a non-zero width or height, meaning that we can determine which media properties are supported in the browser, using a single iframe.

While the first group of media features is used to determine what media features are supported by the browser, the second group probes the values of the supported features. There are a total of 76 media feature values of interest, which we again encode in `<div>` elements using a single iframe. The initial size of these `<div>` elements is also 0. However, this time, there are too many values (76) to encode following the same approach as with the first group, so we have to use a different technique to add them to our fingerprint. Values are queried using media feature expressions. If the media feature expression is satisfied, the element's width and height are automatically set to a non-zero value. The new width value of each element varies across expressions, while the new height is always a fixed value. When an expression is satisfied, the iframe's height is decreased by a fixed value while the width is increased by a variable amount. The sum of heights tells the number of satisfied feature expressions, while the sum of widths differentiates the set of satisfying feature expressions. Encoding the data in this way tells us how many expressions are satisfied, and provides some variance in the width values. While the encoding is not as exact as the first group, it does give our fingerprint more entropy.

We deploy four additional requests that do not use dimensions, as supplementary features to our system using `@supports`. To optimize the number of requests, we apply multiple rules to the same element and order these feature queries from general to specific, starting from the most general rule, and appending conditions in the subsequent queries, as shown in Listing 4 (Appendix E). Overall, we use four elements to probe the browser's support for 12 CSS features.

2.6. Fingerprinting Framework

Our fingerprinting framework is deployed as a stand-alone web service with a database backend, and can be seamlessly integrated into web applications. Deployment has no dependencies on the target site, and is agnostic of the underlying web framework and infrastructure. The platform requires only one line of HTML markup, (see Listing 5 in Appendix E) to embed its main iframe object, and all subsequent fingerprinting payloads are sent directly to the backend. Additionally, many techniques can be used to render the iframe invisible to users [45]. For example, the iframe can be positioned offscreen using `position: absolute; left: -9999px;`, it can be rendered to a size of 0, or it can be hidden with the `visibility` property. Device characteristics are inferred based on the dimensional data collected, which reveal information about the device, and are combined into an identifier for uniquely identifying devices.

2.7. Threat Model

We consider a malicious or privacy-invasive service that aims to fingerprint the user's device, allowing it to re-identify and track the user across sessions. We assume that the attacker is able to (*i*) trick the user into visiting the fingerprinting website, or (*ii*) inject a single line of HTML code into a legitimate web page (as shown in Listing 5 in Appendix E) to include the fingerprinting payload in user responses, or (*iii*) leverage a man-in-the-middle proxy service to inject the fingerprinting code in proxied web responses.

3. Bypassing Anti-fingerprinting Defenses

Here we discuss our system's ability to bypass defenses. Our analysis focuses on the most popular browsers and tools that explicitly implement privacy-preserving countermeasures against fingerprinting. Since privacy-focused browsers are actively deploying anti-fingerprinting measures (albeit focused on JS-based techniques) and other tools are also available, we empirically explore whether and how existing defenses affect our techniques.

Experimental setup. For our empirical analysis, which requires testing our system across a wide combination of client environments, we use online services [46], [47] as well various physical devices from our lab. We test multiple versions of operating systems and browsers, and also experiment with different changes to the systems' configurations to assess whether our stylistic fingerprints capture the updated characteristics. For instance, we change the OS language, and install new fonts to verify the collected fingerprints. We use the latest version of browsers and tools at the time of writing, including Firefox v100, Brave (Nightly) 1.39.42, Tor 11.0.10, Safari v15.5, Opera v87.0.4390.36, Ghostery Dawn v2022.4.1, and recently-downloaded extensions. We enable the anti-fingerprinting feature in these browsers if necessary (e.g., Firefox), and use these browsers and privacy tools to visit state-of-the-art fingerprinting systems (e.g., FPJS and AmIUnique) and our StylisticFP system to evaluate the effectiveness against the countermeasures.

Findings. Table 3 summarizes our system's effectiveness. It breaks down the attributes of StylisticFP and indicates whether they are effective against anti-fingerprinting browsers, extensions, and detection systems. Our system is able to differentiate not only the browser engines but also differentiate browsers that use the same engine in certain environments; we can distinguish Edge and Opera from other Chromium browsers running on Windows, Tor and Ghostery from Firefox, and Mobile Safari from desktop Safari. The framework is effective against both desktop and mobile devices. Note that Safari is the only browser available on iOS devices, as other browsers are merely skins on top of Webkit. Consequently, browsers on the same iOS device have identical fingerprints. Our approach also allows us to distinguish various major browser versions based on the observation that certain elements are rendered differently across versions. For instance, in Windows 11, Firefox v100 renders several elements in different sizes compared to v99 (e.g., `<address>` and `<select>`). Also, browsers are gradually adding support for media properties, especially those in the working draft (e.g., Media Queries Level 5), which also allows differentiation. For example, Firefox v100 supports `@media (video-dynamic-range: standard)` and `@media (dynamic-range: standard)`, while v99 does not. Our system generates 11 fingerprints for Firefox v80-101, and ten fingerprints for Chrome v80-101. While certain versions can be uniquely identified, others are grouped into a subset of similar versions. As mentioned, our system also distinguishes Opera and Edge from other Chromium browsers in Windows, due to elements being rendered differently, such as the `<number>` element in Opera and the input field element of type `time` in Edge.

Our Platform attribute provides more details than the corresponding JavaScript API `navigator.platform`. For example, we can distinguish Windows 8 from Windows 10 and Windows 11, while the JavaScript API returns the value `Win32` for all of these systems. The Font Preferences row refers to the font customization setting in browsers, which allows users to configure the font size and default font families for Standard, Serif, Sans-serif, and fixed-width fonts. We also have attributes for identifying if users have disabled Javascript through browser settings or extensions. For example, users can disable JavaScript in the site settings in Chrome and using `about:config` in Firefox. Alternatively, they can use an extension such as NoScript [48] and Disable JavaScript [49].

3.1. Brave

Brave recently added protection against language fingerprinting and font fingerprinting starting with version 1.39 [50]. Our approach can effectively collect both fingerprints.

Anti-language fingerprinting. Brave defends against language fingerprinting by reducing and randomizing the information available in the `navigator.language` and `navigator.languages` APIs, as well as in the `Accept-Language` header. If the fingerprinting protections are set to Strict, Brave will always report "English." More importantly, there is no way to detect the OS language in modern browsers using JavaScript (the legacy Internet Explorer can obtain it using `navigator.systemLanguage`). Our system does not obtain the browser language preferences, but determines OS languages by observing the dimensions of the language-related iframe.

Anti-font fingerprinting. Brave defends against font fingerprinting by randomly removing entries from the browser's font family list during each session, so that the fingerprinter does not get a stable view of the available font families; however, the browser still allows CSS access to local font files. We can thus check if a font is available on the user's device by loading the local font file, allowing us to bypass their defense.

In order to support fingerprinting for browsers that do not block font families, we assign font families to `<span>` elements and divide them into three groups to reduce network traffic. As with other attributes we collect, we use the sum of the elements' widths and heights to establish which font families are present in the browser. For Brave, we also use our shadow font families that mirror the existing set of font families. These shadow font families are defined using original font families' local font files, which are not blocked by Brave. For example, the Arial shadow font family contains: Arial Regular, Arial Black, Arial bold, which we access directly through font files. We use these two sets of font groups to identify whether font family blocking is enabled, and to retrieve the proper font values for our fingerprint. We provide a video demonstration of our system against Brave [51].

3.2. Tor Browser

The Tor browser is built on a stripped-down version of Firefox that is heavily geared towards enhancing privacy by removing features. Tor was the first browser to tackle fingerprinting, and also employs Javascript hooking for spoofing certain fingerprinting APIs. Tor's overarching strategy is to have all Tor users expose the exact same fingerprint,

TABLE 3: Stylistic fingerprinting attributes and their effectiveness against popular countermeasures: ✓denotes that our technique is effective, ✗ denotes that it is ineffective, and ⊕ denotes that it is partially effective.

