Styled to Steal: The Overlooked Attack Surface in Email Clients

Leon Trampert
leon.trampert@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Daniel Weber
daniel.weber@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Christian Rossow
rossow@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Michael Schwarz
michael.schwarz@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Abstract

Email is still a widely used communication medium, particularly in professional contexts. Standards such as OpenPGP and S/MIME offer encryption while maintaining compatibility with existing infrastructure. Within the end-to-end encryption threat model, email servers are untrusted, which creates opportunities for attackers to inject malicious HTML or CSS into encrypted emails—either live during email transport, or by re-sending leaked emails.

In this paper, we show that isolation mechanisms in widely used email client software remain inadequate. We present a novel scriptless attack that extracts arbitrary plaintext from encrypted emails using only CSS without requiring JavaScript. Once the email is opened, three benign-looking CSS features—container queries, lazy-loaded web fonts, and contextual font ligatures—map each character of the ciphertext-carried plaintext to a unique network request to the attacker's server. This attack technique can incrementally reconstruct the entire plaintext in a single rendering pass, with no JavaScript, no visual artifacts, and depending on the configuration, even without any user interaction. The technique differs considerably from prior work: it achieves complete plaintext recovery without script execution, evades state-of-the-art sanitizers such as DOMPurify, and succeeds across multiple browser engines. We demonstrate the severity of this threat on Mozilla Thunderbird and KMail, with end-to-end attacks successfully exfiltrating PGP-encrypted text from an email rendered in the latest version of the respective clients. Furthermore, we show that our technique affects code integrity tools and sanitization techniques reused in software stacks, including Meta's Code Verify. Our findings led to practical mitigations in Thunderbird, as well as a revision of Meta's threat model to include CSS. These results underline the need for robust content isolation in email client software and challenge the assumption that existing mitigations fully prevent encrypted content leakage.

CCS Concepts

• **Security and privacy** → **Web application security**; **Software security engineering**.

Keywords

Email Client; PGP; CSS; Content Exfiltration

ACM Reference Format:
Leon Trampert, Daniel Weber, Christian Rossow, and Michael Schwarz. 2025. Styled to Steal: The Overlooked Attack Surface in Email Clients. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765189

1 Introduction

Despite the widespread availability of secure end-to-end encrypted messaging applications, email remains a popular and widely used communication medium, especially in professional settings. While emails are typically transferred over TLS-encrypted connections from hop to hop, every email server involved in email delivery sees email contents in plain [13]. As a result, two popular end-to-end encryption standards, OpenPGP [3] and S/MIME [49], have emerged to protect email content. These technologies are fully backward compatible with existing email infrastructure, allowing users to send encrypted emails via any email server. In particular, inline PGP remains popular, as it allows users to send encrypted emails even when the recipient's email client does not support PGP natively [42]. This is achieved by embedding the PGP-encrypted content directly in the email body.

Within the threat model of end-to-end encryption, only the sender and recipient, with their respective email clients, are trusted parties. Importantly, no involved email server has to be trusted. The email can be intercepted and modified by malicious parties, such as email providers, ISPs, or even state actors. This allows for malicious parties to inject untrusted content into the email body, which the email client of the recipient then renders. Moreover, leaked encrypted emails can be resent to the original recipient, containing the encrypted content with additional injected untrusted content. Such untrusted content can include HTML and CSS, which are commonly used to format emails. We refer to an email containing encrypted content and untrusted parts as a *mixed-context* email.

In 2018, the Efail attack [39] demonstrated a direct content-exfiltration attack, where specifically crafted HTML injected by an attacker tricks the parser into including the decrypted PGP content as part of a URL fetched from an attacker-controlled server, leaking the content to the attacker. In response to this attack that affected at least five widely used email clients, including Mozilla Thunderbird and Apple Mail, the security community has focused on preventing direct content exfiltration attacks. However, we argue that Efail is merely an instance of a more general class of attacks that exploit the lack of isolation between untrusted and trusted content.

In this paper, we revisit the attack surface from a "CSS-only" angle and answer the following research questions: **Have the mitigations against direct content exfiltration attacks fully closed the attack surface for content exfiltration attacks in email clients? Can we still mount attacks using only HTML and CSS that undermine email encryption in a single rendering pass, i.e., when simply opening an email?**

We systematically analyze the current behavior of email clients when rendering mixed-context emails. We find that while a direct content-exfiltration attack, as in Efail, is no longer possible, at least three widely used PGP-enabled email clients (Mozilla Thunderbird, KMail, and Apple Mail with the GPGSuite plugin) still allow untrusted stylesheets to be applied to PGP-encrypted content, showing a lack of isolation. Control over stylesheets is often ignored or regarded as low-severity issues [40, 41], with existing scriptless (CSS-based) attacks being tailored to the web (i.e., browser) setting. Existing attacks primarily focus on leaking HTML attributes, such as anti-CSRF tokens or the values of input fields [17, 21], but not text. While there are some attacks that are capable of leaking text to some extent, they do not apply to the email context. Existing approaches typically rely on repeated interactions, scrollbars, or browser-specific features, making them unsuitable for universally applicable real-world scenarios, such as email client exploitation.

Thus, to answer the second research question, we introduce a novel scriptless attack using only CSS. Our proposed attack allows an attacker to exfiltrate arbitrary text from an encrypted email via the following four steps. (1) The attacker crafts a message containing the encrypted text for the recipient, combined with HTML and CSS. (2) Upon opening, the client renders the payload, i.e., decrypts the ciphertext and applies the attacker-provided stylesheet. (3) Three standard CSS features—container queries, lazy-loaded web fonts, and contextual ligatures—encode each plaintext character to a unique request for a remote image to the attacker server. (4) The loading of remote images incrementally leaks the entire plaintext during a single rendering pass. The technique neither shows visual artifacts nor triggers warnings, yet recovers arbitrary text.

