Artifact Evaluated NDSS
Available
Functional
Reproduced

Cascading Spy Sheets: Exploiting the Complexity of Modern CSS for Email and Browser Fingerprinting

Leon Trampert*, Daniel Weber*, Lukas Gerlach*, Christian Rossow*, Michael Schwarz*
*CISPA Helmholtz Center for Information Security
Email: {leon.trampert, daniel.weber, lukas.gerlach, rossow, michael.schwarz}@cispa.de

***Abstract*—In an attempt to combat user tracking, both privacy-aware browsers (e.g., Tor) and email applications usually disable JavaScript. This effectively closes a major angle for user fingerprinting. However, recent findings hint at the potential for privacy leakage through selected Cascading Style Sheets (CSS) features. Nevertheless, the full fingerprinting potential of CSS remains unknown, and it is unclear if attacks apply to more restrictive settings such as email.**

In this paper, we *systematically* investigate the modern dynamic features of CSS and their applicability for script-less fingerprinting, bypassing many state-of-the-art mitigations. We present three innovative techniques based on fuzzing and templating that exploit nuances in CSS container queries, arithmetic functions, and complex selectors. This allows us to infer detailed application, OS, and hardware configurations at high accuracy. For browsers, we can distinguish $97.95\,\%$ of 1176 tested browser-OS combinations. Our methods also apply to email applications—as shown for 8 out of 21 tested web, desktop or mobile email applications. This demonstrates that fingerprinting is possible in the highly restrictive setting of HTML emails and expands the scope of tracking beyond traditional web environments.

In response to these and potential future CSS-based tracking capabilities, we propose two defense mechanisms that eliminate the root causes of privacy leakage. For browsers, we propose to preload conditional resources, which eliminates feature-dependent leakage. For the email setting, we design an email proxy service that retains privacy and email integrity while largely preserving feature compatibility. Our work provides new insights and solutions to the ongoing privacy debate, highlighting the importance of robust defenses against emerging tracking methods.

I. Introduction

User tracking has played a crucial role in targeted advertisements and the overall economy of the web. Cookies, which are small text files that a website can store on a user's device to identify them uniquely, have been widely used for this purpose [1], [2]. However, with increased awareness that tracking infringes privacy, the use of such cookies has been gradually restricted [3], e.g., by blocking third-party cookies [3], [4] and legally requiring opt-in for certain cookies [3]. With the deprecation of third-party cookies in Google Chrome [4], the industry is shifting from tracking individual users to less invasive alternatives that still enable targeted advertising, such as Google's Topics API [5].

Although browsers have made tracking more difficult, the threat still exists. *Browser fingerprinting* [6], a thriving alternative to third-party cookies, can re-identify browsers and is often equivalent to re-identifying a user [7]. This technique has gained traction in commercial products [8] as an alternative to traditional cookies. Libraries, such as FingerprintJS [9], are utilized by numerous websites [8] to track visitors. JavaScript-based approaches can access various client-specific properties. Noteworthy properties include installed fonts [10], [11] and plugins [12]–[14], rendering differences with emojis and geometric primitives [15], [16], GPU capabilities [17], [18], screen resolution, language, CPU model, system microarchitecture, or available device memory [19]–[21]. These properties identify browser instances uniquely and can be used to track users, allowing browser fingerprinting [6].

Unfortunately, even disabling JavaScript, which is arguably the most radical countermeasure, does not eliminate browser fingerprinting. Lin et al. [22] anecdotally demonstrated that the increasing complexity of CSS, the style sheet language used for websites, can be leveraged for browser fingerprinting. They derive *stylistic fingerprints* using CSS media queries to learn the dimensions of an HTML element aligned to an iframe, which indirectly infers characteristics of the client environment. Such CSS-based fingerprinting is widely applicable as—in contrast to JavaScript—browsers or extensions do not provide means to disable CSS. However, as we show, stylistic fingerprints are just the tip of the iceberg of CSS-based fingerprinting. In fact, recent advances of the CSS standard have introduced several—seemingly more limited—features that come close to code execution by providing conditionals and arithmetic operations, exposing undocumented leakage signals.

In this paper, we thus aim to understand the risks of CSS fingerprinting *holistically*, rather than focusing on individual CSS features. We evaluate the identified privacy threats both in the traditional web setting, and also introduce the novel concept of *email client fingerprinting*. We systematically explore modern CSS features and categorize them into three different avenues for script-less fingerprinting. Our first technique uses CSS's

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230238
www.ndss-symposium.org

new container query rule to measure a container's width. By combining font and element rendering with container sizes, this technique can determine if a specific font is installed, as well as identify OS-, browser-, and context-specific styles. The second technique involves using dynamically-calculated size expressions, including trigonometric functions. Depending on the browser version, CPU architecture, and operating system, these functions have slightly different outputs for specific inputs. Finally, we offer a template-based method to identify differences in the CSS-inferable properties of standard HTML elements. The fingerprinting outcome of these techniques can be obtained by selectively loading an external resource that is observed by attackers through CSS selectors, without the need for JavaScript code.

Our techniques rely on specific corner cases of mathematical functions, font measurements, context-specific styles, and functions. To discover these corner cases, we use a combination of fuzzing and templating. We develop a fuzzer to generate potential corner cases for our techniques and compare the results across different systems. If at least one system produces a different output, this corner case can be used to uniquely identify that system and its user. With our combined approaches, we can distinguish 1152 of 1176 (97.95 %) tested browser-OS combinations without using JavaScript. Furthermore, we suggest a series of CSS-based fingerprinting techniques that identify user-specific configurations, such as the use of translation or browser extensions.

The variety of identified attacks underscores that CSS is more capable of fingerprinting features than expected. Additionally, our fingerprinting methods require fewer assumptions and yield higher accuracy than previous techniques. Our new techniques do not rely on browser-specific HTML elements, such as iframes [22], [23], and are currently the only viable technique in a more restricted environment that prohibits iframes. This observation introduces a new type of attack: *email client fingerprinting*. In this threat model, attackers attempt to link web visitors to email recipients or anonymous email accounts. Additionally, the information gathered about the recipient's environment can be used to improve phishing mails or targeted exploits [18]. To evaluate the threat of CSS-based fingerprinting in email clients, we analyzed 21 native and webmail clients. Our analysis reveals that even if email clients implement mitigations against script-less fingerprinting, they often use spot mitigations that only block previously exploited features. Our techniques are effective in most email clients, enabling attackers to identify email clients (i.e., users) when victims read an email controlled by the attacker. Additionally, we discovered a lack of isolation in one webmail client, which allows for the extraction of all email subjects using only CSS (assigned CVE-2024-24510).

The techniques presented in this paper are currently unmitigated by all state-of-the-art defenses. We thus propose two fundamental defenses to target the problem at its root while retaining full compatibility—one for browsers, one for email clients. For browsers, we preload all conditional resources referenced in CSS styles. This eliminates leakage signals from conditional requests via unconditional preloads, preventing attackers from learning the fingerprinting properties. We evaluate this defense on the Tranco Top 200 websites [24], showing an average increase in network traffic by 30 %. For email clients, we present a mitigation that can be deployed as an email proxy service to protect against leakage through HTML emails. The proxy fetches all remote resources and inlines them, guaranteeing the email's integrity and the recipient's privacy. Additionally, the proxy converts top-level stylesheets to style attributes, the most supported method for defining styles in HTML emails. This approach ensures integrity among different clients while confining the email's style to its own context.

Contributions. We summarize our contributions as follows.

- We systematically explore modern CSS features and present three novel CSS-based fingerprinting techniques based on "`@`" rules, arithmetic functions, and container queries. We show how these CSS-based fingerprinting primitives outperform prior anecdotal techniques regarding versatility, accuracy, and applicability.
- We investigate new fingerprinting use cases in environments where attackers are unable to use certain HTML elements, such as iframes. In particular, we showcase CSS-based fingerprinting in web and standalone email clients.
- We propose two fundamental defenses that convert conditional resource loading into unconditional ones. Although these defenses introduce a considerable overhead, they remove the root cause for all previously known, new, *and* potential future CSS-based fingerprinting techniques.

Responsible Disclosure. We reported our browser-fingerprinting findings to Tor and Brave, which consider fingerprinting a threat. The Tor project acknowledged our techniques and discusses possibilities for effective mitigations. We reported the missing email isolation in the Alinto SOGo client (CVE-2024-24510). Alinto applied a hotfix in their nightly version and is investigating principled mitigations. In addition, we are in the process of reporting the bypass on the filtering of inline styles to Mozilla and the ability to execute JavaScript inside of remote iframes loaded by emails using Samsung Email to Samsung.

Availability. All experiments and proof-of-concepts are available at https://github.com/cispa/cascading-spy-sheets.

II. Background

In this section, we introduce the background for the paper.

A. Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a stylesheet language used on the web in addition to Hypertext Markup Language (HTML) [25]. CSS annotates the style in which the HTML content is displayed. On a high level, a web page is processed as follows: The HTML content is fetched, the HTML is parsed, and further resources such as CSS are loaded. The CSS is parsed, and the contained style information is applied when rendering a web page. CSS is crucial for the modern web as it plays a fundamental role in responsive design, i.e., allowing a website to adapt to the user agent. For example, a developer can use CSS to fit content to the screen resolution [26]. Due to its flexibility, CSS is used beyond the web in other user agents, such as email clients or ebook readers [25].

While every modern browser supports the CSS standard, the exact set of supported features differs per browser [25]. Furthermore, the CSS standard is continuously developing as the web keeps evolving. Developers of user agents, e.g., browser vendors, rapidly adopt many of these new changes. The frequently changing landscape of CSS features makes it hard to reason about the privacy implications of these features.

B. Tracking Users on the Web

For purposes such as targeted advertisements, various entities are interested in tracking a user's browsing behavior. Until recently, tracking parties mainly leveraged third-party cookies to collect user information. As of January 2024, Google is implementing a new set of features called *Tracking Protection*, part of their *Privacy Sandbox* initiative [4], [27]. With the adoption of tracking protection, third-party cookies are no longer a reliable fingerprinting tool.

Hence, *browser fingerprinting*, i.e., deanonymizing a user's browser based on its characteristics and configuration, becomes increasingly important. Browser fingerprinting is typically performed using JavaScript due to its feature-richness [6], [15]. Consequently, users wanting to minimize fingerprinting disable JavaScript. An example is the Tor browser that ships with the *NoScript* extension [28], [29].

Recent research has shown that browser fingerprinting is not limited to JavaScript; CSS also allows fingerprinting users [22], [30]. As a response, Tor has reduced the precision of CSS features related to the viewport, i.e., the screen dimensions [31]. Additionally, the Tor and Brave browsers mitigate font fingerprinting, a technique where websites track users based on the fonts installed on their system [29], [32]. The Tor browser uses an allowlisting-based approach, which features a fixed set of fonts available on all systems, effectively making every user's browser appear identical regarding font availability [29]. The Brave browser randomizes the set of reported fonts for each website [32]. This method creates inconsistencies in the font data available to trackers, making it challenging to build a consistent fingerprint. Additionally, the *NoScript* extension can block iframes, further hardening browsers against CSS-based fingerprinting [28]. Hence, mitigations exist for JavaScript-based fingerprinting and the most potent CSS-based fingerprinting [22].

C. Tracking Users from Their Email Clients

Tracking users in email clients is fundamentally different from fingerprinting on the web. The content of email clients is highly restrictive, e.g., JavaScript is disallowed, subdocuments are often disallowed, and generally, dynamic content is more restricted. Nevertheless, emails support the embedding of HTML and CSS, enabling so-called tracking pixels [33]. A tracking pixel is a transparent small image embedded in the email. The tracking party logs all metadata on the server side when the client fetches the tracking image. This metadata, which can be used to identify the user opening the email, includes the user agent, the IP address, the timestamp, and other headers that may reveal sensitive information (e.g., `Referer`) [33]. As a countermeasure to protect users' privacy, email clients may choose to fetch resources via a proxy server [33]. This way, instead of leaking their own information to the tracking party's server, it remains hidden behind a proxy server. More drastic countermeasures are disallowing external content and stripping tracking pixels from emails (e.g., the *DuckDuckGo Email Protection*) [34].