Feature	Brave	Tor Browser	Firefox	Firefox w/ FP Protection	Safari	Opera	Chrome w/ Anti-FP Extensions	Ghostery Browser	FP-Inspector [29]
Browser	✓	✓	✓	✓	✓	✓	✓	✓	✓
Browser major version	✓	✓	✓	✓	✓	✓	✓	✓	✓
OS	✓	✓	✓	✓	✓	✓	✓	✓	✓
Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓
OS Language	✓	✓	✓	✓	✓	✓	✓	✓	✓
Font Preferences	✓	✓	✓	✓	✓	✓	✓	✓	✓
Scrollbar Settings (OS X)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Available Fonts	✓	⊕	✓	⊕	⊕	✓	✓	✓	✓
Ad blocker Use	✓	✓	✓	✓	✓	✓	✓	✓	✓
Javascript disabled	✓	✓	✓	✓	✓	✓	✓	✓	✓
Screen resolution	✓	✗	✓	✗	✓	✓	✓	✓	✓
Supported media features	✓	✓	✓	✓	✓	✓	✓	✓	✓
Media features' values	✓	⊕	✓	⊕	✓	✓	✓	✓	✓

allowing them to blend into the anonymous crowd. When tested on AmIUnique, Tor spoofs the User Agent and Content Language attributes in the HTTP headers, as well as an additional 25 attributes. Apart from the newly-introduced attribute for detecting the presence of an adblock extension, the remaining 33 attributes listed on AmIUnique are not spoofed by Tor because they have relatively low entropy. Examples include the use of IndexedDB and the visibility of the menu bar.

Media queries. Tor also forces certain media queries to report identical values. For instance, `prefers-color-scheme` always returns `light`, `color` always returns 8, and the device width and height return generic values, e.g., 800*1000. On the other hand, some queries (e.g., `forced-colors: none`) only compute true in recent browser versions, allowing us to identify certain versions. This makes our approach to fingerprinting media features' values partially effective against Tor. More importantly, Tor does not spoof `min-width` and `min-height` media features; as such, all of the dimensional data we obtain are actual values. As a result, the stylistic features derived from dimensional data are *not* affected by Tor's defenses.

Fonts. To prevent font fingerprinting, Tor has introduced a font allowlisting mechanism which only allows certain system fonts to be used in the browser. The allowlist can be edited in `about: configure`. Traditional JavaScript font fingerprinting relies on including multiple <span> elements with the same text using different font families and baseline fonts as the fallback option, and then comparing their dimensions to that of the baseline fonts. If a given font family is not supported by the system, the element will use the fallback fonts and the dimension of this element will equal the dimension of the baseline element. If the font family is available, the element's size will differ for that specific font. The baseline fonts used in fingerprinting are typically generic font families like monospace, sans-serif, and serif. However, if the fingerprinting script sets the fallback font to monospace, the code will always detect the specific font as available because Tor never falls back to monospace. Specifically, if a font is unavailable, Tor skips the fallback font monospace and falls back to a different font. The element's size thus always differs between the monospace base font and the specific font with a monospace fallback font. As a result, traditional font fingerprinting strategies will detect that all font families are available in the browser. In fact, both AmIUnique and FingerpintJS use monospace and are thus ineffective against Tor.

Additionally, Tor bans the use of `@font-face` local files, regardless of being allowlisted or blocklisted. Even if we load a local font file and refer it to an allowlisted font family (e.g., Arial), this font family will be inaccessible. As a result, we cannot access the non-allowlisted fonts using `@font-face` as we did with Brave; however, our approach can accurately detect the available fonts on the allow list, by using the three font family groups and the shadow groups as described in our font discussion on the Brave browser. If the font family is allowlisted and available, the three font family groups will have access to it, while the shadow groups will not because they utilize `@font-face`. The shadow groups will load preselected fallback fonts instead. Consequently, the iframes associated with the family groups and the shadow groups will be of different sizes. While our system cannot infer all the fonts present in the user's system, it accurately identifies support or the lack thereof for the set of fonts in the allowlist.

3.3. Firefox

The default version of Firefox does not prevent our system from collecting any of the fingerprinting attributes. However, Firefox has also incorporated Fingerprinting Protection [52], an experimental feature that is disabled by default. Firefox has opted to not include this option in the settings menu and, instead, users can access this option by typing `about:config` in the address bar. This feature includes a series of protections, some of which affect our system while others are ineffective.

Specifically, our approach is still able to bypass spoofing attempts in which the browser reports a specific, common version number, and operating system. Our approach still detects the actual operating system, browser, and major browser versions. Additionally, while the language is disguised, our system correctly detects it. Finally, while `Window.devicePixelRatio` always returns a value of 1, our approach infers the actual value through `@media(-webkit-device-pixel-ratio)`.

On the other hand, Firefox also uses a font-allowlisting mechanism in which only certain system fonts are made accessible to websites. This defense is more robust than Brave's because it blocks fonts at the local font file level. Any font families that use font files that are not allowlisted are blocked. Interestingly, Firefox's font protection is not identical to Tor's. Even though they both block local font files they use a different allowlist, and Tor bans the use of all local font files while Firefox only bans the use of local files that are not on the allowlist. Additionally, the CSS screen resolution is spoofed, and certain media queries report misleading information (e.g., the value of `@media(color)` is reset to 8).

3.4. Other Browsers

Safari. Safari only renders the default system fonts unless it is a web font included by any website (since these do not indicate if a local font is available). Safari also blocks the use of local font files that are not from a system font family. Our system is partially effective and detects fonts from the allowlist.

Ghostery. Ghostery is built on top of Firefox and provides additional privacy features. Our system is also effective against this browser, with the majority of fingerprinting values being identical to Firefox. Moreover, our system distinguishes Ghostery from Firefox due to the support of additional CSS feature values, like `grid-template-columns:masonry`.

3.5. Extensions and Tools

Anti-fingerprinting tools. We use Chrome to test six anti-fingerprinting extensions that target common fingerprinting attributes. Table 7 (Appendix F) lists the extensions that we study along with their number of users as provided by the Chrome web store. None of the tested tools affects our fingerprinting process. A demonstration of our system's capabilities against spoofing and JS-blocking extensions is available [38].

Ad blocking. We test eight popular ad blocking options, namely the Opera browser (which has integrated ad-blocking functionality) and seven Chrome browser extensions. The goal of this experiment is to explore whether our system can identify the presence of ad blockers but also uniquely identify each tool based on the unique combination of elements it blocks. Table 8 (Appendix G) shows how the ad blockers affect the specially-crafted ad elements included in our system. We analyzed the source code of these popular extensions and the DOM element styles added by Opera browser to find differences in their blocking strategies. Based on that, we have a general element (ad1) that probes the presence of an ad blocker and deploys five other elements that can only be blocked by certain ad blockers. Apart from Adblock Plus and Adblock blocking the same subset of ad elements, all the other ad blockers affect a distinct subset of ad elements and are, thus, uniquely identifiable. Interestingly, during our analysis we found a bypass against Opera's ad-blocking functionality, which we detail in the Appendix G.

FP-inspector. We test a state-of-the-art fingerprinting detection system proposed recently [29]. The paper includes a list of fingerprinting API keywords that are frequently used in fingerprinting scripts. We also check their OpenWPM-extending [53] script instrumentation. We consider fingerprinting attributes that use these fingerprinting APIs to be ineffective. None of the APIs had any effect on our system. This is partially expected due to their classifier having been trained to detect fingerprinting based on JavaScript APIs. We emphasize that we include this experiment purely for the completeness of our empirical evaluation. We consider this proposal an important contribution towards the development of robust anti-fingerprinting defenses.