At the core, we apply fonts with specifically crafted ligatures to the targeted text inspired by attacks against browsers [26, 29, 37]. If the defined ligature matches the content of the targeted text, it applies a unique width to the text, which we can measure purely in CSS. This still poses the major challenge of how to leak arbitrary content *in a single shot.* To tackle this challenge, we leverage CSS animations to repeatedly apply fonts with different ligatures without having to reload the content or open the email multiple times. Consequently, we can recover arbitrary text character-by-character and exfiltrate it via character-dependent remote-resource loading. By relying on lazy font loading, we can dynamically craft fonts with the required ligatures based on the already extracted text parts. Thus, we do not require large fonts, allowing us to stay within the practical limits of fonts.

Our attack introduces three critical innovations compared to previous variants. First, we introduce a novel mechanism leveraging CSS animations and lazy-loading fonts, enabling the incremental *extraction of arbitrary plaintext content without multiple injections or user interactions*, which is not possible with previous CSS-based attacks [18, 19, 26, 29, 37]. Second, unlike previous attacks requiring browser-specific features [18, 26, 29, 37], our approach relies exclusively on regular CSS container queries–a recently *standardized CSS feature universally supported* by all modern browser engines and multiple email clients. Third, we propose an adaptive, server-side font generation method that dynamically builds ligatures based on previously leaked characters, enabling the practical and *efficient extraction of arbitrary-length text despite inherent font limitations.* The approach differs significantly from prior scriptless techniques: it needs no script execution, achieves full plaintext recovery instead of only HTML-attribute recovery, works across multiple email clients, and can even evade popular sanitizers such as DOMPurify because the injected CSS is fully standard-compliant.

To demonstrate the security implications and severity of our attack, we conduct end-to-end attacks to fully recover the content of end-to-end encrypted emails. The victim only needs to open a single email, from which we can reliably extract the decrypted content at a rate of 2 B/s for arbitrary text and instantly for text in a known format (e.g., credit card numbers). The attack is fully stealthy, running in the background without any visual clue for the victim. Even worse, the attack is applicable to an email client that was previously unaffected by Efail's direct exfiltration attack. This highlights that the mitigations against direct content-exfiltration attacks are insufficient to prevent all types of content exfiltration.

While not our primary focus, our technique also affects state-of-the-art defenses against malicious content exfiltration in web applications. First, the recent academic proposals [14] and industry implementations [27] regarding the concept of *Accountable JavaScript* aim to vet JavaScript code. While initially targeting only JavaScript, Meta acknowledged the security impact of our attacks and extended their *Code Verify* browser extension [27, 28] to verify the integrity of JavaScript *and CSS.* Thus, our attack shows an oversight in the threat model of these defenses in that they focus only on JavaScript and not CSS, undermining the security guarantees.

Second, HTML sanitizers aim to filter untrusted user input before inserting it into the DOM [20]. For example, the popular DOMPurify [20] sanitization library can be used to prevent DOM-based XSS. Such HTML sanitization libraries do not protect against scriptless attacks in their default configuration, allowing attackers to leak web content. As such, the attack is highly relevant to sites that may not be susceptible to XSS but still allow style injection. This can be due to sanitization or a strict script-restricting CSP.

Finally, to defend against our attacks, we discuss concrete mitigation strategies. For emails, we suggest restricting remote content and strict content isolation between trusted and untrusted content. We advocate for stricter default configurations in sanitization libraries. However, we fear that this only happens when there is sufficient awareness of the severity of these novel types of attacks.

To summarize, we make the following contributions:

(1) We systematically analyze the current behavior of email clients when rendering mixed-context emails.

(2) We present a CSS-based scriptless attack that fully recovers arbitrary plaintext from encrypted emails, not just HTML attributes or short tokens.
(3) We present end-to-end exploits to recover the content of PGP-encrypted emails in Mozilla Thunderbird and KMail.
(4) We present a proof-of-concept exploit against Meta's *Code Verify* implementation of Accountable JavaScript. Meta consequently updated their implementation to also verify CSS.
(5) We showcase that the HTML Sanitizer *DOMPurify* does not mitigate our scriptless attacks in its default configuration.

Outline. Section 2 provides background. In Section 3, we introduce the threat model. Section 4 systematically analyzes email clients' rendering behavior for mixed-context emails. Section 5 presents an overview of our novel attack. Section 6 discusses its implementation. In Section 7, we present our real-world exploit on Mozilla Thunderbird. In Section 8, we discuss mitigation approaches. Section 9 demonstrates the applicability of our scriptless attack on the web. Finally, we discuss our results in Section 10.

Responsible Disclosure. We disclosed our findings regarding Thunderbird, KMail, and Apple Mail with the GPGSuite plugin to Mozilla, KDE, and GPGTools & Apple, respectively. Mozilla will issue a fix for Thunderbird in the next stable release. The vendor response from GPGTools indicates that the client is not exploitable. Meanwhile, KDE has acknowledged the issue and plans to fix it in an upcoming release. Furthermore, we have discussed the gap in the default configuration of DOMPurify and Firefox's HTML Sanitizer API with the respective maintainers. While they acknowledge the issue, they do not plan on changing the default configuration. Lastly, Meta has extended the threat model of the Code Verify extension to account for CSS as a response to our findings.

Availability. Our artifact is available on GitHub at https://github.com/cispa/stylemail. Furthermore, it is archived at Zenodo with the DOI: https://doi.org/10.5281/zenodo.17019769.