III. Systematic Analysis of CSS-based Fingerprinting

In this section, we systematically analyze CSS for its use in script-less fingerprinting. We manually survey modern CSS features for features that come close to code execution, specifically conditionals and arithmetic operations. We investigate CSS at-rules [35] that change the behavior of the CSS engine (Section III-B). We analyze dynamic functions in CSS (Section III-C) and CSS properties influenced by the environment (Section III-D) via fuzzing and template attacks [18], respectively. We also analyze different exfiltration channels that attacks can use to gather the leaked information (Section III-E).

A. Threat Model

CSS-based fingerprinting is especially relevant in the context of existing anti-fingerprinting mitigations that mainly defend against client-side scripting [28], [29]. We assume the most restrictive mitigation, i.e., no JavaScript available. We further assume the tracking party controls a web server that is reachable from the victim user agent. For most attacks, we assume that the tracking party controls the HTML document object model (DOM), including the stylesheets. While this is a reasonable assumption for fingerprinting in the browser, other user agents are more restrictive. Thus, we also discuss attacks that do not always require full DOM access on more restricted user agents, e.g., email clients or browsers with additional protections, such as blocking web fonts.

B. CSS At-Rules

We systematically analyze all 17 at-rules from the CSS standard [35]. CSS at-rules are meta statements that control how CSS is interpreted. Such at-rules can, e.g., be a `@charset` statement defining the encoding of a CSS document or an `@import` statement that is used to include external style sheets. We group CSS at-rules into different categories based on their semantics: environment, browser, and style (Table I). Especially interesting for fingerprinting are the environment and browser categories. Previous work only evaluated two of these at-rules, namely the `@media` rule and the `@font-face` rule [22], [30]. In addition, they used `@supports` to differentiate a fixed selection of browsers [22].

We manually investigate all rules for their fingerprinting potential. We identify 1 previously unexplored rule with fingerprinting potential: `@container` allows to query various

TABLE I: We divide all CSS at-rules into 3 categories based on their semantics. **Bold** rules are exploited in this paper, *italic* rules are used in previous work.

Env.	@*media*, @scope, **@page**, **@container**, @*font-face*
Browser	**@*supports***, **@import**
Style	@starting-style, @keyframes, @counter-style, @font-feature-values, @font-palette-values, @property, @layer, @charset, @namespace, @color-profile

dimensions of elements. Additionally, we identify 2 rules with the potential to leak information in the more restricted setting of emails: `@import` to include additional, potentially non-sanitized CSS, and `@page`, which triggers when printing a document (e.g., a web page or email). Section IV presents our fingerprinting techniques based on these previously unexplored at-rules. Furthermore, we analyse the effectiveness of using `@supports` to differentiate major browser releases.

C. CSS Functions

We systematically analyze CSS functions, allowing dynamic calculation of values. CSS functions can be used to evaluate predefined statements, e.g., calculations, whose results can be the value of a CSS property. These functions can, e.g., be used to fit content to different screen sizes dynamically. The MDN Web docs currently list 103 different value functions that are grouped into 12 separate categories [36].

In line with CSS at-rules, we do not see any fingerprint possibility with purely style-targeted functions. These functions account for 9 out of the 12 function categories. Here, we are mainly limited due to a lack of CSS-based exfiltration channels that can amplify implementation differences to other styles or even network requests. We see clear fingerprinting potential in the remaining 3 categories, i.e., mathematical, image, and reference functions. For the reference functions, `env` allows direct access to specific environment properties, leaking information about, e.g., the device's display shape. For mathematical functions, the result of complex arithmetic expressions depends on the implementation details of the browser, which differ across versions, operating systems, and CPU architectures. Section V-A details our fuzzing-based approach for finding inputs to these functions that enable fingerprinting. In the image function category, the `image-set` function may take multiple arguments with attached constraints on supported formats or the viewport. The browser only loads and renders the first image where the constraint is fulfilled, thus leaking information via loading remote images.

D. CSS Properties

We systematically analyze the default values of CSS properties to detect values leaking information from the environment. CSS properties are used to define the style of HTML elements. An example of such a property is `color: red`, which changes the text color of the HTML element it is assigned to. While a set of standardized properties that all browsers support exists, there are additional browser-specific extensions such as `-moz-*` for Firefox and `-webkit-*` for WebKit-based browsers [30]. For this analysis, we use a template attack [18] to determine the fingerprinting potential of CSS properties. We reveal differences induced by the environment by automatically querying all CSS properties of supported HTML elements in the same browser running in different environments. Section VI details this approach and the found properties that have the potential for fingerprinting. We categorize such properties based on the type of value they encode, e.g., dimension or color. We show that properties that influence dimensions can be leaked with existing exfiltration techniques [22] and our new exfiltration techniques described in Section VI. We discuss that the CSS standard does not prevent exfiltrating other properties, which should be considered when browser vendors further implement features of the current CSS standard.

E. Exfiltration Phase

After leaking fingerprint features, they have to be transmitted to the attacker via network requests. There are two options to encode features into network requests. Either the network request is conditional, i.e., the remote resource is only fetched if a specific condition is met, or parts of the URL are feature dependent. As current browser implementations do not allow dynamically generated URLs in CSS, only the first option is possible. CSS specifies 4 variants for such conditional HTTP requests.

Conditional (Group) Rules. Conditional group rules, such as `@media`, `@supports`, or `@import`, allow for exfiltration if their condition depends on a feature. If the styles within the group contain URLs, these URLs are fetched only if the rule triggers. As in previous work [22], [30], we rely on such rules, which are the most generic. Listing 1 shows an example of how such a rule-based conditional HTTP request is capable of detecting the installation of Microsoft's Office Product Suite. It leverages the font *Leelawadee*, which requires a license that allows the distribution and is most commonly installed with this product suite.

Conditional Directives. The `image-set()` directive presents the user agent with a set of resources and corresponding selection criteria, e.g., a specific image format. The user agent fetches the resource that best matches the criteria, e.g., the supported image format with the highest resolution.

Multiple Source Specification. CSS attributes, e.g., the source of `@font-face`, can allow multiple URLs for resources. Upon successfully loading a resource, other fallback resources are ignored. Previous work [22], [30] used this behavior for font fingerprinting. We are unaware of other such attributes, making this exfiltration channel unique for fonts.

CSS Selectors. CSS selectors can become conditional if the DOM changes, e.g., due to an ad blocker. Previous work [22] exploited this behavior to detect the presence of an ad blocker

```
#container {
  container-type: inline-size;
  font-family: 'Leelawadee';
  font-style: normal;
  font-weight: 100;
  font-size: 11px;
  width: 1cap;
}

@container (width > 7.5px) {
  * {
    background-image: url(/office-yes);
  }
}
@container (width < 7.5px) {
  * {
    background-image: url(/office-no);
  }
}
```

Listing 1: The above CSS statement determines whether the Microsoft Office Product Suite is installed on a system by checking for the presence of the *Leelawadee* font.

removing entire HTML elements. Consequently, the remote request of the CSS selector was *not* triggered.

IV. Leakage via CSS Rules

In this section, we introduce novel CSS fingerprinting techniques based on previously unexplored at-rules. In Section IV-A, we show that the recently added `@container` rule is well-suited for browser fingerprinting. Container queries can replace all state-of-the-art techniques for fingerprinting, have fewer requirements, are more challenging to mitigate, and are applicable in more scenarios. Section IV-B shows that `@supports` efficiently distinguishes browsers and browser versions, and `@import` and `@print` have privacy implications when used in emails.

A. Container Queries

Container queries (`@container` rules) introduce the conditional application of styles depending on the computed size of their parent container (e.g., width and height) [25], [37]. Listing 2 shows an example of a container query combining a style with a size query. Modifications to the properties of the container propagate to its child elements via the container query. A container query can be a boolean combination of multiple *container conditions*. A container condition may query a size feature of the container or perform a *style query* to query the computed style of the container. Currently, style queries are only implemented for the `<custom-ident>` CSS data type [25]. The W3C working draft does, however, not mention this restriction [37].

We introduce a novel technique that leverages a comparison of the width (or height) of a container specified using a relative unit and a baseline value specified in an absolute unit (`px`). The technique allows us to infer the absolute size of a relative unit, e.g., `lh` (relative to `line-height`), or `vw` (relative to the viewport width). Using multiple container queries, boolean combinations, or nesting, we find the exact absolute

```
<style>
#container {
  container-type: inline-size;
  width: 20vw;
  --theme: dark;
}
@container style(--theme: dark)
      and (width > 100px) {
  p {
    background-color: black;
    color: white;
  }
}
</style>
<div id="container">
  <p>Lorem!</p>
</div>
```

Listing 2: This example applies conditional styling to the `<p>` tag, if the container has a property called `--theme` which is set to `dark` and the computed width of the container is greater than 100px.

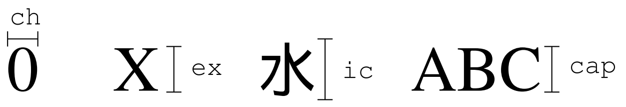

Fig. 1: The CSS length units `ch`, `ex`, `ic`, and `cap` are specified relative to the dimensions of glyphs of the active font.

value. Relative units carry information about the rendering environment, e.g., font availability (see Section IV-A1), user configuration (see Section IV-A2), or the viewport width (see Section IV-A3). In addition, as showcased by Lin et al. [22], the width of HTML elements also presents a great fingerprinting surface (see Section IV-A4).

1) Font Detection: We can infer if a particular font is available on a system using one or multiple font-relative units. As sizes of characters differ between glyphs and fonts, they reliably identify a font. As illustrated in Figure 1, the units `ch`, `ex`, and `ic` refer to concrete sizes of individual glyphs, while `cap` refers to the height of capital letters of a font [25], [38]. Here, `ch` represents the *advance width* of the glyph that represents the character '0' of the current font. Similarly, for international fonts, the `ic` unit represents the height of '水' (CJK water ideograph, U+6C34). The unit `ex` is defined as the height of the glyph representing the character 'X' of a font. Similarly, the unit `cap` is the approximate height of a capital Latin letter.

To test if a specific font is installed and available to the browser, we proceed as follows. We create two containers with the same width of this unit type. We assign the font we want to test for to only one of the containers. We infer the actual size of the container by comparing it to an absolute unit, such as `px`. Directly comparing the width of the container is not possible. By comparing the measurements of the container with the default fallback font applied to the container with the font we want to test for, we can observe if a font is available.

2) Default Property Values: The unit `lh` is defined as the current value of the `line-height` property. While this value is independent of the viewport, each major browser ships with different default line heights. For example, the default value is 18 in Google Chrome and 19 in Firefox. In contrast, the default font size in all major desktop browsers is 16. However, this value is customizable and may uniquely identify users.

3) Replicating Media Queries: The `@media` at-rule, as used in previous work [22], [30], can query information about the viewport. These queries include the width, height, aspect ratio, and orientation. Such viewport-related media queries can also be replicated using container queries, e.g., if media queries are restricted.

We apply a relative viewport width or height, e.g., `vh` or `vw`, to the container. They are each defined as one percent of the width or height of the viewport [38]. Next, we perform a container query that compares the computed width or height with a value specified in an absolute unit. This query effectively leaks the same information that standard media queries can gather. Combined with the units `vmin` and `vmax`, defined as the minimum or maximum of `vh` and `vw`, we can also identify the aspect ratio [38].