TABLE 4: Comparison of number of iframes and requests between the initial and optimized design of our system.

Request Source	Initial	Optimized
Main iframe	1	1
CSS files	171	1
Number of sub-iframes	170	25
Requests by iframes	340	50
Requests by @font-face	up to 512	0
Requests for ad blockers	2	2
Requests by other media features	up to 35	4
Total Requests	up to 1,231	83

3.6. Summary

Our empirical analysis demonstrates that StylisticFP is effective at bypassing the protection offered by privacy-oriented browsers, extensions, and detection tools. The majority of our techniques work against all browsers and extensions, and even when they are not completely effective (e.g., supported fonts), they are still better than state-of-the-art systems. Existing fingerprinting countermeasures typically block or manipulate JavaScript fingerprinting APIs' values. As a result, these browsers and extensions impact FingerprintJS (the most popular fingerprinting library) and AmIUnique (a state-of-the-art academic system used in numerous studies, e.g., [11], [18], [54]). On the other hand, our system is mostly unaffected because our approach does not use any JavaScript code. Overall, our empirical analysis highlights the long-term implications of our research. Future countermeasures will require a broader view of how fingerprinting can be achieved and not limit their focus to JavaScript APIs. Crucially, implicit techniques that indirectly infer system properties pose an additional challenge that needs to be taken into account.

4. Experimental Evaluation

To further substantiate our results, this section describes additional experimental aspects of our fingerprinting framework and a pilot study conducted within a research organization.

4.1. Design Optimization

As outlined in §2, our system is driven by a precisely designed construction of HTML elements and CSS features to overcome the impractical overhead of a straightforward CSS-based fingerprinting approach. Table 4 provides a comparison of key behavioral and structural aspects between our

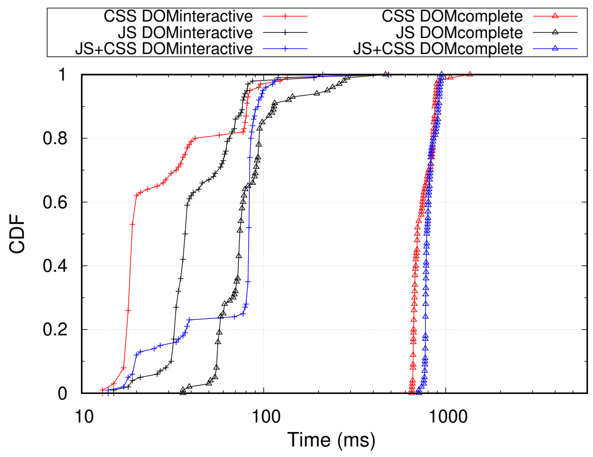


Figure 2: Comparative fingerprinting technique performance.

optimized design and our initial implementation that relied on a straightforward use of the same CSS features. The most important optimization is driven by the choice to leverage dimensional data, avoid `@font-face` requests while focusing on 52 font families for font fingerprinting, and combining multiple media features with logical operators. As shown, our optimized design significantly reduces the resources needed by the system across all categories. Crucially, the implementation can achieve a ∼15x reduction in the number of network requests generated (depending on characteristics of the user's system). To further reduce the size of transferred resources we employ server-side compression, resulting in transferred resources of about 330 KB.

Overhead. To quantify the system's overhead and assess its impact on user experience, we compare our approach to FingerprintJS, and test three scenarios: a standalone deployment of each system as well as a combined deployment of both tools. Each experiment is executed 100 times on a 2019 MacBook Pro i9 running Chrome. To measure the performance overhead, we use Google's Lighthouse [55] to capture the `domInteractive` and `domComplete` timestamps, which mark when the DOM is ready and when the page and all of its subresources are ready, respectively. We ran it in a lab environment to avoid external factors (e.g., network jitter) from affecting the measurements. As shown in Figure 2, the impact on the page's rendering is negligible and the delay for user interaction is less than 100 ms. Moreover, our approach is stable and the entire page's loading time is less than 1 second in 98% of the runs. We note that there is no heavy rendering on our website as the page only renders native HTML elements, resulting in only 83 network requests. Indicatively, Amazon's homepage issues over 300 requests and Facebook's feed starts with about 230 requests. Overall, our design of CSS-based fingerprinting is practical and can also be combined with traditional JS-based techniques to maximize the amount of collected entropy.

4.2. Pilot study

Next, we aim to assess the efficacy of stylistic fingerprinting under challenging conditions in a realistic deployment

TABLE 5: Comparison of uniquely identified devices by our system (**StylisticFP**) and FingerprintJS (**FPJS**) in a pilot study.

		Visits		Unique Fingerprints	
Browser	**Devices**	**Avg**	**Max**	**StylisticFP**	**FPJS**
Chromium	278	4.35	43	168	**180**
Brave	16	3.45	8	**13**	11*
Edge	41	3.83	11	**33**	32
Firefox	379	5.18	278	248	**253**
Safari	152	6.16	210	**72**	63
Total	866			534	**539**

*Visits within the same session, randomized values did not change.

scenario *over time* (so as to also capture the effects of anti-fingerprinting defenses). We conducted a 9-week pilot study in which we deployed the fingerprinting system on three different online portals hosted in a large organization, which are only accessible after authentication. It is important to note that the study's population is comprised of computer scientists and may not provide a representative population in terms of browser selection or configurations. As the pilot study was announced, certain actions may also deviate from normal user behavior and indicate users purposefully modifying their environment to test the system. Nonetheless, as detailed in §2, the true impact of our technique is evident against more privacy-aware users. Moreover, our study captures an especially challenging environment as the device population is heavily skewed towards more specific, homogeneous models that are approved and managed by an institutional IT office.

Metric. First, we focus on the discriminatory power of our novel stylistic fingerprinting system, by comparing our system's ability to uniquely identify devices against the latest version (v3) of FingerprintJS (FPJS), a prevalent state-of-the-art browser fingerprinting library. FPJS deploys various fingerprinting attributes using JavaScript, including both basic (e.g., `colorDepth` and timezone) and advanced features (e.g., Canvas and Fonts) and newly introduced CSS media features (e.g., `forcedColors` and `monochrome`) and font preferences.

Setup. The deployed system sets an HTTP cookie with a random string for distinguishing devices, which provides the necessary ground truth for our analysis. Moreover, since certain defenses rely on randomizing values, we filter out devices that were not observed at least twice, so as to assess each fingerprinting system's effectiveness and stability across visits. We also filtered out 77 devices due to different system setups being used across visits (e.g., with and without an external monitor). Data were collected from June 1, 2022 to August 8, 2022.

Results. Table 5 breaks down our study's results for the 866 devices that remain after filtering, grouped by browser vendor, and shows how many devices were uniquely identified by each system. Of those devices, 541 ran macOS, 295 ran Windows, and 30 were Linux-based. While many users connected over a Chromium-based browser, which is expected, more than half of the devices used an alternative browser. Findings show that our system and FPJS are comparably effective across the entire dataset, uniquely identifying 534 and 539 devices respectively. Due to the study's homogeneous environment, where many workers have the same physical devices, we observe lower detection percentages of both systems compared to prior fingerprinting studies that were conducted in the wild (i.e., in a more heterogeneous ecosystem). Importantly, our technique is particularly effective at uniquely identifying privacy-focused browsers (Brave and Safari), and also correctly identified the three devices that blocked JavaScript and evaded FPJS.