2 Background

In this section, we provide the necessary background. We provide a brief overview of end-to-end encryption in the context of emails. Additionally, introduce the modern font formats TrueType and OpenType and their relevant features.

2.1 End-to-end Encrypted Email

While emails are typically transferred over TLS-encrypted connections from hop to hop, every email server involved in email delivery sees email contents in plaintext [13]. In 1991, Phil Zimmerman invented PGP (Pretty Good Privacy) encryption, later standardized as OpenPGP by the IETF [3]. It provides cryptographic privacy and authentication to ensure that email servers (e.g., of the sender's or recipient's email provider) cannot break the confidentiality or integrity of emails. Each communication party has a private and public key. The public key is used for encryption and signature validation, and the private key is used for decrypting and signing.

Structure. Generally, emails are structured with a header and body. The body contains the message content, which can be plain text, HTML, or a combination of other types. The type of content is specified in the header using MIME (Multipurpose Internet Mail Extensions) types [45]. When an email contains multiple types of content, such as text and attachments, it uses the multipart MIME format. This format divides the email into parts, each with its own MIME-type header. A common multipart type is `multipart/mixed`, which allows for specifying independent parts of different types. A boundary string separates each part of a multipart email.

PGP in Email. There are two main techniques to include PGP-encrypted content in emails. First, with *PGP/Inline* [42], the email body directly contains the PGP-encrypted data. The body is usually of type `text/plain` and occasionally `text/html`. The approach is usually only used to encrypt text and is regularly used with clients that do not natively support PGP, as it allows for easy interoperability with plugins. An example of a third-party extension that leverages PGP/Inline is the *Mailvelope* browser extension, which enables PGP encryption, e.g., on `gmail.com`. Second, with *PGP/MIME* [42], the email body has the MIME type `multipart/encrypted`. It contains an entire email body, including, e.g., attachments, and allows for encrypting arbitrary MIME types. While PGP/MIME is preferred over PGP/Inline, it is not universally supported.

2.2 Fonts

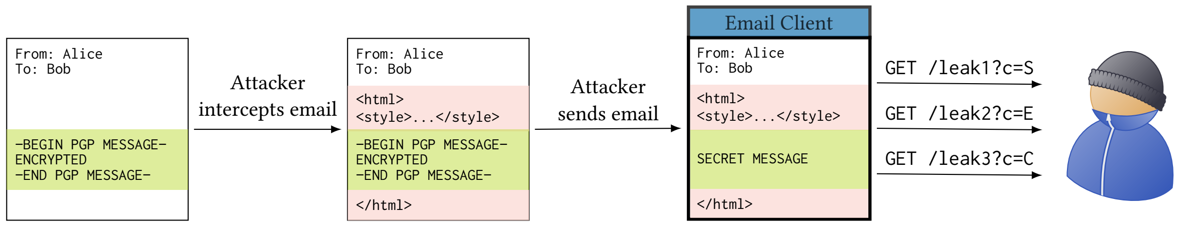
Fonts are crucial for HTML rendering in both email clients and browsers. Modern font formats, such as TrueType [2] and OpenType [30], utilize outline-based representations to map characters to visual forms. These formats define each character using mathematical descriptions of lines and curves, ensuring scalability across different sizes and resolutions. Generally, fonts are shipped as files that contain tables that map characters to their visual representation, also known as glyphs. Content providers frequently ship custom fonts to ensure a consistent visual appearance of their content. Notably, web developers may use the `@font-face` CSS rule to load custom fonts from a remote server [6].

TrueType and OpenType. TrueType is a font format initially developed by Apple and Microsoft in the late 1980s. It is widely used for both screen and print applications. OpenType is a successor of TrueType and PostScript Type 1 font formats [31]. It was introduced in 1996 by Microsoft and Adobe Systems and supports advanced typographic features, such as ligatures. TrueType has been partially extended to support OpenType features, such as ligatures. Both formats are widely used on the web and universally supported.

Font Ligatures. Ligatures map two or more characters to a single glyph [1]. In OpenType, there are different types of ligatures, such as standard, discretionary, and contextual ligatures. The former only leverages the preceding characters, while the latter two can be context-dependent with their built-in conditional logic [1]. Ligatures are often used to improve the visual appearance of text, such as combining characters that would otherwise overlap or be far apart. For example, the characters `f` and `i` are represented by a single glyph, *ﬁ*, which moves their individual representations closer together. Ligatures are crucial for many languages, such as Arabic, where the shape of a character depends on its position [57].

3 Threat Model

In our threat model, an attacker aims to recover the content of an encrypted email. We assume the attacker can access such encrypted emails (e.g., from leaked emails or as a malicious party involved in



(a) **Alice encrypts a secret message with Bob's public key.** (b) **The attacker adds a malicious stylesheet to the email.** (c) **Bob opens the email, decrypts and renders the message.** (d) **Network requests leak the decrypted content (see Figure 2).**

Figure 1: **The end-to-end workflow of our attack. The attacker obtains a PGP-encrypted email. They then add a malicious stylesheet to the email. Upon opening the email, the victim's client decrypts and renders the email. The malicious stylesheet and decrypted content are rendered in the same context, which allows for exfiltrating the content via network requests.**

sending or receiving emails). Note that in the first case, the attacker does not have control over an email server. Moreover, the attacker knows the intended recipient of the email, i.e., the victim. Without altering the encrypted block, the attacker wraps the original email inside a new HTML email, augments the message with arbitrary standard-compliant CSS, and sends this composite to the victim. We assume the victim opens the attacker's email at least once, for the email client to decrypt and render the email content. Modern email clients can decrypt and render such content automatically when the user opens the message, creating a *mixed context* in which trusted plaintext is processed together with untrusted markup.