4) Measuring Element Dimensions: In addition to the information leaked by dimensional units, we can directly measure the dimensions of HTML elements. For this, we wrap a container (`<div>`) and the element we want to measure the dimensions of in a `<div>` element called the wrapper. We let the wrapper scale to the width of its content using `width: fit-content`. By default, the container scales to the full width of its parent, which is the same width as the element we want to measure. We can leverage a container query to infer the exact size of the element. This technique is visualized in Figure 2. Most browsers adapt their default styles for certain elements (e.g., the `<button>` element) to fit the operating system [22]. In addition, the language setting of the browser also directly influences the width of some elements, such as the `<input>` element with the type `file`, which allows users to choose files from their device storage. The width of elements depends on the *advance width* of the glyphs the element contains and, therefore, the font used for rendering. Note that this technique can also be used for general font fingerprinting. Our approach is similar to Lin et al. [22], which leverages iframes and media queries to achieve the same.

Building on the ability to measure the width of HTML elements, we can also detect the use of translation tools, and their respective target languages. Translation tools built into the browser, such as Google Translate, directly modify the content of the DOM. Those modifications also influence the width of the containing elements since different glyphs are used to render the element. Attackers can provide texts in different languages and detect which are translated. A single different glyph can be leveraged to detect the exact translation. Detecting whether translation happens, or even knowing the target language, provides hints about the languages a user understands and prefers.

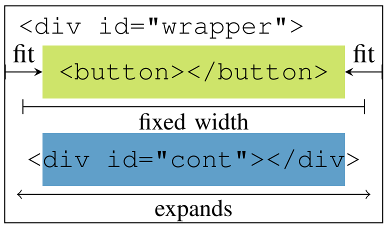

Fig. 2: We can measure the width of an HTML element, e.g., a `<button>`, by wrapping the element and a container in another element. We let the wrapper scale to the width of its children. The width is now defined by the element we want to measure. By default, the `<div>` container element scales to the full width of its parent, which allows us to measure the width of our target by measuring the width of the container.

B. Other Rules

We identify additional rules for supporting fingerprinting.

Support Queries. Support queries using `@supports` allow querying if the current browser enables a CSS feature. By probing which CSS features are supported, we can form fingerprints that uniquely identify browser and operating system configurations in many cases.

External CSS. The `@import` rule allows conditional loading of additional CSS stylesheets. While `@import` does not directly leak information, it is useful in two scenarios. First, as discussed in Section III-E, this rule can be used in the exfiltration phase. Second, we show that this rule is easily overlooked when sanitizing CSS, e.g., in email clients. Any sanitization must be applied recursively to imported stylesheets, which is, e.g., not done in the SOGo email client (Section VIII-C2).

Print Detection. The `@page` rule triggers when a page is printed, similar to the `@media print` rule. The rule does not provide information about the environment but a user's action. Additionally, such a rule can stealthily change the printed email's content or prevent printing the email entirely by hiding the content. Note that currently, only `@media print` provides advanced capabilities, and the support for `@page` is still rudimentary, only supporting margins.

C. Evaluation

In this section, we evaluate the at-rule-based techniques for fingerprinting. Our evaluation is performed on seven major consumer operating systems: Microsoft Windows 10 and 11, macOS 14 Sonoma, Ubuntu 22.04 LTS, ChromeOS 120, Android 14 and iOS 17.3. We include the latest versions of Google Chrome, Microsoft Edge, Opera, Mozilla Firefox, and Apple Safari, covering the vast majority of the market share [39]. Further, we also include three privacy-focussed browsers, Brave, Ghostery, and the Tor browser, in their respective latest versions. A complete table of the browser versions can be found in Table IV in the appendix. Our evaluation contains 1176 pairs of OS-browser combinations. We perform the evaluation on fresh installations, which constitutes the worst-case scenario for the fingerprinting capabilities while preserving the ability to identify systemic differences.

1) Container Queries (Font Detection): We first evaluate our container-query-based font fingerprinting approach using the list of typefaces included with Microsoft Windows 11 [40], the list of typefaces included with Apple Mac OS X 10.0 through macOS 10.14 [41], the allowlist of the Tor browser on Linux [29], and the list of fonts used by FingerprintJS [42], also used by prior work [22]. In addition, we also check the fonts behind the generic font families (e.g., `system-ui`). We exclude the Brave browser, as it follows a randomization-based approach to mitigate font fingerprinting, which would induce false positives in our evaluation. This experiment design leaves us with 861 OS-browser combination pairs. Out of these, we can distinguish 831 (96.5 %) pairs. Our font fingerprinting approach can generally distinguish all operating systems, except when using the Tor browser, where we cannot detect differences between Windows 10 and 11. In addition, we can reliably differentiate the major browser engines. We did not identify differences between Chromium-based browsers, except on ChromeOS and macOS. On ChromeOS, we can distinguish Google Chrome and Opera, the only two available Chromium-based browsers for the system in our evaluation. On macOS, Opera modifies the default font of the generic family `system-ui`, allowing us to differentiate Opera from Google Chrome and Microsoft Edge. We can differentiate Mozilla Firefox-based browsers, except on macOS, where we cannot detect differences between Firefox and Ghostery. Note that we cannot detect differences between the fonts on iOS since the browsers are all WebKit-based.

Bypassing Countermeasures. The Tor browser implements an allowlist-based approach to mitigate font fingerprinting [29]. However, our container-query-based technique identifies a notable exception in Tor's mitigation strategy. Specifically, the browser does not adhere to its font allowlist for the font family "Gill Sans", a licensed font distributed with Microsoft Office [43]. This oversight allows the detection of Microsoft Office on a system, even on the highest security level of the browser. The Tor project acknowledged the issue, indicating an awareness of the exception. However, they (wrongly so) expected this to be mitigated by the other font fingerprinting mitigations (i.e., disabling scripting and loading Web fonts). The behavior seems to stem from a distinct code path for "Gill Sans" in Mozilla Firefox, potentially linked to Bug 551313 [44]. As a result of our report, we are working with the project maintainers to find a solution.

2) Support Queries: We analyze the uniqueness of the features browsers and their major releases provide by leveraging `@supports` rules. We rely on CSS feature-compatibility data provided by Mozilla via their MDN Web Docs [25], [45]. The dataset features the 6 major desktop browsers: Chrome, Edge, Firefox, Opera, Safari, and Internet Explorer. In addition, it features the 5 major mobile browsers: Chrome Android, Firefox Android, Opera Android, Safari iOS, and Samsung Internet. Since Safari on iOS is also used as a rendering engine for the `WebView` component, the dataset also features the Android equivalent *WebView Android*. For these 12 browsers, the dataset records 922 different major releases. Out of these, 702 (76.1 %) releases are unique in the CSS features they support, which attackers can fingerprint using `@supports` rules.

Looking at the major three desktop browsers (i.e., Chrome, Firefox, Safari) and their latest 10 releases recorded in the dataset, we find that 16 out of the 30 releases are unique in their feature set. There are no overlaps between different browser engines. For browser versions, we only find overlaps of size 2, except for Firefox, where Firefox 121 to 124 have the same features. Note that this may also stem from an incomplete dataset, as the current major release of Firefox is 122. As such, the CSS rule `@supports` can distinguish browser engines and, in most cases, their major releases.

In addition, 39 CSS features are first deployed behind flags that users must explicitly enable. On the other hand, features may also be turned off. The modification of these feature flags can likewise be inferred via CSS. As such, the browser-supported features may identify a user who has customized their browser via the feature flags.

3) Translation Identification: We evaluate the width-measurement-based translation identification technique using the built-in translation tool of Google Chrome that leverages the Google Translate service. For this, we translate the sentence "I can understand the words used by my grandfather and like coffee." into 131 languages supported by the Google Translate API. We find that the rendered width of the translation is unique for 127 languages and can thus be accurately distinguished using our container-based width measurement technique. The only exceptions are two language pairs, where the width of the translation is the same: *Simplified Chinese* and *Traditional Chinese*, and *Filipino* and *Tagalog*. For the first pair, all fonts are monospaced and only the visual of a character differs, never the amount of characters. The second is Filipino and Tagalog, which the API does not seem to differentiate. Also note that Filipino is the standardized form of Tagalog.

V. Leakage via CSS Functions

In this section, we introduce a novel source of scriptless leakage based on CSS functions. These functions allow dynamic calculation of property values based on environment properties, such as the viewport size. Based on our systematic analysis (cf. Section III), we focus on two of these functions, `calc` and `env`. Section V-A shows differences in the results of `calc` functions between browsers, operating systems, and CPUs. Section V-B shows that the environment variables accessible using `env` provide information about the used device. As an exfiltration channel, we use container queries to query the width of a container that was previously calculated using a `calc()` or `env()` expression. These functions can also be used in media queries as an alternative exfiltration channel.

A. CSS `calc()` Expressions

CSS allows mathematical expressions and functions to dynamically compute the value of properties using the `calc()`

function [38]. As an example, it can be used to responsively calculate an object's width with respect to its parent's width: `calc(100%/3 - 2*1em)`. Components of a `calc` expression may be literal values (e.g., `5px` or `123`), mathematical functions (e.g., `sin()`), or other expressions that evaluate to valid argument types. Per standard, user agents must support expressions of at least 20 terms. The expressions are typed and may be used with and as numeric values of most numeric data types (e.g., `<length>` units). The standard mandates type checks to be conducted during sub-expression evaluation [38].

The computed value is clamped to the range depending on the target context of the computation. Note that these ranges are only partially standardized. For example, the `width` property may not accept negative values [38]. However, it may accept fractional values of the unit `px`. Here, the standard does not mandate the number of fractions that must be representable. We found that Chromium- and WebKit-based browsers allow representable values for the unit `px` that are multiples of $\frac{1}{64}$. Firefox only supports 61 representable numbers in the range $[0, 1]$ that do not share a common factor. In addition, the representable values are only in the range $[0, 33554428]$ for Chromium-based browsers and $[0, 8947841]$ for Firefox. These implementation-specific limitations highlight that `calc()` expressions can be a source for fingerprinting. Besides these details, there are also differences caused by implementation differences in the browser and third-party math libraries.

Finding Differences. We employ a differential grammar-based fuzzing approach to generate and test expressions to find implementation-specific differences when evaluating complex expressions. Our fuzzer searches for differing results and automatically explores the limitations of the mechanisms that implement parsing, type checking, and evaluation of the expressions by automatically generating test cases that adhere to the grammar specified in the CSS standard. It is entirely implemented on the client side, where we directly evaluate the expressions to retrieve the results.

Our fuzzer generates expressions that include arithmetic operations (i.e., `+`, `-`, `*`, `/`), function calls supported by all major browsers, and nested `calc()` expressions. Currently, only trigonometric functions (e.g., `sin`, `asin`) are widely supported, with additional functions (e.g., `abs`) [38] described in the standard but not yet universally implemented [45]. The depth of generated expressions can be controlled by configuration parameters, allowing for targeted testing of specific limitations. Each test case is tested directly on the DOM by creating a `<div>` element and applying a style that features the width set as the expression. If the attacker-measurable width differs between browsers or operating systems, the underlying expression can be used as a fingerprinting channel.

B. CSS `env()` Function

The `env()` function, as defined in the CSS Environment Variables Module Level 1 draft [46], allows using predefined environment variables in CSS functions. The draft defines two types of environment variables: `safe-area-inset-*` and `viewport-segment-*`. The former defines the inset required on the four sides to create a rectangular display area. For example, notched displays require special care when placing content, as the notch may obscure parts of the interface. Similarly, the latter is used to define the position and dimensions of logically separate regions of the viewport. For example, this is used with folding devices where the hinge creates two separate display areas. Since these values depend on hardware features, they provide an additional fingerprinting vector to differentiate (particularly mobile) devices.