Surprisingly, FPJS was able to uniquely identify five more Firefox devices than our system, which is due to the users not enabling Firefox's advanced FP Protection feature. In other words, while Firefox has the capability to better protect users from JS-based fingerprinting, the subjects in our pilot study had not enabled that option. While that may be a conscious decision for some users, it is very likely that others were not aware of it. This highlights the dilemma that browsers face when it comes to enabling strict privacy-enhancing features by default instead of making them opt-in, due to potential functionality breakage.

We also identify another important detail regarding randomization defenses. Specifically, FPJS is able to identify Brave devices in cases where randomized fingerprint attributes were the same across visits. This happens because the visits occurred within what Brave perceived as the same session, so the randomized values did not change. As a result, while the FPJS fingerprints were the same across visits in these instances, in practice, FPJS would be unable to identify those devices across different browsing sessions (e.g., when the browser is closed between visits).

Collisions. Our system is more stable across visits, as FPJS fails to identify 188 devices (by calculating different fingerprints across visits), while our system fails against 41. At the same time, our system exhibits more fingerprint collisions with 95 device collisions, while FPJS has 55. Collisions occur in cases where multiple devices (e.g., with identical hardware and software configurations) are assigned the same fingerprint value. We hypothesize that because the stylistic fingerprints are more stable, and because the organization devices are relatively homogeneous, this creates more collisions than FPJS. Even so, our system is able to provide useful information for devices even when it cannot uniquely identify them. It is better to always assign a device to a set of a few potential devices (in our experiments sets typically had two devices, the largest had 12) instead of calculating a completely different fingerprint each time. In practice, this can be leveraged by adding more stylistic features for increased entropy, or using other features (e.g., IP addresses and geolocation).

Features. In the cases where our system outperforms FPJS, we find a wide range of differentiating features collected by our system, including stylistic features (e.g., browser font preferences, special characters rendering), the OS language for Chrome users, and the media feature values for Safari users.

Further analysis reveals that our system mainly fails to identify devices due to ad-blocker extensions being toggled on and off. Furthermore, the behavior of ad-blockers varies during visits, as they may block a specific ad element in one visit but not in another. A few users disabled JS in some visits while enabling it in others. Surprisingly, in other cases, users changed the browser display mode, with certain visits exhibiting a 15px difference in height in all iframes.

On the other hand, FPJS mainly fails for the following reasons: the screenFrame and canvas attributes are unstable across visits. This is more problematic in Safari, while the audio attribute is also unstable in Safari (some visits have an abnormal value of -3). FPJS fails to identify Brave devices due to the randomization of various fingerprinting attributes, while blocking JS also results in FPJS's failing to identify devices.

Overall, our pilot study demonstrates that implicit stylistic fingerprints are not only a viable alternative to existing techniques but possess sufficient discriminative power to outperform FPJS against existing defenses. This highlights the inherent double-edged sword of personalization: the flexibility to alter and personalize one's computing environment, and the corresponding supportive functionality that browsers expose to websites, create ample opportunity for diverse fingerprinting techniques. While preventing browser fingerprinting remains a challenging task, we believe that our work will provide a stepping stone for browser vendors and the research community to develop more robust and comprehensive countermeasures.

Entropy. We also quantify the discriminating power of the various fingerprinting features using the normalized Shannon entropy proposed by AmIUnique [11]. Table 1 shows the entropy of our stylistic fingerprinting features. We also calculate the entropy of FPJS fingerprinting attributes in Table 9 (Appendix I) for comparison. The entropy is computed from 1,848 devices that were encountered during our pilot study (including single-visit and returning devices). For our system, the feature with the highest entropy is Media-2 (0.58), which probes into the values of recent media properties. Font and shadow font features also have high entropy values ranging from 0.45 to 0.56. Using the same set of font families, the font attribute in FPJS has a lower entropy of 0.31. The reason for this is that we use the dimensional data rendered by the specific font family rather than looking at the font family name. Dimensional data detects the underlying environment and allows us to distinguish between fonts with the same name. The most important features in the environment category are Env-9, Env-10, and Env-13, with entropy values ranging from 0.48 to 0.53. Env-9 and Env-10 both include different types of `<input>` elements that vary depending on the system language, region, and time format preferences, while Env-13 includes elements that render four different types of special characters. The environmental feature Env-6 contains information about user scrollbar settings with an entropy of 0.44. JS-block features have the lowest entropy because the majority of users do not disable JavaScript for intranet portals. The FPJS attribute with the highest entropy is canvas (0.53), however, it is ineffective against privacy-focused browsers and tools. Overall, we find that within a larger population of devices our fingerprinting system is comprised of high-entropy elements with more discriminating power than FPJS. We consider a large-scale deployment in the wild as part of future work.

4.3. Prior CSS techniques

A few straightforward CSS-based approaches have been previously proposed [56]–[58]. While they collect certain media feature values, screen resolution, and available fonts, they employ simple approaches that suffer from significant limitations. First, these approaches simply use known media features (e.g., any-pointer), resulting in relatively limited data collection. In contrast, we develop a novel practical technique that builds upon a carefully constructed collection of HTML elements and observes how their dimensions differ based on the environment. In more detail, apart from the screen resolution and fonts, all of the media feature values collected by prior CSS approaches are a *subset* of a *single* feature of our system (Media-2 with an entropy of 0.58), and this feature reveals far more discriminative information than existing media features, such as platform, operating system, settings and preferences, etc., highlighting the vast difference in capabilities between our approach and prior work. Second, these approaches flood the network with requests; for instance, [56] generates 1,347 requests while our system only needs 83. To collect media feature values, they require a request for each media feature, so the number of requests equals the number of media features. Conversely, our system probes into 76 values of 23 media features using a single iframe and only two requests. Similarly, to fingerprint available fonts they require a request for each unavailable font, while our system groups multiple fonts and utilizes elements' dimensions so each font group only needs two requests, and the differences in dimensions further detect the environment and eliminate font name collisions. We employ shadow font groups to detect protection against font fingerprinting. All these advantages stem from our deliberate design and novel implicit fingerprinting approach.

We also note that [59] fingerprints CSS features using the `window.matchMedia()` JS API, thus fundamentally differing from our CSS-based approach while also facing the limitations of all JS-based techniques. Moreover, [58] uses strategies (e.g., for detecting the browsers and OS) that are obsolete or blocked by privacy-oriented browsers (e.g., Tor and Firefox).

Crucially, prior approaches cannot bypass browsers' anti-fingerprinting defenses. For example, Tor bans the use of `@font-face` local files, and prior work will incorrectly identify all tested fonts as unavailable. Tor and Firefox force certain media queries to report identical values. Prior work solely relies on their return values; we identify devices using dimensional data and are thus robust against the countermeasures. Brave's anti-language fingerprinting also prevents all prior techniques.

5. Discussion and Future Work

Mitigation. Our technique could be prevented by using two straightforward strategies, both of which would have significant negative side-effects on websites' functionality.

Blocking iframes. One possible mitigation is to completely block iframes, e.g., by using a browser extension like Auto Iframes Remover [60]. However, iframes are extremely common across the web and crucial for a multitude of legitimate use cases, and disabling iframes will break many websites' functionality. We crawled the Tranco top 100k [61] and found that 49.26% of the 83,476 accessible websites use iframes on their landing pages. Indicatively, removing iframes on Google's account login page breaks the login functionality.

Blocking Media queries. Tor sacrifices some functionalities by reporting fake values for a few media features. However, it is infeasible to spoof all media features because they are a key part of responsive web design [62]. Particularly, the `width` and `height` features allow websites to adjust their layout in response to the viewport of a wide variety of devices.

Additional mitigations could include dynamically monitoring requests for server-side resources or adding noise by applying random CSS properties to fingerprinting elements. However, sites can correspondingly disguise requests to bypass detection, and leverage CSS precedence to prevent additional CSS properties from being applied to fingerprinting elements. Alternatively, static analysis could potentially be used to detect our technique by examining chained media queries.