Fundamentally, the attacker can only inject CSS and HTML, and no JavaScript. The attacker has no code execution on the victim's system, and does not rely on classic software vulnerabilities [51]. Moreover, the attacker does not exploit any bug in the client but only relies on the lack of isolation between trusted and untrusted content. Our threat model largely follows the one of prior work [39].

Scenario. Figure 1 illustrates the concrete steps for an attack. Alice writes Bob a PGP-encrypted email. An attacker who can obtain Alice's email, e.g., on any involved email server, cannot read the plaintext but modify the email before delivering it to Bob, and thus inject a malicious stylesheet into the email. Bob receives and opens the email, which is then rendered by his email client (e.g., Mozilla Thunderbird). The email client decrypts the PGP-encrypted message and renders it in the same context, i.e., document, as the malicious stylesheet. Depending on the email client and Bob's settings, this step may require Bob to press a button to decrypt the message. The attacker-controlled stylesheet is now applied to the decrypted content and can make network requests *that depend on the decrypted content.* The attacker receives those requests on their web server and can thereby infer the decrypted content. There is no visual indication for Bob that the decrypted content is leaked, and the stylesheet can present decoy content, making it indistinguishable from a regular email. The content of the stylesheet that actually leaks the decrypted content is presented in Section 5.

Note that the attacker does not have to control the email server but can also leverage emails that have been obtained by other means (e.g., data leaks) and resend them to the victim.

4 Systematic Investigation of Email Clients

In this section, we present a framework for testing the susceptibility of email clients to a lack of isolation between the decrypted and untrusted content. We systematically analyze PGP-compliant email clients. For each desktop platform, we select the most popular PGP-compliant email client and test the latest version available at the time of writing (cf. Appendix D for a table of tested clients). Each client is tested on a fresh installation with default settings.

4.1 Payload Construction and Evaluation

To test the susceptibility of an email client, we devise a broad range of test cases. Most importantly, the email client has to support PGP encryption, HTML emails, and remote content. We focus on HTML emails, as they provide the greatest attack surface and have been shown to be susceptible to a lack of isolation [39]. Furthermore, not every email client allows the same methods for including stylesheets. For each requirement of the vulnerability, we construct several emails that make use of the respective feature. Feature support is determined based on the visual rendering of the email. As an example, to test the support for web fonts, we construct emails that include web fonts and some text that is styled using the web font. We test different inclusion methods, such as inline stylesheets, remote stylesheets, and data URLs. In total, we end up with 5 test cases for this feature. Each test case is then sent to each email client, where the email is opened manually and the visual rendering is inspected. For testing the support of remote content, we use remote images that are loaded via the `<img>` tag and the `background-image` CSS property.

First, we test for the support of inline stylesheets defined via `<style>`. We test the `<link>` tag with an `https://` remote URL and a data URL in case remote content is treated differently. While the `<base>` tag cannot be used directly to include stylesheets, it can be used to redirect relative URLs of existing stylesheet inclusions to an attacker-controlled server. Furthermore, prior research has shown that CSS feature availability is inconsistent across email clients [55]. We test the support of top-level stylesheets and the availability of at-rules, such as `@font-face` and `@container` since they indicate a broader support for CSS features that can be used for attacks. To test a CSS feature, we construct an HTML email that uses the feature for each inclusion method. We also examine recursive imports via the `@import` directive [55]. Each email is then opened using the tested client. Feature support is determined based on the visual rendering of the email.

In the context of end-to-end encrypted emails, a *mixed context* refers to a scenario where encrypted and unencrypted content are present within the same email thread. It enables the exfiltration of the decrypted content using our scriptless attack. To test susceptibility, our framework leverages three PGP setups for detecting a mixed context, two targeting PGP/Inline and one PGP/MIME. The first setup directly features a body of the MIME type `text/html` that contains PGP-encrypted content. The second setup uses `Content-Type: multipart/mixed` and contains two separate parts. One part is again HTML, while the other is plaintext (i.e., MIME type `text/plain`) and contains the PGP-encrypted content. This setup is designed to target clients that block PGP/Inline in HTML, but do not account for multipart emails. The last setup targets PGP/MIME and uses `Content-Type: multipart/mixed`. One part is HTML, while the other uses `Content-Type: multipart/encrypted` with the protocol set to `application/pgp-encrypted`. Our test cases ignore the possibility of malicious HTML in the encrypted MIME structure, as the threat model would require a user to embed third-party stylesheets into their email. A mixed context is determined based on the visual rendering of the decrypted email using a custom stylesheet that applies text-altering properties to all elements using the CSS universal selector (i.e., `*`). In total, we define 5 properties that alter the visual appearance of the text drastically, where each is defined using the `!important` keyword. This ensures precedence over styles defined by the client. We leverage one test for each combination of inclusion method and setup. Our test corpus is comprised of 31 distinct test cases spanning 6 popular clients, resulting in 186 test cases. Appendix D provides more details.

4.2 Findings

The results of the email client study are shown in Table 1. Most importantly, Thunderbird, KMail, and Apple Mail, with the GPGSuite plugin, allow a mixed context in which untrusted stylesheets can be applied to decrypted content. Further, they all support the vast majority of CSS features, including at-rules and remote content. Interestingly, KMail was not susceptible to Efail [39], showing that the lack of isolation goes beyond the original attack vector. By default, Thunderbird requires a button press to allow the loading of remote content and a second button press to perform the decryption. The same button presses are required by KMail, with the addition of a third button press to enable HTML rendering. Note that this behavior is highly customizable in most clients. Thunderbird, for example, allows users to grant default permissions globally, per sender or per domain. As discussed in Section 7.1, an attacker can leverage exceptions to bypass the default remote content policy via sender spoofing. Further, blocking remote content is challenging, as also shown by prior research [39] and our investigations.