C. Evaluation

In this section, we evaluate the leakage of the `calc` and `env` functions using the same targets as in Section IV-C.

1) Mathematical Differences: We evaluate our fuzzing approach for the `calc` by creating 10 000 expressions using the same random seed and collecting their results. We then leverage the collected data to find differences between browsers' results. These expressions can directly be combined with container queries to generate test cases that differentiate browsers. Concretely, our expression reveals differences in 1116 OS-browser combination pairs (94.9 %). We can always distinguish browser engines based on the results of their functions (e.g., Firefox Gecko and Chromium Blink). In addition, the implementation of browser engines changes frequently, allowing us to distinguish major releases. For example, the 10 recent major releases of Mozilla Firefox feature four different behaviors. Most importantly, we can differentiate release 115 (extended support), used in the Tor browser, and the two most recent releases, 121 and 122. We can also differentiate Opera from the other Chromium-based browsers, as the most recent release, 106, still uses Chromium 120, while the others have already updated to Chromium 121. Further, we can also generally distinguish operating systems. The only exceptions are Apple Safari on Apple Silicon, which behaves the same as the WebKit engine on iOS, and Windows 10 and 11, where we can only distinguish 36 of the 49 browser combinations (73.4 %).

Distinguishing Architectures. The `calc()`-expression-based technique can also differentiate instruction set architectures (ISAs). For example, we can differentiate Microsoft Edge on Windows 11 on ARM and x86-64. Furthermore, we can distinguish between a browser's 32-bit and 64-bit versions, even for the Tor browser on Windows 11. These are the first architectural differences observable from CSS only.

2) Screen Geometry: We evaluate the `safe-area-inset-*` environment variables for fingerprinting on all 5 most recent iPhone generations (i.e., iPhone X up to iPhone 15). As of 2023, Apple's iPhones account for nearly one quarter of the global smartphone market share [47]. We observe differences in the values of the variables when the devices are in landscape mode, which is explained by the devices' notch not obscuring any content when the site is initially loaded. Here, we find 3 clusters among the values of the 5 devices when using Safari. The iPhone X and the iPhone 15 are unique, while the iPhones 12-14 appear the same concerning the values of the variables. Interestingly, Google's Chrome, which on iOS also uses the WebKit engine, presents two clusters. The first cluster is again the iPhone X, where the values are the same as on Safari. All 4 remaining phones are in the second cluster, distinct from any value combinations observed using Safari. As such, this feature also allows differentiating mobile browsers. These findings align with the reported viewport sizes when in landscape mode.

VI. Leakage via CSS Properties

In this section, we introduce a template-attack-based approach to detect CSS properties that leak values from the environment. The central intuition of this approach is that operating systems, browser extensions, and user settings influence CSS properties accessible to websites. We show that our automated approach finds orders of magnitude more leaking properties than previous work [22]. The evaluation is performed in the same fashion as described in Section IV-C. As such, we detect any influences the default configuration of the OS or the browser has on the base styles of HTML elements. Further, we detect browser extensions that inject stylesheets in the context of a site controlled by a tracking party.

A. Finding Environment Differences

In the following, we describe our template-attack-based approach to detect CSS properties that differ between environments. We employ an HTML document containing all elements described in the HTML5 standard. The default styles of all elements in one environment, e.g., Chrome on Windows 10, are then compared against a different environment, e.g., Firefox on macOS 14. All detectable differences can automatically be encoded into their corresponding generic exfiltration channel. For example, we can infer the `width` property of an element by first making the element a container and then using a container query that depends on the container's width.

Technically, we frame the outer HTML document to a fixed size of `200x200` to eliminate direct influences of the viewport dimensions. We then use JavaScript to collect the computed styles of every HTML element in the document via `getComputedStyle`. Note that JavaScript is not necessary for fingerprint generation and is only used for the initial data collection. We collect styles for every HTML element together with its XPath, allowing the detection of stylistic differences induced by element nesting.

As of the writing of this paper, container queries are still limited to querying dimensions (e.g., width). We thus only consider those CSS properties that an attacker can actually exfiltrate. In the future, container queries will likely be fully implemented as described in the `@container` standard. Consequently, they could become a general way to detect *arbitrary* differences in an element's representation. We include a hypothetical what-if analysis to show how expanding the set of available exfiltration channels would increase fingerprints.

In line with prior work on extension fingerprinting [14], we propose a novel CSS-based approach to detect modifications of the DOM and added styles. This is commonly done by browser extensions that inject stylesheets into the context of a site. As such, we can detect and identify the presence of browser extensions that modify the DOM. We can directly leverage the framework introduced by Laperdrix et al. , which identifies extensions that inject stylesheets and pinpoints the injected styles [14]. In the following, we can again filter the injected stylesheets for styles where a CSS-based exfiltration channel exists and automatically combine the trigger element generated by the framework by Laperdrix et al. with the corresponding exfiltration channel.

B. Evaluation

In this section, we evaluate the differences in the computed styles of HTML elements and the ability to detect browser extensions using only HTML and CSS. For the first part, we use the same set of browsers and operating systems as described in Section IV-C.

1) Leaking Properties: As of the writing of this paper, container queries are limited to the dimensions as described in Section IV-A. Thus, our evaluation focuses on the properties that can be inferred using our container-query-based approach. This includes all element-dimension-related properties (e.g., `width`) as well as the line height and the font family and size, as they directly influence length units, which we can query as established in Section IV-A. Note that the evaluation is conducted in an iframe with a fixed viewport to eliminate viewport-induced differences.

We can differentiate 97 % of all browser-OS combination pairs in our investigation. Generally, our templating framework finds differences between all operating systems and browser engines. Here, the only exception is Google Chrome on Windows 10, which cannot be differentiated from Google Chrome and the Brave browser on Windows 11. In addition, we can usually detect differences within the browsers that use the same engine on the same OS. Notable exceptions are the Brave and the Ghostery browsers, which can only be distinguished from Chrome and Firefox, respectively, on our ChromeOS device. Further, we also cannot distinguish Chrome, Edge, and Brave on Ubuntu; Chrome, Edge, Brave, and Opera on macOS; Chrome, Brave, and Opera on Android.

Interestingly, we can identify Opera on iOS, although all browsers on iOS use the same engine. This is because Opera injects stylesheets that slightly customize the styles of HTML elements. In particular, Opera slightly enlarges some form elements. All other iOS browsers are not distinguishable via their default styles. In the hypothetical scenario of an exfiltration channel for every CSS property, we can differentiate 11 additional pairs, increasing our accuracy to 98 %.

2) Extension Detection: We also test how CSS properties can be used to detect DOM modifications by browser extensions. We test 10 Chrome extensions: 9 extensions that serve as examples in the artifacts of prior work [14] and the popular

NoScript extension, which is relevant to the CSS fingerprinting threat model.

We can construct CSS snippets that perform conditional requests and allow fingerprinting for all extensions. In two cases, we can leverage simple CSS selectors as the extensions modify the DOM. One extension, e.g., adds a class name to an existing HTML element of the DOM. In one case, we leverage our container-query-based approach to detect the presence of a stylesheet that modifies the font family for certain elements. Another case leverages the behavior that background images of elements with `display` set to `none` are not fetched. For the remaining cases, we use container queries to detect modifications to the element width.

For example, we provide a CSS snippet that can detect the media-blocking functionality of the popular *NoScript* extension using a simple CSS selector. In place of the original media element (e.g., `<audio>`), the extension places an `<a>` tag with a hyperlink reference to the remote file. This tag has a unique class name, `__NoScript_PlaceHolder__`, which we can target using a class name CSS selector that triggers a request as described in Section III-E. Note that the approach would also work without this class name, as we could also leverage relative selectors (e.g., the child selector) or measure the width of the element using container queries.

VII. Case Study: Browser Fingerprinting

In this section, we evaluate the combination of our novel fingerprinting techniques on the 1176 browser and operating-system combination pairs described in Section IV-C. We further show that our techniques work reliably under the highest security setting of the *NoScript* extension, also used by the Tor browser. At the highest security setting, the extension blocks the execution of JavaScript and plugin objects (e.g., Adobe Flash). In addition, the loading of HTML5 audio, HTML5 videos, subdocuments (e.g., iframes), Web fonts, the content of `<noscript>` tags, and any request to the local network are prevented [48]. This security setting also blocks all requests where the browser is not able to identify the request type. Lastly, it also includes a mitigation (*restricted CSS*) against CSS PP0 [49], which prefetches DNS records to prevent the DNS-based exfiltration technique. Similar restrictions are applied in the context of HTML emails.

We evaluate how well our techniques infer implementation specifics of `calc()` functions, available fonts, and the computed styles of all standard HTML elements. We exclude the Brave browser when comparing the font-based fingerprinting due to the randomness-based mitigation described in Section IV-A. While custom fonts installed on a system reveal additional information, we evaluate our techniques on fresh installs of the operating system to concretely identify the information leakage source. Hence, this acts as a lower bound on the fingerprinting capabilities of our techniques. While viewport dimensions can be used for fingerprinting, their values can easily change during a user session, e.g., by maximizing the browser window. We exclude these features from our evaluation for a more stable fingerprint.

Overall, out of the 1176 combinations in our evaluation, we can distinguish 1152 of them (i.e., $97.95\,\%$). The combination of our novel techniques can generally distinguish all operating systems included in our evaluation, including the Tor browser with NoScript, both configured to the highest security level. The only exception here is the Brave browser on Windows 11, which is indistinguishable from Google Chrome on Windows 10. Within an operating system, our techniques can distinguish the major browser engines, i.e., Blink (Chromium), Gecko (Mozilla), and WebKit (Apple). The only exception is Apple iOS, where all browsers are restricted to using the same WebKit engine. Note that this is subject to change with iOS 17.4 [50]. Distinguishing browsers using the same engine is often possible due to the usage of different engine versions and the browser's individual customization. For example, the Ghostery browser ships with a different version of the font `Twemoji Mozilla`, a fallback emoji font of Firefox.

VIII. Email Client Fingerprinting

In this section, we show that our techniques can be applied in highly restricted environments, such as emails. Emails do not allow scripting languages, such as JavaScript, and typically block iframes, thus preventing the technique proposed by Lin et al. [22]. Email client fingerprinting introduces some interesting scenarios that differ from browser fingerprinting. Attacks might aim to link the web sessions of a visitor to their email account or identify all email addresses belonging to a user, which we refer to as *Email Address Linking*. This threat vector is similar to *Session Linking* on the Web [6]. Possible targets include email aliases, throwaway email addresses, and deanonymizing subscribers of mailing lists. Such information can be particularly valuable for spearphishing campaigns. Our techniques can also be used as a security measure to detect compromised email addresses or the leakage of email contents, both via email forwarding and printing of emails.

We investigate the HTML and CSS feature set supported by 21 email clients across different platforms (cf. Section VIII-B). Our analysis shows that 9 email clients allow all our techniques for email client fingerprinting while 18 expose at least one fingerprinting vector. As the subset of supported CSS features differs heavily among email clients, we showcase different ways to track users in 2 case studies (cf. Section VIII-C).

A. *The State of Rendering Engines in Email Clients*

Generally, email clients use rendering engines that are based on those used in web browsers such that CSS-based fingerprinting also applies to email clients. For web clients, such as Gmail, this is trivially the rendering engine of the browser. Native desktop clients often use a rendering engine provided by the operating system, or by a different application on the system. For example, on Windows systems, the Outlook 2007 desktop client uses the Word rendering engine [51]. On iOS and macOS, the Apple Mail client uses the WebKit engine [52]. Similarly, mobile clients often use, or even are forced to use, the rendering engine provided by the operating system via a `WebView` component. Here, the Android WebView component uses the Blink engine [53], while the iOS WebView component uses the WebKit engine [54].