Fingerprinting detection. Preventing our browser fingerprinting technique presents a major challenge due to its inherent reliance on HTML elements and CSS features that have legitimate uses and are crucial for a website's appearance and functionality. Unlike many traditional fingerprinting approaches that capture static meta properties of the environment through programmatic APIs, stylistic fingerprints rely on more dynamic, intrinsic attributes that are generated by the browser and that are parametric on environment characteristics. While blocking or modifying certain features may be feasible, interfering with other features will require a case-by-case strategy. This motivates the use of machine learning classifiers to differentiate fingerprinting from legitimate functionality (e.g., [29]). However, the fact that our approach is based on pure CSS and HTML (and also implicitly infers system characteristics) further complicates machine learning-based detection and mitigation strategies, due to their prevalent use of these features for legitimate non-fingerprinting functionality. Nonetheless, we consider this a promising direction for developing more robust defenses.

Entropy reduction. The elements or media queries used by our system may yield reduced fingerprinting entropy over time. To counteract such a potential degradation, new HTML elements as well as novel W3C and WHATWG feature suggestions can be incorporated into StylistcFP.

Non-tracking use cases. Our study focuses on the privacy threat presented by stylistic fingerprints. Nonetheless, browser fingerprinting can also be used in security applications, such as user account protection [36], [63] and bot detection [64]. For instance, attackers can replay session cookies and block JS fingerprinting, whereas our system can still generate a reliable fingerprint. We consider the exploration of our system's suitability for these scenarios interesting future directions.

Ethics. Prior to our pilot study, we consulted with internal review boards regarding our research methodology and data collection. Our study was exempted from IRB oversight as we do not derive any insights from human subjects' behavior. Though we do not collect sensitive personal information and cannot identify individuals from the collected data, we went through a rigorous formal internal privacy review process which ensured that our empirical methods comply with the institutional and human resources privacy policies. We provide more details in Appendix H.

Disclosure. Our research demonstrates how trackers can effectively bypass the anti-fingerprinting defenses deployed by popular privacy-focused browsers. The techniques have privacy implications for the design of future countermeasures and, thus, necessitate the responsible disclosure of our findings. We have disclosed our findings to the browsers included in our experiments, and provided them with a detailed description of our techniques in order to facilitate their remediation efforts. Chrome responded that our system could be used as a benchmark in their Privacy Sandbox project [65] to combat fingerprinting. Firefox and Tor expressed interest and requested access to our source code and a paper draft, respectively, for further investigation. Brave awarded a bounty for finding the bug in their font fingerprinting protection and recently fixed the bug in version 1.44.x - Nightly. Safari is also investigating this issue. We have opted against publicly sharing our code due to the obvious privacy risk that our techniques pose to users.

6. Related Work

Our work presents a novel browser fingerprinting system that is precisely constructed using only HTML and CSS features, thus overcoming the limitations of JS-based approaches. In this section we discuss pertinent prior research in browser and system fingerprinting, and proposed mitigations.

Browser and device fingerprinting. Since the seminal paper by Eckersley [10], which demonstrated that fingerprints could be used to uniquely identify a user's device using JavaScript APIs, fingerprinting has garnered significant attention by the research community. Mowery and Shacham [15] demonstrated how the Canvas API can be misused for fingerprinting, while Fifield and Egelman [42] explored the discriminatory power of fonts supported by users' systems. Mulazzani et al. [16] demonstrated how websites can infer a user's actual browser despite the presence of modified User Agent strings. Cao et al. [14] explored the possibility of cross-browser tracking through fingerprinting, and proposed a technique that identifies OS and hardware features through a series of rendering tasks. More recently, Laor et al. [66] proposed a novel timing-based technique that targets GPUs and identifies devices based on unique properties of their GPU stacks. In a more holistic exploration, Laperdrix et al. [11] deployed AmIUnique for collecting user fingerprints, and subsequently provided an in-depth examination of the discriminatory power of different fingerprinting attributes across both mobile and desktop platforms. Vastel et al. [18] focused on the longitudinal evolution of devices' fingerprinting attributes and identified a subset of robust features that remain relatively stable for longer periods of time. Akhavani et al. [67] demonstrated how browser versions are uniquely identifiable based on the unique set of JavaScript functionalities they support. In contrast to the studies above, our work introduces a novel, robust fingerprinting technique that uses pure CSS and HTML features in lieu of JavaScript features that are detected, blocked, or impacted by existing anti-fingerprinting defenses.

In a complementary line of research, studies have shown how browser fingerprints can be augmented by identifying installed extensions [23], [24], [68]–[72]. Interestingly, Laperdrix et al. [73] demonstrated how the presence of specific browser extensions could be inferred from the modifications that occur from style sheets they inject into pages.

Fingerprinting measurements. Prior work has also shed light on fingerprinting in the wild. Yen et al. [74] and Nikiforakis et al. [28] discussed the effectiveness of tracking techniques used in existing fingerprinting tools and measured their adoption across the web. Acar et al. [12] presented FPDetective, a framework for detecting fingerprinting, and conducted a large-scale study. Many subsequent studies have explored detection methods and quantified various aspects of browser fingerprinting [13], [17], [75]–[77].

Fingerprinting mitigations. Prior work has also proposed anti-fingerprinting countermeasures that aim to protect users. PriVaricator [35] and FPRandom [78] add randomness to the values returned by certain JavaScript APIs while also focusing on minimizing functionality breakage. FPGuard [79] presents a runtime fingerprinting detection and prevention approach based on predefined metrics. These academic proposals have motivated subsequent defenses deployed by privacy-oriented browsers (e.g., in Brave [31]). Datta et al. [80] provide an experimental comparison across various privacy-enhancing technologies and suggest that Brave and Tor outperform other privacy tools in defending against browser fingerprinting. Importantly, our empirical analysis (§3) shows that our fingerprinting strategy is highly effective against deployed countermeasures. The core characteristic of our approach is that it does not rely on JavaScript, which has been the driving force behind modern browser fingerprinting, and is thus not affected by existing fingerprinting detection and prevention techniques.

Scriptless Attacks. Heiderich et al. [81] discussed XSS payloads that do not rely on JavaScript and demonstrated attacks that exfiltrate sensitive data via the injection of HTML and CSS. While these attacks and our technique both leverage CSS, they are unrelated attacks with different attack vectors. Importantly, our novelty lies in the meticulous design and construction of an attack that relies on the inference of dimensional data, and many underlying features are different across the two attacks (e.g., we do not use CSS-based Animations, CSS content property, scrollbars, while making heavy use of native HTML elements).

7. Conclusions

This paper highlights and empirically demonstrates that the magnitude of the privacy challenge browser vendors face due to the fact that fingerprinting is more formidable than previously perceived. Specifically, we detail how modern fingerprinting attributes can be *implicitly* inferred in a purely JavaScript-less approach. Our findings pose significant complications for potential countermeasures, as they will need to also take into account HTML and CSS features when trying to curtail fingerprinting attempts. When taking into consideration the already strenuous task of differentiating between legitimate and fingerprinting functionality, these implications are further exacerbated. Overall, we hope that our work will motivate and inform new anti-fingerprinting techniques against implicit non-JavaScript-based fingerprinting and will, ultimately, lead to more comprehensive and robust defenses being deployed by browsers.

Acknowledgements: We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Mike Sava for his instrumental support in our pilot study. This work was supported by the National Science Foundation under grants CNS-1934597, CNS-2211574, CNS-2143363, and the U.S. Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the NSF, the Department of Defense, or the U.S. Government.

References

[1] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *Proc. USENIX Security Sym.*, 2016.

[2] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *Proc. IEEE Sym. Security and Privacy*, 2020.

[3] "WebKit - Full Third-Party Cookie Blocking and More," https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.