By default, Apple Mail does not require any user interaction to load remote content. However, the plugin aims to prevent the loading of remote content using the API provided by Apple Mail should a message be decrypted in a mixed context. Interestingly, we still see some remote images in a mixed context, even without user interaction. The vendor response indicates that this only affects previously cached remote content, and is thus not considered exploitable. Several users have reported issues with remote content not being blocked, at least indicating inconsistent behavior. [1]. This underlines the challenge of entirely blocking remote resources.

[1]https://mjtsai.com/blog/2024/06/07/apple-mails-broken-block-all-remote-content/

```
<body>
 <table class="moz-header-part1 moz-main-header">
     <tbody>...</tbody>
 </table>
  <link rel="stylesheet" href="data:text/css;base64,..">
 <div class="moz-text-html"><pre>DECRYPTED MESSAGE</pre></div>
</body>
```

Listing 1: A simplified version of the DOM rendered by Thunderbird after decrypting PGP/Inline. The untrusted stylesheet that enables our attack is highlighted in red.

4.3 Vulnerability Analysis

In the following, we analyze the lack of isolation with Mozilla Thunderbird as an example. Note that the same issue also applies to the other affected email clients. We discover that Mozilla Thunderbird does not correctly isolate encrypted inline PGP contexts. An HTML email with encrypted inline PGP is first rendered without performing the decryption. If the user does not have automatic decryption enabled, they are presented with a button for the decryption. After decryption, all HTML elements are removed from the existing DOM, and instead, the decrypted content is inserted. Due to reusing the same DOM, stylesheets persist and are applied to decrypted content. Listing 1 shows a simplified version of the resulting DOM structure. As such, Efail's original direct exfiltration attack is completely mitigated. However, the untrusted stylesheet still remains within the same context as the decrypted content. In essence, this setup is similar to the one of a traditional CSS exfiltration attack in the browser [18].

Limitations of Existing CSS Exfiltration Attacks. CSS exfiltration attacks are a well-known class of attacks that leverage CSS features to exfiltrate data from the DOM of websites. Prominent examples include attacks that leverage attribute selectors to exfiltrate data from HTML attributes. HTML attributes are often used to store sensitive information, such as API keys or anti-CSRF tokens [40]. In our case, however, as showcased in Listing 1, the targeted data is not stored in HTML attributes but rather in the text content of HTML elements. Here, attack techniques are sparse and often

Table 1: **Results on PGP-compliant email clients. ● shows that plaintext and untrusted styles are rendered in the same context.**

Type	Client	Plugin	Mixed Context
Cross-Platform	Thunderbird	-	●
Windows	Outlook	gpg4o	
	Outlook	gpg4win	
Linux	Evolution	-	
	KMail	-	●
macOS	Apple Mail	GPGSuite	●

have limited applicability. As an example, Heiderich et al. [18] proposed a technique that is able to exfiltrate text of HTML elements of short length, e.g., four-digit PIN codes, using scrollbar-selector-based width measurements. Since this technique basically performs a dictionary attack, it cannot be used to exfiltrate arbitrary text. Furthermore, the scrollbar selector, which is widely used by CSS exfiltration attacks [18, 26, 29, 37], is not universally supported across rendering engines. In our case, the selector is, for example, not supported by the Gecko engine used by Thunderbird.

5 Exfiltrating PGP-encrypted Emails with CSS

Since both Efail's direct exfiltration and existing CSS techniques are not capable of exfiltrating arbitrary text from HTML elements in email clients, we propose a new technique that demonstrates the feasibility of CSS-based exfiltration attacks in email clients.

In the following, we provide a high-level overview of our attack technique. In a nutshell, we recover the text (e.g., email) content of HTML elements using a combination of width measurement and repeated text rendering with specifically crafted ligatures in custom fonts. Figure 2 provides an overview of the technique. We first create font ligatures (①) that uniquely change the dimensions of the rendered text based on its first unknown character (Section 5.1). As such, the width of the text directly encodes the first unknown character of the text element. An attacker can measure these dimensions for a single ligature (②), which is then used to load a unique resource from the attacker's server (③). The attacker thereby learns the respective character (or even several characters) that are represented by the ligature (Section 5.2). Such leakage can be repeated arbitrarily often using the lazy loading of fonts combined with CSS animations (④) to recover larger contents fully deterministically (Section 5.3). In particular, the lazy-loading of our custom fonts via the animations allows the incremental construction of a known prefix where leverage the known prefix to target the next unknown character. This section introduces the general design concepts behind the attack. We provide more implementation details in Section 6.

5.1 Content-Based Font Dimensions

We first introduce a font-based technique that maps the textual content of an element to a unique width that encodes information about the content. To allow an attacker to iteratively leak a text character by character, we encode a known prefix together with guesses for the next character. Each guess has a unique width, which the attacker can infer, e.g., using container queries (Section 5.2).

When rendering text, characters or symbols are visually represented by glyphs as assigned by the font. The mapping is performed using lookup tables stored in the font file. The horizontal width of text depends on the *advance width* of the glyphs the element contains and, therefore, the font used for rendering. Using an OpenType feature called *contextual ligatures*, we may substitute a sequence of glyphs with a single glyph. By assigning a unique advance width to the substitution glyph, we can distinguish character sequences based on their width, which we can measure, e.g., using CSS container queries. The technique requires the loading of a custom font and its use for rendering. This uses the CSS directive `@font-face` and the property `font-family`, which are universally supported.

```
@Letters = [a b c d e f ... z];

feature clig {
  ignore sub @Letters s' e' a';
  sub s' e' a' by width1;
  ignore sub @Letters s' e' b';
  sub s' e' b' by width2;
  ...
} clig;
```

Listing 2: An example of contextual ligatures that map character sequences to unique widths. The `ignore sub` keyword instructs the next substitution to be ignored upon match.