Alternatively, some clients ship with a rendering engine. For example, Mozilla Thunderbird uses the Gecko engine [55] which is also used by the Firefox browser. Also, the KMail client for KDE uses Chromium via the QtWebEngine [56].

B. Analysis of Email Client Features

We analyze 21 different mail clients for their supported HTML and CSS feature sets. The tested clients include web, desktop, and mobile clients. Our findings show that many of the previously discussed CSS features are supported by email clients, making email client fingerprinting a realistic threat.

1) Methodology: We develop a set of test cases that cover the full range of CSS features applicable to fingerprinting, as well as the features required to exfiltrate the information via HTTP requests. Our test cases test the three major at-rules `@media`, `@supports`, and `@container` that are useful for fingerprinting. For media queries, we have a distinct test case for every type of media query. In addition, we check for the implementation of additional at-rules, such as the `@font-face` and `@import` rules. The `@font-face` rule has been the most prominent method for font fingerprinting, while `@import` can be used recursively to load CSS, potentially circumventing filters.

We consider style tags and external CSS via the `<link>` element, both in the head and body of the email. In addition, we test for restrictions on style attributes of HTML elements and the `srcset` and `sizes` attributes of the `<img>` tag. The latter may be used as a replacement for media queries. Some email clients allow using `<link>` tags to include external CSS or even the `@import` rule; we recursively check if the same constraints are applied to external stylesheets.

We further check for email clients that unconditionally load all remote resources. To do so, we have a test case containing two media queries with mutually exclusive conditions. A client that respects the conditions loads one of these resources, while a client that unconditionally prefetches all resources loads both. To also detect the fetching of resources before the email is opened, we rerun test cases without opening the email. We also log the user agent header and the IP address of incoming requests to check for the usage of a proxy service.

2) Results: We analyze 21 different email clients. For a detailed list of clients and their versions, we refer to Table VI in the appendix. Our analysis shows that 17 (81 %) of the clients fetch remote resources without requiring any user interaction. Only SOGo, RoundCube, Thunderbird, and Apple Mail (Desktop) require user interaction by default before remote resources are fetched. Moreover, the commercial clients GMX (Web/Android/iOS), and Samsung Email do not use a proxy for fetching remote resources, leaking at least the IP address and user agent. For Outlook (Web/Desktop/Android/iOS), this depends on the configuration of the server and user. The open-source clients SOGo and RoundCube do not employ a proxy either, although it can be added to RoundCube via a plugin [57]. Despite its general intentions to stop JavaScript execution, we found that the Android application "Samsung Email" allows an HTML email to include iframes that leverage JavaScript. As such, the client permits leveraging state-of-the-art JavaScript-based fingerprinting techniques and does not constitute a restricted context. Thus, we reported the vulnerability to Samsung, and excluded the app from the following statistics.

13 clients support executing more than 90 % of media queries. 9 clients support container queries and allow media queries containing the `calc` function. Nevertheless, one of these 9 clients, namely ProtonMail, prevents the exfiltration of information such that we deem it not fingerprintable. The remaining 8 clients support more than 75 % of HTML that we can leverage for property fingerprinting as described in Section IV-A. This includes the support for HTML elements such as `<input type="file">`, which has been shown to feature good fingerprinting capabilities [22]. Our container-based technique (cf. Section IV-A) can be applied to 8 clients, allowing, e.g., fingerprint the font set, operating system, and default styles. 4 clients do not require any user interaction: Apple Mail on iOS and the iCloud Web client, and GMX on Android and iOS. Note that no evaluated client allows loading remote iframes, thus preventing the fingerprinting techniques presented in prior work [22]. On 40 % of the tested clients, i.e., 8 clients, all techniques proposed in this paper can be applied to fingerprint clients. Table II summarizes our findings. For Microsoft's Outlook, the proxy behavior depends on the server and user configuration [58].

A limitation of email client fingerprinting is the blocking of remote content. As shown by prior work, remote-content blocking is difficult due to the complexity of modern HMTL and CSS [59]. Further, the statements provided by many email clients may mislead users into underestimating the privacy implications of remote content, especially with the adoption of proxy services that eliminate many privacy risks [33].

3) Fingerprinting Framework Comparison: In the following, we compare the ability of different CSS-based fingerprinting frameworks to be leveraged for Email Client Fingerprinting. The comparison of their teciques is tabularized in Table III. For this, we have selected relevant candidates from academia [22], [30], hobbyists [60] and industry [61]. As none of these candidates discusses or evaluates the use in the email scenario, we evaluate their applicability. Note that we ignore information collected via the User-Agent header, as our analysis has shown that remote resources are usually loaded by a proxy service. Table II shows which clients use a proxy service to load remote resources, but are fingerprintable using our techniques.

We notice that the general approach is to collect all information available via media queries. This carries information about the screen size, color support, and user preferences. In addition, all frameworks query feature support to determine the browser. Here, Takei et al. leverage non-standardized CSS properties that accept URL arguments, while all remaining leverage 4 `@supports` queries. Second, font fingerprinting

TABLE II: Support of the container queries CSS feature by tested email clients. ◐ indicates that the remote-content loading requires user interaction. In the case of ProtonMail, exfiltration is prevented using unconditional preloading, which we denote by ○. * means that the proxy of Outlook depends on the server and user configuration.

Client	Type	Proxy	@container
iCloud	Web	✓	●
RoundCube	Web		◐
SOGo	Web		◐
ProtonMail	Web	✓	○
AOL	Web	✓	
Yahoo	Web	✓	
Outlook	Web	*	
Gmail	Web	✓	
GMX	Web		
Thunderbird	Desktop		◐
Apple Mail	Desktop	✓	◐
Outlook	Desktop	*	
Windows Mail	Desktop		
GMX	Android		●
Outlook	Android	*	
Gmail	Android	✓	
Apple Mail	iOS	✓	●
GMX	iOS		●
Outlook	iOS	*	
Gmail	iOS	✓	

is usually conducted using the `@font-face` rule with the exception being Lin et al. , which leverage a combination of `@font-face` and their iframe-based techique.

We identify 14 features that can be fingerprinted in email clients. In contrast, the sophisticated techiques by Lin et al. are largely restricted to the Web setting, due to their use of iframes, leaving only 7 features that could theoretically work in email clients.

C. Case Studies

In this section, we present 2 case studies that highlight how certain fingerprinting techniques apply to email clients (cf. Section VIII-C1), creating new threats (cf. Section VIII-C2).

1) Case Study: Applying Browser Fingerprinting to Email Clients: We demonstrate that leveraging techniques for browser fingerprinting can be used in email clients.

Font Fingerprinting. As a first demonstration, we detect installations of Microsoft Office from Apple iCloud Mail using our container-query-based font fingerprinting technique. Detecting if Microsoft Office is installed is, e.g., relevant for CVE-2017-0199 that exploits the Windows HTML Application Handler using a malicious Office RTF document to gain arbitrary code execution without user interaction [62]. The vulnerability was among the most exploited vulnerabilities in 2022 [63]. Detecting an Office installation using our technique even works in the Tor browser using the *Gill Sans* font due to an incomplete mitigation (Section IV-C). For our proof-of-concept, we leverage a container query that detects if the font *Leelawadee* is installed. As this font is a non-free Microsoft font for the Thai Language, we do not expect users without Microsoft Office to have it installed [64].

```css
#container {
  container-type: inline-size;
  width: calc(...);
}
@container not (width: 293694.0625px) {
  body {
    background-image: url(/not-windows);
  }
}
@container (width: 293694.0625px) {
  body {
    background-image: url(/windows);
  }
}
```

Listing 3: A CSS snippet that leverages the `calc()` expression in order to distinguish Windows 10 and 11 from other operating systems when using Mozilla Thunderbird.

OS Fingerprinting. As a second demonstration, we remotely leak the OS via fingerprinting on Mozilla Thunderbird. The email leverages a `calc()` expression that results in a different result on Windows than on Linux or macOS systems. Since Mozilla Thunderbird is based on the same Gecko engine as used in Firefox, we can leverage an expression that evaluates to a different result depending on the OS when using Firefox 115 ESR. Interestingly, while the expression also evaluates to different results depending on the OS in Thunderbird, those results are also different from the results produced by Firefox ESR. As such, we are able to distinguish not only the OS from Thunderbird but also Thunderbird from Firefox. Listing 3 provides an example of a CSS snippet that is capable of distinguishing Windows 10 and 11, from other operating systems when using the Desktop Client of Mozilla's Thunderbird. Only the URLs have to be modified to point to a remote server. It leverages a `calc()` expression that was left out of the listing due to its complexity. In total the expression features 5 function calls, 3 constants and 22 arithmetic operations.

2) Case Study: Email-specific Threat Vectors: As a second case study, we demonstrate threat vectors specific to the context of email-client fingerprinting.

Bypassing Blocklists. Our test suite detects 2 clients where externally loaded CSS is not subject to the same constraints as inline styles. According to our tests, this affects Mozilla Thunderbird and Alinto SOGo. With both clients, external CSS can use container queries while they are removed from inline styles. This even works with data URLs, circumventing the need for the user to allow loading remote content explicitly. Thunderbird still requires the user's consent for any requests issued by CSS, as long as this is not generally allowed in the Thunderbird settings.

TABLE III: Features that can be identified using the techniques of CSS fingerprinting frameworks from Takei et al. [30], Lin et al. [22], `csstracking.dev` [60], and No-JS fingerprinting by the maintainers of FingerprintJS [61]. We denote features that work on all discussed user agents by ●, whereas features that only work on a subset of user agents are denoted by ◕. Features that would theoretically work on a subset of clients but were not discussed by the respective works are denoted by ○. Features that can only distinguish some instances of a class are denoted by ◐.

	Environment / Feature	Web					Email				
		[30]	[60]	[61]	[22]	**Ours**	[30]	[60]	[61]	[22]	**Ours**
System	Browser	●	●	●	●	●	○	○	○	○	◕
	Browser Major Version	◐			●	●	○			○	◕
	Operating System	◐			●	●	○				◕
Hardware	Screen Resolution	●	●	●	●	●	○	○	○	○	◕
	Screen Geometry					●					◕
	Color Support		●	●	●	●		○	○	○	◕
	Architecture					●					◕
User	Client Language				●	●					◕
	Installed Fonts	●	◐	◐	●	●	○	○	○	○	◕
	Font Preferences				●	●					◕
	User Preferences		◐	◐	●	●		○	○	○	◕
Plugins	AdBlocker Identification				●	●				○	◕
	Extension Identification					●					◕
	Translation Identification					●					◕

In the case of SOGo, using the `@import` rule to load external CSS does not only circumvent the restrictions imposed on the allowed styles but also breaks email isolation. SOGo restricts the styles of an email to only the context of that email using a technique known as *namespacing* [65] in combination with filtering the stylesheets. They prepend a unique identifier to every class name or identifier in the HTML or CSS. The same procedure is not applied to external stylesheets included from a remote server. This allows for performing an attack known as *Blind CSS Data Exfiltration* [66]. In the case of SOGo, this allows exfiltrating the subjects of other emails in a victim's inbox. Our findings were acknowledged by Alinto and received *CVE-2024-24510*. The issue has since been fixed in SOGo Version 5.9.1.20240124-1.

Print Detection. Using `@media print`, the sender of an email may set up a notification mechanism for if and when an email is printed. Benign usage of this rule ensures that a document is optimized for printing (e.g., large background images are removed). However, this rule also allows loading remote content upon initiating the print dialog from the browser.

Forward Detection. The supported CSS features of an email client often directly reveal the email client used to open an email. This technique can detect if an email is forwarded to or opened by a different client. Further, even if the client remains the same, a difference in the CSS fingerprint of the client opening the email may indicate either a compromise of an email account or the forwarding of the email to an unintended recipient. The fingerprint could even be leveraged directly in the email to obfuscate the content if forwarded.