[4] S. Englehardt and A. Edelstein, "Firefox 85 Cracks Down on Supercookies," https://blog.mozilla.org/security/2021/01/26/supercookie-protections/, 2021.

[5] P. Snyder, "Partitioning Network-State for Privacy," https://brave.com/privacy-updates/14-partitioning-network-state/, 2021.

[6] R. Boucher, "Realclearpolicy - congress is finally listening to consumers on internet privacy," 2020, https://www.realclearpolicy.com/articles/2020/01/15/congress_is_finally_listening_to_consumers_on_internet_privacy_111354.html.

[7] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (gdpr)," *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, vol. 10, no. 3152676, 2017.

[8] "California consumer privacy act (ccpa) website policy," https://oag.ca.gov/privacy/ccpa.

[9] Y. Dimova, G. Acar, L. Olejnik, W. Joosen, and T. Van Goethem, "The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion," in *Proc. Privacy Enhancing Technologies*, 2021.

[10] P. Eckersley, "How unique is your web browser?" in *Proc. Privacy Enhancing Technologies*, 2010.

[11] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *Proc. IEEE Sym. Security and Privacy*, 2016.

[12] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proc. ACM Conf. Computer and Communications Security*, 2013.

[13] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proc. ACM Conf. Computer and Communications Security*, 2016.

[14] Y. Cao, S. Li, and E. Wijmans, "((cross))-browser fingerprinting via os and hardware level features." in *Proc. Sym. Network and Distributed System Security*, 2017.

[15] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," in *Proc. IEEE Work. Web 2.0 Security and Privacy*, 2012.

[16] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Proc. IEEE Work. Web 2.0 Security and Privacy*, 2013.

[17] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *Proc. World Wide Web Conf.*, 2018.

[18] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-stalker: Tracking browser fingerprint evolutions," in *Proc. IEEE Sym. Security and Privacy*, 2018.

[19] I. Agadakos, N. Agadakos, J. Polakis, and M. R. Amer, "Chameleons' oblivion: Complex-valued deep neural networks for protocol-agnostic rf device fingerprinting," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 322–338.

[20] A. Das, G. Acar, N. Borisov, and A. Pradeep, "The web's sixth sense: A study of scripts accessing smartphone sensors," in *Proc. ACM Conf. Computer and Communications Security*, 2018.

[21] V. Mishra, P. Laperdrix, A. Vastel, W. Rudametkin, R. Rouvoy, and M. Lopatka, "Don't count me out: On the relevance of ip address in the tracking ecosystem," in *Proc. World Wide Web Conf.*, 2020.

[22] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Trans. the Web*, vol. 14, no. 2, 2020.

[23] O. Starov and N. Nikiforakis, "Xhound: Quantifying the fingerprintability of browser extensions," in *Proc. IEEE Sym. Security and Privacy*, 2017.

[24] S. Karami, P. Ilia, K. Solomos, and J. Polakis, "Carnus: Exploring the privacy threats of browser extension fingerprinting," in *Proc. Sym. Network and Distributed System Security*, 2020.

[25] K. Solomos, P. Ilia, N. Nikiforakis, and J. Polakis, "Escaping the confines of time: Continuous browser extension fingerprinting through ephemeral modifications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2675–2688.

[26] K. Solomos, P. Ilia, S. Karami, N. Nikiforakis, and J. Polakis, "The dangers of human touch: Fingerprinting browser extensions through user actions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 717–733. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/solomos

[27] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickel, P. Ilia, Y. Shoshitaishvili, A. Doupé, and J. Polakis, "Unleash the simulacrum: Shifting browser realities for robust Extension-Fingerprinting prevention," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 735–752. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/karami

[28] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *Proc. IEEE Sym. Security and Privacy*, 2013.

[29] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *Proc. IEEE Sym. Security and Privacy*, 2021.

[30] "Browser Fingerprinting: An Introduction and the Challenges Ahead," https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead/.

[31] "Brave Fingerprint Randomization," https://brave.com/privacy-updates/3-fingerprint-randomization/.

[32] "Firefox's protection against fingerprinting," https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting.

[33] S. Bird, V. Mishra, S. Englehardt, R. Willoughby, D. Zeber, W. Rudametkin, and M. Lopatka, "Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection," *arXiv preprint arXiv:2003.04463*, 2020.

[34] C. F. Torres, H. Jonker, and S. Mauw, "Fp-block: usable web privacy by controlling browser fingerprinting," in *Proc. European Sym. Research in Computer Security*, 2015.

[35] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proc. World Wide Web Conf.*, 2015.

[36] X. Lin, P. Ilia, S. Solanki, and J. Polakis, "Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1651–1668.

[37] "FingerprintJS," https://github.com/fingerprintjs/fingerprintjs.

[38] "Demonstration of our StylisticFP approach against anti-fingerprinting extensions," https://vimeo.com/737723235/c2b4c00b9f.

[39] P. N. Bahrami, U. Iqbal, and Z. Shafiq, "Fp-radar: Longitudinal measurement and early detection of browser fingerprinting," *arXiv preprint arXiv:2112.01662*, 2021.

[40] "AmIUnique," https://amiunique.org/.

[41] "HTML elements reference," https://developer.mozilla.org/en-US/docs/Web/HTML/Element.

[42] D. Fifield and S. Egelman, "Fingerprinting web users through font metrics," in *Proc. Int. Conf. Financial Cryptography and Data Security*, 2015.

[43] "Media Queries Level 3," https://www.w3.org/TR/mediaqueries-3/, 2022.

[44] "Media Queries Level 5," https://www.w3.org/TR/mediaqueries-5/, 2022.

[45] X. Lin, P. Ilia, and J. Polakis, “Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill,” in *Proc. ACM Conf. Computer and Communications Security*, 2020.

[46] “BrowserStack,” https://www.browserstack.com/.

[47] “CrossBrowserTesting,” https://crossbrowsertesting.com/.

[48] “NoScript,” https://noscript.net/.

[49] “Disable JavaScript,” https://github.com/dpacassi/disable-javascript.

[50] Brave, “Protecting against browser-language fingerprinting,” https://brave.com/privacy-updates/17-language-fingerprinting.

[51] “Demonstration of our StylisticFP approach against Brave,” https://vimeo.com/739534811/c6f294458d.

[52] Firefox, “Firefox’s protection against fingerprinting,” https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting.

[53] S. Englehardt and A. Narayanan, “Online tracking: A 1-million-site measurement and analysis,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1388–1401.

[54] K. Solomos, J. Kristoff, C. Kanich, and J. Polakis, “Tales of favicons and caches: Persistent tracking in modern browsers,” in *Proc. Sym. Network and Distributed System Security*. The Internet Society, 2021.

[55] “WebDev - Measuring the Critical Rendering Path,” https://web.dev/critical-rendering-path-measure-crp/.

[56] “Css fingerprint,” https://csstracking.dev/.

[57] “No-JS fingerprinting,” https://noscriptfingerprint.com/.

[58] N. Takei, T. Saito, K. Takasu, and T. Yamada, “Web browser fingerprinting using only cascading style sheets,” in *Proc. IEEE Int. Conf. Broadband and Wireless Computing, Communication and Applications*, 2015.

[59] “Fingerprinting CSS,” https://privacycheck.sec.lrz.de/active/fp_css/fp_css.html.

[60] “Auto Iframes Remover,” https://chrome.google.com/webstore/detail/auto-iframes-remover/fhenkighldilmobhdgopkhejbaainnfm.

[61] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, “Tranco: A research-oriented top sites ranking hardened against manipulation,” in *Proc. Sym. Network and Distributed System Security*, 2019.

[62] “Beginner’s guide to media queries,” https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Media_queries.