5.1.1 *Using Ligatures as a Filter.*

In the following, we introduce our use of ligatures to assign a unique width to different character sequences. A contextual ligature replaces a sequence of glyphs with a single glyph (e.g., "ﬃ" instead of "ffi"). Ligatures are implemented through substitution rules defined within an OpenType font's layout tables. These rules specify which character sequences should be replaced by ligature glyphs based on contextual factors such as neighboring characters or glyph positioning.

Listing 2 shows the syntax of OpenType used to define contextual ligatures. `@Letters` is defined as the set of glyphs representing lowercase ASCII letters. Next, we define ligatures that replace a character sequence with some other glyph unless the sequence is preceded by any lowercase ASCII letter. For example, the sequence "sea" is replaced with a glyph defined as *width1*. As the glyph name suggests, we define one glyph per character sequence and use unique widths to identify the character sequence.

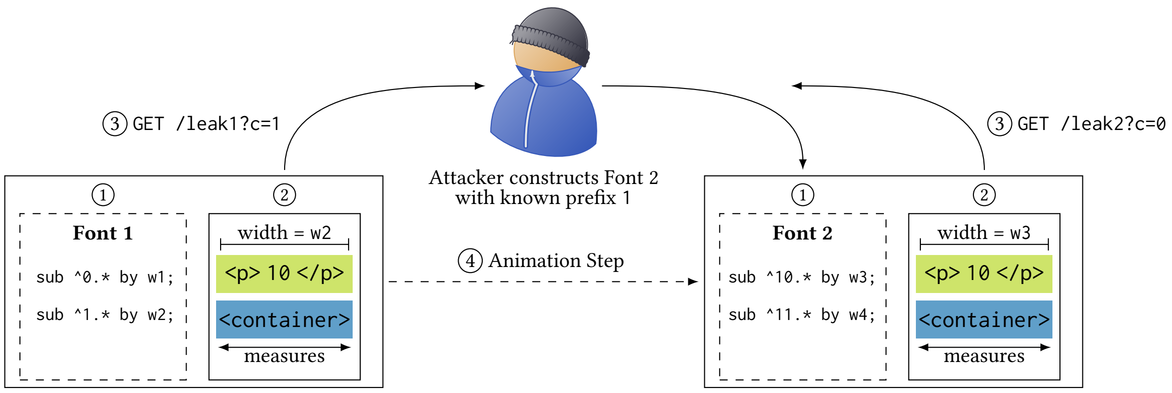
To assign a unique width to the set of possible prefixes, we first map all regular characters to glyphs that have zero width. This prevents any characters that are not part of the prefix from influencing the width of the text. Next, we create a contextual ligature that replaces the corresponding prefix with a glyph with a unique width. This is illustrated in Listing 2, where the combined sequence of the known prefix is "se", and every possible next character is replaced by a glyph with a unique width. This effectively allows us to determine the character that succeeds a known prefix.

5.1.2 *Targeting the First Glyph.*

To leak the entire text character by character using our technique, we start by targeting the first character of the text. Inherently, though, ligatures do not provide means to target the first glyph of a text. Previous work based on prefix-matching approaches did not address this problem and assumed a known prefix [26, 29]. We solve this problem by creating a contextual ligature that targets all glyphs not preceded by another glyph. For this, we leverage the `ignore sub` feature as showcased in Listing 2. It allows the definition of exceptions for the following substitution rule. We create an exception if the sequence is preceded by any other character, i.e., extending `@Letters` in the example to contain all characters. The next substitution rule can only match at the start of text. For our purposes, the charset to ASCII. It can, however, also be extended to Unicode.

5.1.3 *Practical Font Limitations.*

The number of glyphs a font can define, as well as their widths, is bounded by the OpenType standard [30]. For OpenType fonts, this limit is implicit due to the standard's use of 16-bit unsigned integers. Thus, the maximum



(a) **The attacker creates a custom font with ligatures that assign unique widths (*w1, w2*) to prefixes starting with the characters `0` and 1, respectively. All non-matching patters are assigned the width 0. The width is then measured using container queries. Loading a unique width-dependent resource now leaks the first character.**

(b) **The attacker builds Font 2 using the known prefix 1. With the potential next characters `0` and 1, the prefixes 10 and 11 are assigned unique widths (*w3, w4*). This is again measured and leaked to the attacker, revealing the second character. The attacker repeats these steps until the full secret is extracted.**

Figure 2: **A high-level overview of our attack technique. The binary string `10` serves as an example secret. We leverage font ligatures (①) that assign a unique width to text elements. For clarity, we use regex syntax for the ligatures. The element's width is measured using container queries (②), which leads to the loading of a unique, width-dependent remote resource (③). Using CSS animations and lazy font loading (④), the attacker repeats this process for each character, thus incrementally expanding the known prefix character by character. The entire process is invisible for the victim.**

number of glyphs per font is 65 535 (`0xFFFF`). The same 16-bit limit also applies to the advance widths of glyphs. Note that advance widths are defined relative to each other, so a slight difference in advance widths may not be distinguishable in every rendering context. Minimal differences may lead to the same pixel grid alignment, which prevents distinguishing these glyphs based on their width. These two factors limit the amount of information that can be exfiltrated with a single font necessitating the use of multiple fonts.