Conditional Content. Instead of fingerprinting, our techniques can be exploited without an exfiltration step, e.g., in spearphishing campaigns. For example, detecting the OS allows dynamically displaying links that lead to exploits for that particular system. It enables attackers to customize and conceal their attack based on the detected environment, increasing the success rate of their exploits.

IX. Mitigating CSS-based Fingerprinting

In this section, we propose approaches for mitigating CSS-based fingerprinting. We demonstrate a browser-based mitigation that prevents data exfiltration via CSS-initiated requests. Additionally, we propose email-client-specific mitigations that are not limited by a standard on HTML emails and can, therefore, use different mitigation strategies.

A. Unconditional Preloading

Exfiltrating information obtained from conditionals via HTTP is a crucial step in CSS-based fingerprinting, as the threat model assumes no client-side scripting capabilities (cf. Section III-A). As our results show, there are numerous ways of obtaining information with modern CSS. However, the methods for exfiltration are limited (Section III-E). Thus, we propose a mitigation that completely removes conditional CSS-initiated HTTP requests. The general idea is to preload all resources that are referenced in the CSS of a site. As such, the computed styles are still conditional, while the HTTP requests prevent the exfiltration of these conditions to a remote server. We implement a proof-of-concept as an extension for Mozilla Firefox. The extension checks for occurrences of the `url()`, `image()`, or `image-set()` functions in external stylesheets, inline styles, and style attributes. For every identified URL, the extension injects a hidden image into the DOM that fetches the identified resource. As such, every remote resource is requested unconditionally, and later requests are served from the HTTP cache of the browser. Note that our mitigation does not account for client-side scripting due to the restrictive threat model. JavaScript could circumvent our mitigations by injecting stylesheets at runtime.

Overhead. We evaluate our extension on the Tranco Top 200 reachable sites [24] to showcase that it presents an acceptable performance trade-off to increase a user's privacy. In our dataset, the median number of requests a site makes is 57. Our mitigation adds 17 additional requests to the median, an overhead of less than 30 %. This corresponds to an increase in the accumulated response body sizes of 225 kB. Note that our naïve approach may also induce false positive preloading, e.g., URLs contained in comments.

B. Email Privacy Proxy

Further, we introduce a proxy service that rewrites emails to enhance the recipient's privacy and the integrity of the email regarding its consistency across email clients. The proxy has two distinct mechanisms that each achieve a different goal. The first mechanism rewrites top-level CSS rules that are defined by `<link>` or `<style>` elements to style attributes. The second mechanism is rewriting remote resources to be included directly in the email via data URLs.

Style Attributes. Rewriting the CSS defined in an HTML email is a significant step in reducing the vast majority of the fingerprinting surface available to emails. This is because at-rules may only be defined by top-level stylesheets, thereby eliminating a major avenue for potential fingerprinting. Our investigation of state-of-the-art fingerprinting has shown the importance of at-rules for CSS-based fingerprinting as almost all techniques use at least one at-rule (e.g., `@font-face`).

In addition, as underlined by our investigation, style attributes are widely supported among email clients. In addition, the use of style attributes carries certain advantages. For example, it prevents the styles defined by the email author from conflicting with stylesheets defined by the client. This is due to style attributes always taking precedence over other stylesheets. Further, it confines the styles of the email to the email itself, as style attributes provide no means to define styles that apply to parent or sibling elements in the DOM. As such, style attributes provide the smallest risk of the same email appearing different when opened using different email clients. It also prevents conflicting styles between different parts of a multipart email (e.g., original emails attached to a response).

Rewriting URLs. The rewriting of remote resources to data URLs serves two purposes. First, it eliminates the exfiltration of fingerprints and undermines the functionality of tracking pixels. Second, it additionally strengthens the integrity of the email since remote resources could be modified or removed at a later point in time, thus making the same email appear different at two different points in time. This further allows for the proper archiving of emails without encountering references to remote resources that are no longer available.

Proof-of-Concept & Overhead. We implement a proof-of-concept using Python that takes a `.eml` file as input. First, we rewrite stylesheets to style attributes using the PyPi package `css-inline` [67]. Second, we perform the inlining of remote images as data URLs for `<img>` tags and images defined via the CSS property `background-image`. Other remote resources occur less frequently and are thus excluded in our proof-of-concept. However, they may be added analogously. We evaluate the overhead on 200 HTML email newsletters from well-known brands received in 2022. On our dataset, the overall size increases from 13 MB to 159 MB. The linear processing approach takes about 11 min, out of which the most time is spent fetching remote stylesheets and images. Note that the service is only relevant for HTML emails and thus introduces no overhead for plaintext emails.

C. Other Email-specific Mitigations

Preloading can also be used for emails. In fact, the email server of ProtonMail unconditionally fetches all remote sources when receiving an email and rewrites the URLs to an internal one. Thus, leakage from CSS cannot be exfiltrated. Still, unlike web browsers, there is no fixed standard for CSS and HTML in email clients, enabling other mitigations.

Preventing Requests. While blocking only CSS features usable for fingerprinting seems infeasible, preventing remote requests is a viable option. A widely-implemented variant, e.g., in Mozilla Thunderbird, requires user permission to load remote content. However, this shifts the decision to the user, who might not grasp the consequences of granting the permission. Alternatively, fully disallowing the inclusion of remote content also mitigates the exfiltration channel.

While many commercial clients, such as GMail, implement a proxy to fetch remote content, this is not sufficient to prevent our techniques. It only stops leakage of the privacy-relevant headers and the IP [33], while the request is still conditional.

Restricting CSS. Mitigations can restrict CSS to a subset that is not usable for fingerprinting. As we see in our analysis, there are restrictive email clients, such as AOL or Yahoo Mail, that only support a small subset of CSS deemed safe. However, restricting CSS is prone to errors, as shown for the SOGo webmail client (cf. Section VIII-C2).

X. Discussion

In this section, we discuss privacy considerations of the W3C and the outlook if the proposals continue to be implemented (Section X-A), as well as related work (Section X-B).

A. Privacy Considerations

Overall, most of our findings are not adequately captured by the privacy concerns of the corresponding W3C specifications.

For example, the specification of container queries does not state *any* privacy considerations [37], [68]. Our research shows that container queries increase the capabilities of media queries and allow accurate font fingerprinting. While the privacy considerations of the `CSS Fonts Module` discuss font fingerprinting in the context of the `@font-face` rule [69], the discussion is restricted to loading *remote* fonts. However, our novel font fingerprinting technique via container queries does not require the loading of remote fonts (cf. Section IV-A). For the `@supports` rule, the standard mentions its potential for fingerprinting [70], but downplays the significance as this information is exposed through a variety of other vectors. However, especially in restricted scenarios, this function may be the only one available. Lastly, while the specification introducing the `@import` rule expands on concerns regarding the threat model of the Same-Origin Policy [71], it does not discuss the implications of recursive inclusions on sanitization. Our work asks for better scrutiny when assessing the privacy impact in formal specifications.

Unfortunately, the threat surface increases further if browsers continue to adapt more specified features. One such example is that implementations of container queries might extend to a generic method of querying the computed styles of HTML elements, not just their dimensions. Our technique would then provide a generic way to replace the JavaScript API `getComputedStyle` and thus enable full browser extension fingerprinting as described by Laperdrix et al. [14].

B. Related Work

Various browser fingerprinting techniques have emerged that collect browser- or device-specific information for identification, improving security, or enhancing the user experience [7], [14], [15], [17], [20]. While different proposals exist to mitigate fingerprinting [72]–[81], the abundance of fingerprinting techniques has demonstrated that the only fully effective mitigation is the disabling of client-side scripting.

Early work on CSS-based browser fingerprinting has leveraged CSS properties unique to a specific browser, such as prefixed properties using `-webkit-*` or `-moz-*` [82]. Further, Takei et al. [30] collected information about the screen and installed fonts using `@media` and `@font-face`. Fifield et al. [11] expanded on the analysis of font rendering for fingerprinting and found that the glyph bounding boxes of Unicode characters can often uniquely identify a user. Lin et al. [22] proposed an iframe-based technique for fingerprinting the size of HTML elements without using JavaScript. This approach can determine the bounding boxes of glyphs, which have been shown to provide an adequate fingerprinting surface.

While previous work has manually found CSS functionality to fingerprint, we systematically analyze and uncover a large set of dynamic CSS functionality. Our work expands on the general area of CSS-based fingerprinting by introducing novel techniques with fewer requirements that work in more restrictive settings (i.e., HTML emails), defeating existing spot mitigations (i.e., blocking subdocuments [28]).

We show that existing fingerprinting techniques that rely on JavaScript also work without such capabilities. As such, we build on the work by Laperdrix et al. [14] that identified almost 4500 extensions in the Chrome Webstore that are fingerprintable by the stylesheets they inject into the context of sites. We show that in many cases, extensions can be fingerprinted from CSS alone, allowing fingerprinting in restricted settings such as the Tor browser or even email clients.

Previous work used the JavaScript `Math` object for fingerprinting [83]. They leverage platform-dependent differences in the computed results of trigonometric functions (e.g., `Math.sin()`) to identify the operating system from Firefox and distinguish Chrome and Firefox on Windows, Linux, and Android [83]. The approach is, however, not generally able to differentiate browser versions or the OS from a Chromium-based browser in addition to requiring JavaScript.

Prior work on the privacy implications of email tracking has identified various risks that have led to the large-scale adoption of remote-content proxying [33], [84]. Widespread privacy risks were introduced mainly by leaking information to third parties via headers, e.g., `Referer`, or even directly via the URL [33]. Additionally, third-party cookies allow tracking of users similar to Web tracking [1], [2].

XI. Conclusion

This paper shows the growing challenges in preventing browser fingerprinting due to the evolving complexity of web standards. Our research shows that modern CSS is potent enough to provide fingerprinting without JavaScript. Based on fuzzing and templating, we uncovered innovative techniques, exploiting nuances in container queries, evaluation of arithmetic expressions, and complex selectors to infer browser and OS configurations. We demonstrated the applicability to the highly restrictive setting of HTML emails, expanding the scope of tracking beyond browsers. We propose a comprehensive mitigation for existing and future CSS-based fingerprinting relying on preloading conditional resources. Our work underscores the need for defenses against tracking methods.

Acknowledgment

We want to thank our anonymous reviewers for their comments and suggestions. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. This work was also partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS). We also want to thank Simon Einzinger for helping with some experiments. We further thank the Saarbrücken Graduate School of Computer Science for their funding and support.

References

[1] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "An empirical study of web cookies," in *WWW*, 2016.

[2] M. W. Docs, "Using HTTP cookies," 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies}

[3] J. Schuh, "Building a more private web: A path towards making third party cookies obsolete ," 2020. [Online]. Available: {https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html}

[4] Google, “Third-party cookie deprecation,” 2023. [Online]. Available: {https://developers.google.com/privacy-sandbox/3pcd}

[5] ——, “Topics API overview,” 2022. [Online]. Available: {https://developers.google.com/privacy-sandbox/relevance/topics}

[6] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, “Browser fingerprinting: A survey,” in *ACM Transactions on the Web*, 2020.

[7] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints,” in *S&P*, 2016.

[8] Fingerprint, “Case Studies,” 2024. [Online]. Available: {https://fingerprint.com/case-studies/}

[9] FingerprintJS, “FingerprintJS,” 2024. [Online]. Available: {https://github.com/fingerprintjs/fingerprintjs}

[10] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless monster: Exploring the ecosystem of web-based device fingerprinting,” in *Security and privacy (SP)*, 2013.

[11] Fifield, David and Egelman, Serge, “Fingerprinting web users through font metrics,” in *FC*, 2015.