[63] N. Andriamilanto, T. Allard, and G. Le Guelvouit, “FPSelect: Low-Cost Browser Fingerprints for Mitigating Dictionary Attacks against Web Authentication Mechanisms,” in *Proc. Annual Computer Security Applications Conf.*, 2020.

[64] B. Amin Azad, O. Starov, P. Laperdrix, and N. Nikiforakis, “Web runner 2049: Evaluating third-party anti-bot services,” in *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2020.

[65] “Digging into the Privacy Sandbox - Combat Fingerprinting,” https://web.dev/digging-into-the-privacy-sandbox/#combat-fingerprinting.

[66] T. Laor, N. Mehanna, A. Durey, V. Dyadyuk, P. Laperdrix, C. Maurice, Y. Oren, R. Rouvoy, W. Rudametkin, and Y. Yarom, “Drawnapart: A device identification technique based on remote gpu fingerprinting,” in *Proc. Sym. Network and Distributed System Security*, 2022.

[67] S. A. Akhavani, J. Jueckstock, J. Su, A. Kapravelos, E. Kirda, and L. Lu, “Browserprint: An analysis of the impact of browser features on fingerprintability and web privacy,” in *Proc. Int. Conf. Information Security*, 2021.

[68] A. Sjösten, S. Van Acker, and A. Sabelfeld, “Discovering browser extensions via web accessible resources,” in *Proc. ACM Conf. Data and Application Security and Privacy*, 2017.

[69] G. G. Gulyas, D. F. Somé, N. Bielova, and C. Castelluccia, “To extend or not to extend: on the uniqueness of browser extensions and web logins,” in *Proc. ACM Conf. Privacy in the Electronic Society*, 2018.

[70] I. Sanchez-Rola, I. Santos, and D. Balzarotti, “Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies,” in *Proc. USENIX Security Sym.*, 2017.

[71] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis, “Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat,” in *Proc. World Wide Web Conf.*, 2019.

[72] T. Van Goethem and W. Joosen, “One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions,” in *USENIX Work. Offensive Technologies*, 2017.

[73] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, “Fingerprinting in style: Detecting browser extensions via injected style sheets,” in *Proc. USENIX Security Sym.*, 2021.

[74] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, “Host fingerprinting and tracking on the web: Privacy and security implications.” in *Proc. Sym. Network and Distributed System Security*, 2012.

[75] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, “The web never forgets: Persistent tracking mechanisms in the wild,” in *Proc. ACM Conf. Computer and Communications Security*, 2014.

[76] A. Das, N. Borisov, and E. Chou, “Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures.” in *Proc. Privacy Enhancing Technologies*, 2018.

[77] V. Rizzo, S. Traverso, and M. Mellia, “Unveiling web fingerprinting in the wild via code mining and machine learning,” in *Proc. Privacy Enhancing Technologies*, 2021.

[78] P. Laperdrix, B. Baudry, and V. Mishra, “Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques,” in *Int. Sym. Engineering Secure Software and Systems*, 2017.

[79] A. FaizKhademi, M. Zulkernine, and K. Weldemariam, “Fpguard: Detection and prevention of browser fingerprinting,” in *Proc. IFIP Conf. Data and Applications Security and Privacy*, 2015.

[80] A. Datta, J. Lu, and M. C. Tschantz, “Evaluating anti-fingerprinting privacy enhancing technologies,” in *Proc. World Wide Web Conf.*, 2019.

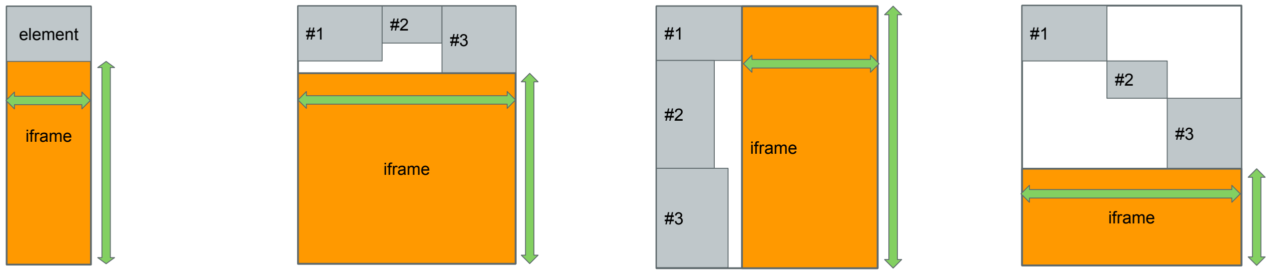
[81] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, “Scriptless attacks: stealing the pie without touching the sill,” in *Proc. ACM Conf. Computer and Communications Security*, 2012.

Appendix A. Element Example

The `<textarea>` element in our system is rendered in Chrome v99 with a width/height of 430px/150px on macOS Monterey 12.2.1, while having a width/height of 432px/162px on Windows 11, and 348px/145px on Ubuntu 18.04. These dimensions may also be different when the browser version changes (e.g., v93). When multiple stylistic elements are employed, the dimensions of certain elements will vary according to the characteristics of the environment, making the device more identifiable.

Appendix B. Element Arrangement Strategy

Figure 3 outlines the different element arrangement strategies, and the information loss avoided by the arrangement employed by our system. A naïve implementation of stylistic fingerprints would deploy an iframe for each HTML element, as illustrated in Figure 3a, resulting in over 100 iframes. Since each iframe needs to send out two requests for dimensional data (width and height), that would incur over 200 query requests in addition to the 100 initial iframe requests. This has a negative effect on page load times. To reduce the number of iframes, we deploy multiple elements with a single iframe. Suboptimal improvements are shown in Figure 3b and Figure 3c,



(a) Dimension Calculation (b) Row Arrangement (c) Column Arrangement (d) Diagonal Arrangement

Figure 3: HTML element arrangement. Figure 3a obtains element dimensions with iframe dimensions. Figure 3b arranges elements in the same row, losing heights of #1 and #2. Figure 3c arranges elements in the same column, losing widths of #2 and #3. Figure 3d arranges the elements diagonally to obtain the sums of the dimensions of all three elements.

which illustrate row and column arrangements, respectively. While these arrangements result in better performance, they suffer from a significant loss of information, namely losing the heights or widths of the arranged elements. Our approach is shown in Figure 3d, whereby we arrange the elements diagonally to obtain the sums of the dimensions of all three elements.

Appendix C. OS Language Detection

Here we provide more details about how our system can detect the OS language. The feature Env-9 in Table 1 is associated with `<input>` elements of the types of `file`, `date`, `month`, and `week`. These elements can be used to detect the operating system language because the OS language defines the browser display language, which in turn determines how these elements are rendered. Note that the browser display language is different from the browser language, which is the language to display website content and is accessible using `navigator.language`. These elements are rendered based on the browser display language rather than the browser language. For instance, the `<input>` element with type="file" displays "choose file" when the OS language is English. When the OS language is Italian and the browser displays in that language, the element shows "Scegli file" instead, and its size differs. The Font-pref-2 element also detects some OS languages because its size depends on the default font and Chromium browsers assign different default font families for some specific languages (e.g., "Hiragino Kaku Gothic ProN" for Japanese and "PingFang SC" for Chinese).

Appendix D. Media Queries

Table 6 summarizes the media features used by our system for features Media-1 and Media-2.

Appendix E. Code Samples

Supplementary to the stylistic fingerprinting features that use dimensional data, we test the browser's support for 12 CSS features directly with requests. To reduce the number of requests, we apply multiple rules to the same element and order these feature queries from general to specific. Listing 4 employs a single request to test three CSS features. If the client browser is Tor, running on a macOS platform, it will skip the first two matched queries and send the `/Firefox-Tor-macOS` request.