5.2 Measuring and Leaking Widths

Glyphs allow the encoding of specific character sequences as ligatures with content-specific widths. As a next step, attackers must measure and leak the content-dependent sizes. This implicitly leaks the otherwise secret content now encoded into a single glyph. To this end, attackers follow a two-step process. First, they measure the width of the glyph. Prior work has identified several methods that allow such measurements, e.g., via media queries [18, 25] and container queries [55]. Based on a specially crafted layout of HTML elements, rendered content affects container dimensions, which can be queried in pure CSS. We present more details and discuss implementation alternatives in Section 6.1. Second, attackers must learn the measurement results via a feedback channel to recover the implicitly leaked content. Given a child element of a container to which we can apply styles, we may leverage width-dependent properties and directives that trigger the loading of remote resources. We provide more implementation details in Section 6.2.

5.3 Constructing Incremental Measurements

In many cases, a malicious actor can only provide a single stylesheet for the attack, for example, in emails or when users get suspicious when a website opens several pop-ups. Thus, any realistic attack has to work "in a single shot" to reduce the attack prerequisites and user interactions. As we show in this section, attackers can use a *single* CSS file that dynamically loads fonts to leak content fully.

5.3.1 Multiple Measurements in a Single Stylesheet. Previous work relied on repeated injections (e.g., multiple popup windows) to leak character sequences [26, 29]. This is not possible for emails unless we assume the target user would re-open the email many times. Instead, our technique can overcome this limitation using CSS animations. By using a custom CSS animation that combines *multiple* measurements, we can load and apply an unlimited number of ligatures in a *single* stylesheet. We define such animations using the `@keyframes` directive that includes all different styles of an element we want to measure as animation frames. We leverage standard-compliant CSS animations without any user interaction, as animations defined via the `@keyframes` directive start automatically upon content rendering. Common email clients (e.g., , Thunderbird and KMail) and browsers do not throttle or block such animations, ensuring stable leakage across multiple repeated experiments. Each animation frame loads and applies a new, attacker-controlled remote font.[2] To this end, we define fonts using `@font-face` that are consumed in order by the animation. We can also control the animation's timing using the CSS property `animation-duration`. This way, we leverage the lazy-loading behavior of remote fonts exhibited by user agents. All major browser engines defer the loading of remote fonts until they are required for rendering. The resulting

[2]For full-text recovery, we have to load remote fonts from an attacker-controlled source. This allows for keeping state on the attacker-controlled server to dynamically create the custom fonts used for measurements. Fonts loaded from data URLs can also be leveraged to infer relevant information, such as performing a dictionary attack.

primitive thus allows us to iteratively (i) load attacker-controlled custom fonts that, one by one, are applied to the target element and change ligatures and (ii) measure the widths of the text once ligatures are applied. So, while each font is still bound by size restrictions (cf. Section 5.1.3), the *sequence* of fonts—and hence, the overall number of ligatures we can test in a realistic scenario—is unlimited. Furthermore, the animation allows us to dynamically create fonts that contain ligatures based on past leaked information. We can thus leverage this technique to load attacker-controlled fonts iteratively and thereby incrementally leak the entire text content.

5.3.2 Incremental Full-Text Leakage. We use the primitive of multiple measurements to incrementally leak the known prefix (and, hence, the text) character by character. Attackers rely on the prefix obtained by prior measurements. This prefix is then added to the substitution rules of the following font loaded from the attacker's server. We use the information obtained by prior measurements as a prefix for ligatures, such that we effectively construct a ligature chain that identifies the text of the target. For example, if the known prefix is "Dear Alic", the attacker can deliver a font with ligatures for "Dear Alica", "Dear Alicb“, "Dear Alicc", and so forth, ultimately leaking the next character and expanding the prefix.

6 Attack Implementation

In this section, we discuss different width measurement techniques that enable the attack as outlined in Section 5, and how we can relay measurements to a remote server. Furthermore, we discuss contextual improvements to the attack and discuss its limitations.

6.1 Measuring the Width of HTML Elements

In this section, we describe how to leverage the CSS-based technique of prior work [55] to measure the width of HTML elements for our attack. The technique does not use any non-standardized features or subdocuments (e.g., iframes) and is thus the first technique that can be leveraged in every standard-conforming context. Currently, the technique has only been used for fingerprinting in an attacker-controlled environment. Thus, we describe how we can apply the technique in a context where we do not control the DOM.

At its core, the technique leverages CSS container queries for querying the dimensions of container elements [5]. The setup to measure the width of an element requires three elements: the target element, one adjacent element, and a common parent element (see Figure 2). We transform the element adjacent to the target element into a container using `container-type: inline-size`. We leverage the adjacency of the elements such that a query of the container dimensions directly translates to the dimensions of the target element. For this, both elements must share a common parent element called the wrapper. We let the wrapper scale to the width of its content using `width: fit-content`. We let the container scale to the full width of its parent using `width: 100%`. Now the width of the parent and the container are equal to the width of the target element, such that a container query reports the dimensions of the target element. Using this technique, we can measure the content width of an element by setting the `width: fit-content` property of the element. Note that we cannot directly transform the target element into a container and measure its width, as the width of a container is independent of its content. Furthermore, the conditional styles inside of container queries can only be applied to children of the container. Thus, the container used in the setup must feature at least one child element. The attacker injects new or transforms existing DOM elements into the measurement setup.

Real-World Measurement Setups. As we have just described, we require a measurement setup where a container is adjacent to the target element. Such a setup can be created by transforming an existing element into a container and is thus applicable in any real-world context. As an example, the setup for Thunderbird is described in Section 7.1 (for KMail, see Appendix B). We can always propagate the target's width to its parent by setting the parent's width to `fit-content` and the `display` property of any other children to `none`.

Overwriting Existing Styles. Inherently, CSS injections conflict with stylesheets defined by the victim. CSS rules are applied according to their specificity [9]. Thus, any properties the victim defines must be overridden using more specific selectors or the `!important` keyword, which allows a rule to override more specific rules. Note that a rule defined using `!important` can be overridden by another rule with the keyword and greater specificity [9].