[12] J. R. Mayer, “Any person... a pamphleteer”: Internet anonymity in the age of web 2.0,” *Undergraduate Senior Thesis, Princeton University*, 2009.

[13] P. Eckersley, “How unique is your web browser?” in *PETS*, 2010.

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

[15] K. Mowery and H. Shacham, “Pixel Perfect: Fingerprinting Canvas in HTML5,” in *W2SP*, 2012.

[16] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, “The web never forgets: Persistent tracking mechanisms in the wild,” in *CCS*, 2014.

[17] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, “Fingerprinting information in JavaScript implementations,” in *W2SP*, 2011.

[18] M. Schwarz, F. Lackner, and D. Gruss, “JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits,” in *NDSS*, 2019.

[19] 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 *Network and Distributed System Security Symposium*, 2022.

[20] L. Trampert, C. Rossow, and M. Schwarz, “Browser-based CPU Fingerprinting,” in *ESORICS*, 2022.

[21] T. Rokicki, C. Maurice, and M. Schwarz, “CPU Port Contention Without SMT,” in *ESORICS*, 2022.

[22] X. Lin, F. Araujo, T. Taylor, J. Jang, and J. Polakis, “Fashion faux pas: Implicit stylistic fingerprints for bypassing browsers’ anti-fingerprinting defenses,” in *IEEE S&P*, 2023.

[23] M. W. Docs, “¡frame¿,” 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/HTML/Element/frame}

[24] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, “Tranco: A research-oriented top sites ranking hardened against manipulation,” in *NDSS*, 2018.

[25] M. W. Docs, “CSS: Cascading Style Sheets,” 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/CSS}

[26] ——, “CSS media queries,” 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_media_queries}

[27] Google, “Third-party cookie deprecation,” 2023. [Online]. Available: {https://blog.google/products/chrome/privacy-sandbox-tracking-protection/}

[28] Giorgio Maone, “NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!” July 2017. [Online]. Available: https://noscript.net

[29] T. T. Project, “The Design and Implementation of the Tor Browser,” 2019. [Online]. Available: {https://2019.www.torproject.org/projects/torbrowser/design/}

[30] N. Takei, T. Saito, K. Takasu, and T. Yamada, “Web browser fingerprinting using only cascading style sheets,” in *International Conference on Broad-Band Wireless Computing, Communication and Applications, BWCCA*, 2015.

[31] T. T. Project, “Letterboxing,” 2023. [Online]. Available: {https://support.torproject.org/tbb/maximized-torbrowser-window/}

[32] B. Software, “Protecting against browser-language fingerprinting ,” 2022. [Online]. Available: {https://brave.com/privacy-updates/17-language-fingerprinting/}

[33] S. Englehardt, J. Han, and A. Narayanan, “I never signed up for this! Privacy implications of email tracking.” *Proc. Priv. Enhancing Technol.*, 2018.

[34] DuckDuckGo, “DuckDuckGo Email Protection,” 2024. [Online]. Available: {https://duckduckgo.com/duckduckgo-help-pages/email-protection/what-is-duckduckgo-email-protection/}

[35] M. W. Docs, “CSS at-rules,” 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule}

[36] ——, “CSS value functions,” 2023. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Functions}

[37] W. W. W. Consortium, “CSS Containment Module Level 3,” 2022. [Online]. Available: {https://www.w3.org/TR/css-contain-3/}

[38] ——, “CSS Values and Units Module Level 4,” 2023. [Online]. Available: {https://www.w3.org/TR/css-values-4/#math}

[39] Stetic, “Browser Statistics,” 2024. [Online]. Available: {https://www.stetic.com/market-share/browser/}

[40] Microsoft, “Windows 11 font list,” 2022. [Online]. Available: {https://learn.microsoft.com/en-us/typography/fonts/windows_11_font_list}

[41] Wikipedia, “List of typefaces included with macOS,” 2023. [Online]. Available: {https://en.wikipedia.org/wiki/List_of_typefaces_included_with_macOS}

[42] FingerprintJS, “Font List for Fingerprinting,” 2023. [Online]. Available: {https://github.com/fingerprintjs/fingerprintjs/blob/master/src/sources/fonts.ts}

[43] Microsoft, “Gill Sans MT font family,” 2022. [Online]. Available: {https://learn.microsoft.com/en-us/typography/font-list/gill-sans-mt}

[44] BugZilla, “Gill Sans font displays incorrectly when using DirectWrite / Direct2D,” 2010. [Online]. Available: {https://bugzilla.mozilla.org/show_bug.cgi?id=551313}

[45] M. on Github, “@mdn/browser-compat-data,” 2023. [Online]. Available: {https://github.com/mdn/browser-compat-data}

[46] W. W. W. Consortium, “CSS Environment Variables Module Level 1,” 2021. [Online]. Available: {https://drafts.csswg.org/css-env/#env-function}

[47] Gartner, “iPhone unit shipments as share of global smartphone shipments from 3rd quarter 2007 to 4th quarter 2023,” 2024. [Online]. Available: {https://www.statista.com/statistics/216459/global-market-share-of-apple-iphone/}

[48] I. Forums, “NoScript 11.3.3 Customization,” 2021. [Online]. Available: {https://forums.informaction.com/viewtopic.php?p=103816}

[49] A. Shusterman, A. Agarwal, S. O’Connell, D. Genkin, Y. Oren, and Y. Yarom, “Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses,” in *USENIX Security Symposium*, 2021.

[50] Apple, “Using alternative browser engines in the European Union,” 2024. [Online]. Available: {https://developer.apple.com/support/alternative-browser-engines/}

[51] Microsoft, “Word 2007 HTML and CSS Rendering Capabilities in Outlook 2007 (Part 1 of 2),” 2014. [Online]. Available: {https://learn.microsoft.com/en-us/previous-versions/office/developer/office-2007/aa338201(v=office.12)}

[52] Apple, “WebKit - A fast, open source web browser engine.” 2024. [Online]. Available: https://webkit.org/

[53] T. C. Project, “Android WebView,” 2024. [Online]. Available: {https://www.chromium.org/developers/androidwebview/}

[54] Apple, “WKWebView - Documentation,” 2024. [Online]. Available: {https://developer.apple.com/documentation/webkit/wkwebview}

[55] F. S. Docs, “Gecko,” 2024. [Online]. Available: {https://firefox-source-docs.mozilla.org/overview/gecko.html}

[56] Q. Wiki, “QtWebEngine/ChromiumVersions,” 2024. [Online]. Available: {https://wiki.qt.io/QtWebEngine/ChromiumVersions}

[57] evandrofisico, “RoundCube ImageProxy Plugin,” 2024. [Online]. Available: {https://github.com/evandrofisico/roundcube-imageproxy}

[58] Microsoft, “Outlook for iOS and Android in Exchange Online: FAQ,” 2023. [Online]. Available: {https://learn.microsoft.com/en-us/exchange/clients-and-mobile-in-exchange-online/outlook-for-ios-and-android/outlook-for-ios-and-android-faq#q-does-outlook-for-ios-and-android-support-proxy-configurations}

[59] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, “Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels,” in *USENIX Security*, 2018.

[60] O. Brotchie, “CSS Fingerprint,” 2024. [Online]. Available: {https://csstracking.dev/}

[61] Fingerprint, "No-JS fingerprinting," 2024. [Online]. Available: {https://noscriptfingerprint.com/}
[62] U. N. V. D. N. (NIST), "CVE-2017-0199," 2017. [Online]. Available: {https://nvd.nist.gov/vuln/detail/CVE-2017-0199}
[63] U. C. . I. S. A. (CISA), "2022 Top Routinely Exploited Vulnerabilities," 2023. [Online]. Available: {https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-215a}
[64] Microsoft, "Leelawadee font family," 2022. [Online]. Available: {https://learn.microsoft.com/en-us/typography/font-list/leelawadee}
[65] W. W. W. Consortium, "Namespaces in XML 1.0 (Third Edition)," 2009. [Online]. Available: {https://www.w3.org/TR/REC-xml-names/}
[66] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *CCS'12*, 2012.
[67] D. Dygalo, "PyPi - css-inline," 2024. [Online]. Available: {https://pypi.org/project/css-inline/}
[68] W. W. W. Consortium, "Media Queries Level 5," 2021. [Online]. Available: {https://www.w3.org/TR/mediaqueries-5/}
[69] ——, "CSS Fonts Module Level 4," 2023. [Online]. Available: {https://drafts.csswg.org/css-fonts/}
[70] ——, "CSS Conditional Rules Module Level 4," 2023. [Online]. Available: {https://drafts.csswg.org/css-conditional-4/}
[71] ——, "CSS Cascading and Inheritance Level 5," 2023. [Online]. Available: {https://drafts.csswg.org/css-cascade-5/}
[72] C. F. Torres, H. Jonker, and S. Mauw, "FP-block: Usable web privacy by controlling browser fingerprinting," in *ESORICS*, 2015.
[73] A. Faiz Khademi, M. Zulkernine, and K. Weldemariam, "FPGuard: Detection and prevention of browser fingerprinting," in *Data and Applications Security and Privacy XXIX*, 2015.
[74] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri, "Countering browser fingerprinting techniques: Constructing a fake profile with google chrome," in *NBiS*, 2014.
[75] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews, "Disguised chromium browser: Robust browser, flash and canvas fingerprinting protection," in *WPES*, 2016.
[76] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *WWW*, 2015.
[77] P. Laperdrix, W. Rudametkin, and B. Baudry, "Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification," in *SEAMS*, 2015.
[78] P. Laperdrix, B. Baudry, and V. Mishra, "FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques," in *ESSoS*, 2017.
[79] A. Sjösten, S. Van Acker, P. Picazo-Sanchez, and A. Sabelfeld, "Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks." in *NDSS*, 2019.
[80] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, "Everyone is different: Client-side diversification for defending against extension fingerprinting," in *USENIX Security*, 2019.
[81] S. Wu, S. Li, Y. Cao, and N. Wang, "Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting," in *USENIX Security Symposium*, 2019.
[82] Unger, Thomas and Mulazzani, Martin and Frühwirt, Dominik and Huber, Markus and Schrittwieser, Sebastian and Weippl, Edgar, "Shpf: Enhancing http (s) session security with browser fingerprinting," in *ARES*, 2013.
[83] T. Saito, T. Noda, R. Hosoya, K. Tanabe, and Y. Saito, "On estimating platforms of web user with javascript math object," in *Advances in Network-Based Information Systems (NBiS)*, 2019.
[84] H. Xu, S. Hao, A. Sari, and H. Wang, "Privacy risk assessment on email tracking," in *IEEE INFOCOM*, 2018.

Appendix A

A. Browser Versions Used for Testing

Table IV describes the OS-browser combinations used in our evaluation. All devices had all updates installed as of the run of the evaluation on February 3, 2024.

TABLE IV: Tested operating systems and browser versions. All browsers had the latest update available on Feb 3, 2024, installed.

Operating System Version	Browser Version
Windows 10	Google Chrome 121
	Edge 121
	Opera 106
	Brave 1.62.156
	Firefox 122
	Tor 13.0.9
	Ghostery 2023.10
Windows 11	Google Chrome 121
	Edge 121
	Opera 106
	Brave 1.62.156
	Firefox 122
	Tor 13.0.9
	Ghostery 2023.10
Ubuntu 22.04 LTS	Google Chrome 121
	Edge 121
	Opera 106
	Brave 1.62.153
	Firefox 122
	Tor 13.0.9
	Ghostery 2023.10
macOS 14 Sonoma	Google Chrome 121
	Edge 121
	Opera 106
	Brave 1.62.156
	Firefox 122
	Tor 13.0.9
	Ghostery 2023.10
	Safari 17.2.1
ChromeOS 120	Google Chrome 120 (32-bit)
	Opera 80.1
	Brave 1.62.156
	Firefox 122
	Tor 13.0.9
	Ghostery 1.0.2343
Android 14	Google Chrome 121
	Edge 120
	Opera 80.1
	Brave 1.61.152
	Firefox 122
	Tor 13.0.9
	Ghostery 2023.10
iOS 17.3	Google Chrome 121
	Edge 121
	Opera 4.5.0
	Brave 1.61.1
	Firefox 122
	Ghostery 3.2
	Safari 17.2.1

B. Browser Extension Fingerprinting

Table V presents the browser extensions and their respective identifiers in the Chrome Web Store used in our evaluation of browser extension fingerprinting (cf. Section VI-B2). In addition, we display the number of active users of an extension

TABLE V: Browser extensions used in the extension fingerprinting evaluation.