Our system can be seamlessly integrated into web applications with one line of HTML markup, as shown in Listing 5. The invisible `<iframe>` element requests the resource from the fingerprinting service and all the subsequent fingerprinting payloads are sent directly to the backend.

TABLE 6: Media features used in our framework.

Media Queries	Media features
Level 3	color, monochrome, orientation
Level 4	any-hover, any-pointer, color-gamut, hover, overflow-block, overflow-inline, pointer, resolution, update
Level 5	dynamic-range, environment-blending, forced-colors, inverted-colors, prefers-color-scheme, prefers-contrast, prefers-reduced-motion, prefers-reduced-transparency, scripting, video-color-gamut, video-dynamic-range

Listing 4: A Basic Example of CSS Features Combination.

```
/* identify Firefox browser */
@supports(-moz-box-align:inherit){
  #probe { background: url(/Firefox); } }
/* distinguish Tor browser from Firefox */
@supports(-moz-box-align:inherit
     ) and (not (hyphenate-character:auto)){
  #probe { background: url(/Firefox-Tor); } } }
/* identify Tor browser running on macOS  */
@supports(-moz-appearance:inherit
     ) and (not (hyphenate-character:auto
     )) and (-moz-osx-font-smoothing:inherit)){
  #probe {
       background: url(/Firefox-Tor-macOS); } }
```

Listing 5: Single HTML markup to enable our system.

```
<iframe src="fp.url" style="visibility:hidden;"/>
```

Appendix F. Extensions

TABLE 7: Fingerprint spoofing and blocking extensions.

Extension	Users
User-Agent Switcher and Manager	200K
Fingerprint Spoofing	50K
Canvas Fingerprint Defender	60K
Font Fingerprint Defender	30K
Trace - Online Tracking Protection	20K
AudioContext Fingerprint Defender	10K

Table 7 details the list of fingerprinting spoofing or blocking extensions that we tested during our experimental analysis.

Appendix G. Ad blocking

Table 8 shows eight ad blockers and their behavior in blocking our crafted ad elements and requests. The differences in blocking behaviors allow our system to discern the tested ad blocker.

Our analysis also finds a bypass against Opera's ad-blocking functionality. Specifically, Opera has a built-in ad blocker that users can easily enable from the right side of the address bar. Opera appends a `<style>` element to the end of the `<head>` element, and it locates ad elements in the `<style>` tag with CSS selectors applying `display:none !important` to remove them from the page. However, ad elements can bypass this protection by taking advantage of precedence in CSS, which defines that inline rules take precedence over those in the `<style>` tag. Thus, we use inline rules to override Opera's rules and render ad elements visible. Although Opera applies the `!important` rule to the `display` property, which overrides all other rules for this specific property on that element, ad elements can also make use of this rule by appending it to `display:block` that renders an element visible. For example, if we add the inline CSS `display:block !important` to an ad element, this rule will have higher priority than Opera's rules in the `<style>` tag, and the ad element will not be blocked and will appear in the page.

Appendix H. Ethics: Pilot Study and Privacy Statement

Prior to our study, we sought advice from various organizational entities to comply with our privacy policies despite getting IRB exemption. This included Human Resources for involving organizational employees, Global privacy review to assess what data is being collected, the security & access control measures in place, and data storage and retention, and Regional privacy review to comply with region-specific regulations (e.g., Europe). Based on their guidance, we provided a privacy statement to inform end users of our data collection (see below).

During our pilot study, we only collected browser fingerprints, including browser fingerprints collected by FingerprintJS and elements' dimensional data collected by our StylisticFP tool. We also set an HTTP cookie with a unique 24-bit random string to distinguish devices for ground truth. We stored all collected data in an encrypted Postgres database, which would only respond to requests from the web service and queries from a set host on our network. Finally, all network traffic was encrypted, with ingress rules for access control.

The privacy disclaimer stated, "The `<redacted>` is collecting anonymized device and browser fingerprinting information for a security research study. The collected data includes web stylistic measurements and device characteristics. The `<redacted>` does not collect sensitive personal information as part of this study. Data will be securely retained until `<redacted>`." The site has additional privacy disclaimers (including data erasure rights) that cannot be shared without revealing institutional sensitive information. We sought and obtained approvals and counsel for this deployment following our institution's strict policies and controls on data acquisition and processing, including region-specific policies.

TABLE 8: Ad blockers' behavior against our system. ✗ denotes that the element or request is blocked.

Ad blocker	ad1	ad2	ad3	ad4	ad5	ad6	req1	req2
AdLock	✗				✗	✗	✗	
AdGuard	✗	✗		✗	✗	✗	✗	
Adblock Plus	✗	✗		✗	✗		✗	
AdBlock	✗	✗		✗	✗		✗	
AdBlocker Ultimate	✗	✗	✗				✗	✗
Ghostery	✗			✗	✗		✗	✗
Opera Browser	✗	✗					✗	
uBlock Origin	✗	✗	✗	✗	✗		✗	✗

Appendix I. Entropy and Effectiveness

In Table 9 we detail the entropy of the various FPJS fingerprinting and header attributes, and whether they are effective against six countermeasures.

TABLE 9: FPJS and header fingerprinting attributes' entropy and effectiveness against popular countermeasures: ✓denotes that the technique is effective, ✗ denotes that it is ineffective, and ⊖ denotes that the feature is not supported by the browser.

Feature	**Entropy**	**Brave**	**Tor**	**Firefox w/FP Protection**	**Chrome w/Anti-FP Extensions**	**FP Inspector** [29]	**JS-Blocked**
fonts	0.31	✗	✗	✗	✗	✗	✗
domBlockers	0.06	✓	✓	✓	✓	✓	✗
fontPreferences	0.34	✓	✓	✓	✗	✗	✗
audio	0.23	✗	✗	✗	✗	✗	✗
screenFrame	0.48	✓	✗	✗	✗	✗	✗
osCpu	0.14	⊖	✓	✓	✗	✗	✗
languages	0.23	✗	✗	✗	✗	✗	✗
colorDepth	0.09	✓	✗	✗	✗	✗	✗
deviceMemory	0.10	✗	⊖	⊖	✗	✗	✗
screenResolution	0.38	✓	✗	✗	✗	✗	✗
hardwareConcurrency	0.21	✗	✗	✗	✗	✗	✗
timezone	0.26	✓	✗	✗	✗	✗	✗
sessionStorage	0.00	✓	✓	✓	✗	✗	✗
localStorage	0.00	✓	✓	✓	✗	✗	✗
indexedDB	0.00	✓	✓	✓	✗	✗	✗
openDatabase	0.05	✓	✓	✓	✗	✗	✗
cpuClass	0.00	⊖	⊖	⊖	✗	✗	✗
platform	0.10	✓	✓	✓	✗	✗	✗
plugins	0.12	✗	✗	✗	✗	✗	✗
canvas	0.53	✗	✗	✗	✗	✗	✗
touchSupport	0.07	✓	✓	✓	✗	✗	✗
vendor	0.13	✓	✗	✗	✗	✗	✗
vendorFlavors	0.09	✓	✗	✗	✗	✗	✗
cookiesEnabled	0.00	✓	✓	✓	✗	✗	✗
colorGamut	0.17	✓	⊖	⊖	✓	✓	✗
invertedColors	0.05	⊖	⊖	⊖	✓	✓	✗
forcedColors	0.05	✓	✓	✓	✓	✓	✗
monochrome	0.00	✓	✓	✓	✓	✓	✗
contrast	0.04	✓	⊖	⊖	✓	✓	✗
reducedMotion	0.02	✓	✓	✓	✓	✓	✗
hdr	0.10	✓	⊖	⊖	✓	✓	✗
math	0.20	✓	✓	✓	✗	✗	✗
Header user agent	0.41	✓	✗	✗	✗	✓	✓
Header accept language	0.35	✗	✗	✓	✗	✓	✓