Extension Name	Extension ID	Active Users
NoScript	doojmbjmlfjjnbmnoijecmcbfeoakpjm	100 000+
TTSReaderX In-Page Text to Speech	pakknklefcjdhejnffafpeelofiekebg	100 000+
Touch VPN	bihmplhobchoageeokmgbdihknkjbknd	8 000 000+
AdBlocker by Trustnav	dgbldpiollgaehnlegmfhioconikkjjh	300 000+
MozBar	eakacpaijcpapndcfffdgphdiccmpknp	1 000 000+
Disconnect	jeoacafpbcihiomhlakheieifhpjdfeo	600 000+
TripAdvisor Browser Button	oiekdmlabennjdpgimlcpmphdjphlcha	80 000+
Awesome Screenshot: Screen Video Recorder	nlipoenfbbikpbjkfpfillcgkoblgpmj	3 000 000+
Hunter: Find email addresses in seconds	hgmhmanijnjhaffoampdlllchpolkdnj	600 000+
Screenshot reader	enfolipbjmnmleonhhebhalojdpcpdoo	5 000 000+

TABLE VI: The set of email clients used for the email client fingerprinting evaluation. For web-based email clients, we show the browser and version, if available. For Desktop clients, the version is displayed.

Client	Client Type	Browser/Client Version
Gmail	Web Client	in Chrome 120
AOL	Web Client	in Chrome 120
Outlook	Web Client	in Chrome 120
iCloud	Web Client	in Chrome 120
SOGo	Web Client	5.9.0 in Chrome 120
RoundCube	Web Client	1.6.5 in Chrome 120
Yahoo	Web Client	in Chrome 120
ProtonMail	Web Client	in Chrome 120
GMX.net	Web Client	in Chrome 120
Thunderbird	Desktop Client	11.5.52
Outlook	Desktop Client	20240126002.13
Apple Mail	Desktop Client	16.0
Windows Mail	Desktop Client	16005.14326.21788.0
Gmail	Android Client	2023.11.26.586591930.Release
Outlook	Android Client	4.2352.1
Samsung Email	Android Client	6.1.90.16
GMX	Android Client	7.41
Apple Mail	iOS Client	iOS 17.1.1
Outlook	iOS Client	4.2401.0
Gmail	iOS Client	6.0.231232.1785580
GMX	iOS Client	9.2

as reported by the Chrome Web Store. The set of extensions was highlighted in the artifacts of prior work [14].

C. Email Client Fingerprinting

Table VI showcases the email clients and their respective versions used in our evaluation of email client fingerprinting. All web clients were opened using the same instance of Google's Chrome Version 120 on Ubuntu 22.04 LTS.

Appendix B
Artifact Appendix

A. Description & Requirements

Our artifact contains several proof-of-concepts that demonstrate the feasibility of the techniques described in the paper. Further, the artifact contains the data we collected for the evaluation of web browsers and email clients. The versions of the web browsers are listed in Table IV in the Appendix of the paper, while the versions of the email clients are listed in Table V in the Appendix of the paper. For simple reproduction, we provide our evaluation setup, data and scripts that aggregate the differences between the web browsers. Lastly, we provide our two proof-of-concept mitigations that can be used to evaluate the overhead of such mitigations.

1) How to access: The artifact is publically available at https://github.com/cispa/cascading-spy-sheets and citable via the DOI https://doi.org/10.5281/zenodo.13712489.

2) Hardware dependencies: The artifact can be run on any device that is capable of running a modern web browser.

3) Software dependencies: The techniques described in the paper do not inherently require any software dependencies. However, the artifact contains a set of proof-of-concepts that are tailored to specific software environments. Thus, we recommend running a Windows 11 and an Ubuntu 22.04 LTS system with the latest version of Google Chrome, Firefox, and the Tor Browser.

4) Benchmarks: None. The artifact is self-contained.

B. Artifact Installation & Configuration

The artifact does not require any complex installation or configuration steps. All experiments are either self-contained HTML/EML files or Python scripts. We have tested Python scripts using Python 3.12.4, but any Python 3 version should work. We provide requirements.txt files where necessary and recommend using a virtual environment to install the dependencies. Each proof-of-concept has to simply be opened in the respective software environment.

C. Experiment Workflow

Our artifact features three distinct types of experiments.

The second type of experiment is the evaluation of the fingerprinting surface of web browsers and the feature support of email clients. These experiments correspond to Section VII and Section VIII.A of the paper.

D. Major Claims

We make the following claims in our paper:

- (C1): CSS `calc()` expressions and container queries can be used to fingerprint web browsers. The claim corresponds to the experiments (E1).
- (C2): We can detect the presence of popular browser extensions using only CSS. The claim corresponds to experiment (E2).
- (C3): We can identify the target language of a translation conducted using Google Translate as built into Google Chrome. The claim corresponds to experiment (E3).
- (C4): We can identify more than 95 % of browser/OS combinations using our fingerprinting techniques. The claim corresponds to experiment (E4).
- (C5): Email clients can be fingerprinted based on the support of HTML and CSS features. The claim corresponds to experiment (E5) and (E6).
- (C6): Our proof-of-concept mitigations can be used to reduce the fingerprinting surface of web browsers. The extension-based mitigation has a low overhead of less than 30 % regarding the number of requests issued by a site. The claim corresponds to experiment (E7). Meanwhile, the email proxy adds a significant size overhead of more than 1000 % to HTML emails. The claim corresponds to experiment (E8).

E. Evaluation

You can find more indepth instructions for each experiment in the respective README files in the artifact. In particular, the section *Reproduction Instructions* in the top-level README.

1) Experiment (E1): [calc() and @container Fingerprinting] [15 human minutes]: The experiment shows that CSS calc expressions and container queries can be used to fingerprint web browsers. The experiment is designed for Firefox, the Brave Browser and the Tor Browser.

[How to] Simply visit the provided HTML files with the respective web browser/OS combination. The result is a visual difference of the rendering of the HTML file.

Path: `pocs/browser/`

[Preparation] For preparation, either transfer the HTML files to the respective environment (e.g., virtual machine) and open the file locally or host the HTML files on a web server and visit the URL with the respective web browser.

For convenience, we host the files on our server. Further instructions are provided in the README of the repository.

[Execution] Simply visit the provided HTML files with the respective web browser/OS combination.

[Results] The fingerprint is the visual difference of the rendering of the HTML file which should be able to differentiate between the web browser/OS combinations. The versions this experiment was tested with are stated in the respective HTML files.

2) Experiment (E2): [Extension Detection] [15 human minutes]: The experiment shows that popular browser extensions can be detected using only CSS. The experiment is designed for Google Chrome.

Path: `pocs/extensions/`

[How to] Simply visit the provided HTML files once with the respective extension installed and enabled and once without the extension installed and enabled. The result is a visual difference of the rendering of the HTML file.

For convenience, we host the files on our server. Further instructions are provided in the README of the repository.

[Results] The fingerprint is the visual difference of the rendering of the HTML file which should be able to differentiate between the presence and absence of an extension.

The versions this experiment was tested with are stated in the respective README.

3) Experiment (E3): [Translation Detection] [15 human minutes]: The experiment shows that we can identify a translation performed using Google Translate as built into Google Chrome. The experiment is designed for Google Chrome.

Path: `pocs/browser/poc_chrome_translate.html`

[How to] Simply visit the provided HTML file with Google Chrome. The result is a visual difference of the rendering of the HTML file, when a translation is performed. Simply visit the provided HTML files with the respective translation (i.e., English, German or Catalan).

[Results] The fingerprint is the visual difference of the rendering of the HTML file which should be able to differentiate between the different translation.

4) Experiment (E4): [Evaluation Browser/OS Combinations] [10 human minutes]: The experiment shows that we can distinguish more than $95\,\%$ of browser/OS combinations. It leverages the data collected using the Browser versions listed in Table IV in the Appendix of the paper.

Path: `evaluation/browser/generate_results`

[How to] Simply run the Jupyter Notebook `Results.ipynb`. It generates the fingerprint of each browser/OS combination and generates a matrix of which combinations can be distinguished.

[Results] The result shows that we can distinguish 1141 out of 1196 browser/OS combinations. Additional data can be collected using the setup in `evaluation/browser/` and should not yield significantly different results (i.e., less than $90\,\%$).

5) Experiment (E5): [Evaluation Feature Support Email Clients] [10 human minutes]: The experiment shows that 9 email clients allow all CSS features relevant to state-of-the-art CSS-based fingerprinting.

Path: `evaluation/email/`

[How to] Simply run the Python script `gen_supported_features.py`. It generates parts of Table II in the paper.

[Results] The result shows that 9 email clients allow all CSS features relevant to state-of-the-art CSS-based fingerprinting.

6) Experiment (E6): [Email Client Fingerprinting PoC] [15 human minutes]: The experiment shows that state-of-the-art CSS-based fingerprinting can be applied to email clients. For this, we provide a set of proof-of-concept emails that demonstrates the feasibility of our techniques in some email clients.

[How to] Simply visit the provided EML files with the respective email client/OS combination. The result is a visual difference of the rendering of the email.

Path: `pocs/email/`

[Preparation] For preparation, either transfer the EML files to the respective environment (e.g., virtual machine) and open the file locally or send the EML files to the respective email client. For this, we provide a helper script (i.e., `sender.py`)

[Execution] Simply visit the provided EML files with the respective email client/OS combination.

[Results] The fingerprint is the visual difference of the rendering of the email which should be able to differentiate between the email client/OS combinations.

7) Experiment (E7): [Extension Mitigation] [15 human minutes]: The experiment corresponds to the evaluation of the extension-based mitigation in Section IX.A of the paper.

[How to] The evaluation runs a crawl of the Tranco Top 50 reachable sites with and without the extension-based mitigation enabled. It aggregates network traffic statistics which can be compared using a Python script. Instead of running the crawl yourself, you may use the provided output of our evaluation and simply recalculate the statistics.

Path: `mitigation/browser/eval/`

The README in the respective directory provides further instructions.

[Results] The overhead of the extension-based mitigation should be less than $30\,\%$ regarding the number of requests issued by a site. We expect no overhead greater than $50\,\%$ on a successful crawl.

8) Experiment (E8): [Email Mitigation] [15 human minutes]: The experiment corresponds to the evaluation of the extension-based mitigation in Section IX.B of the paper.

[How to] The evaluation of the email privacy proxy runs on a set of EML files.

Path: `mitigation/email/`

The README in the respective directory provides further instructions.

[Preparation] First, export a set of emails from one of your inboxes. The EML format is supported by a wide range of clients. We recommend using Thunderbird.

[Execution] Next, run the provided script on the exported files. The script will fetch all remote resources contained in the files and inline them as data URLs. The output will be to a new folder. Next, compare the sizes of the initial and the output folder.

[Results] The size overhead of the email proxy is expected to be large. It ranges between $100\,\%$ and $2000\,\%$ depending on the source emails.

F. Customization

The data collection for the evaluation of web browsers and email clients can be customized by adding additional web browsers or email clients to the evaluation setup. Instructions can be found in the respective directories.