Alternative Width Measurement Techniques. While width measurements using CSS is not inherently new, our technique stands out because it only requires standardized CSS features, which were previously considered harmless, thus making it the first method that is not only applicable to all major browsers but also to email clients. Prior work [18] has identified two other CSS-based approaches that allow for approximating the width of elements. Those techniques leverage iframes, the `::-webkit-scrollbar:horizontal` selector and media queries. The idea is to fill an iframe with an element of a fixed width. The width of this element is the threshold above which a request to the server is issued, where the threshold is determined via media queries or the presence of a scrollbar. Similarly, Lin et al. [25] used the same technique for CSS-based fingerprinting. In general, both techniques are less flexible than our approach due to their use of iframes. The use of iframes is often restricted, e.g., in email clients [55]. Both techniques require injecting iframes adjacent to the target element. This is a much stricter requirement than the setup used by container queries, as container queries allow for existing elements to be repurposed.

6.2 Exfiltrating Measurements

On a high level, we can transform the width of HTML elements into conditional styles. However, the width measurements must be relayed to a remote attacker-controlled server that performs post-processing to recover the textual content. Given a child element of a container to which we can apply styles, we leverage various properties and directives that trigger the loading of remote resources. For example, we can use the `background-image` property to load remote images using the `url()` function.

Encoding. In email clients, each request issued by CSS is usually only performed once, and all subsequent uses of a resource are served from a cache. This even applies when cache-control headers indicate that a resource should not be cached. Since CSS does not provide a way to force the reloading of resources, each

```
@keyframes CustomAnimation {
  0.0%  { font-family: "CustomFontA"; }
  50.0% { font-family: "CustomFontB"; }
}
@font-face {
  font-family: "CustomFontA"; src: url("/font/next?it=0");
}
@font-face {
  font-family: "CustomFontB"; src: url("/font/next?it=1");
}
```

Listing 3: CSS animations can be leveraged to implement full-text leakage as described in Section 5.3. Each font leaks a character of the target element such that we can leak two characters. The fonts are applied to the element in order via the custom animation. The width measurement setup is omitted for brevity (see Appendix A).

container query can only be used once to relay a measurement. For this reason, every measurable state must map to a unique set of container queries. Moreover, all measurable states must be mutually exclusive to allow a direct recovery of the text content without any post-processing. For our purposes, we leverage one query per character at each point in the ligature chain. Assuming we target the 26 lowercase letters, we require 26 distinct container queries multiplied by the number of characters to recover. Thus, the number of container queries grows linearly with the length of the text. Outside of email clients, caching is often not an issue, which allows reusing the same container query multiple times.

Remote Images. For the exfiltration, we require the ability to load remote content. Our implementation uses the `background-image` property. In some scenarios, exfiltration may be prevented simply by blocking remote content. Examples include a strict CSP or email clients that prevent the loading of remote resources in email threads with encrypted messages. Prior research [39] and our investigations (e.g., sender spoofing, cf. Section 7.3), however, show that blocking remote content is often challenging and may constitute an orthogonal problem in email clients.

6.3 Incremental Full-Text Leakage

Listing 3 shows an example implementation of the full-text leakage introduced in Section 5.3. We omit the measurement setup and process. For completeness, the omitted parts are listed in Listing 4 in Appendix A. Our example leverages two fonts and is thus capable of leaking two characters. Each font contains a set of ligatures similar to the example in Listing 2. In our example, each font is applied to the target element for 500 ms due to the animation duration of 1 s. The second font, i.e., `CustomFontB`, is only constructed on the server after the measurement generated by the first font is received. Our server implementation is a simple Python script of about 100 lines using `fonttools` [44] and `Flask` [43].

6.4 Attack Enhancements

In this section, we describe a set of enhancements that allow for greater stealthiness and flexibility during the exploitation phase. They are, however, not required for successful exploitation.

Stealthiness. The attack can be hidden entirely from the user by limiting the visibility of the measurement setup. In particular, we can use the `visibility: hidden` property to hide the measurement setup entirely. Alternatively, we can set `opacity: 0`, use fonts without any visible glyphs, or even color the text the same as the background. To further conceal the attack, we can introduce decoy content that mimicks an actual email. This can, for example, be achieved using the `::before` and `::after` pseudo-elements with the `content` property. This property can be used to define arbitrary text that is rendered before or after an element. Ultimately, this allows the attack to be concealed in such a way that it is indistinguishable from a regular email. Depending on the email client configuration, the attack requires no user interaction in the best case, and up to three clicks in the worst case, excluding the initial email opening. Note that these clicks are also required when opening benign emails. There are no popups or other user interface elements that would indicate an attack is in progress.

Recursively Loading Stylesheets. In Chromium-based browsers, the `@import` rule is non-blocking, which allows the attack to leverage the lazy loading of stylesheets instead of only fonts [17]. This allows the attack to be split across multiple stylesheets or even to circumvent CSPs that do not allow remote fonts.

Restricting the Charset. The charset of the target text may be restricted to only lowercase or uppercase characters using the CSS directive `text-transform`. This effectively reduces the number of characters we have to take into account by 26, which allows for encoding more information in a font or minimizing its size.

Leaking Character Pairs. Furthermore, depending on the charset, we can easily leak character pairs, or even triples, instead of single characters (see Section 5.1.3). This doubles or triples the leakage rate of the attack technique.

6.5 Attack Limitations

The attack is only limited by the speed at which the client can load and apply the custom fonts. As such, the limit is determined by the client hardware and round-trip time (RTT) to the attacker server. It determines the maximum speed of the animation described in Section 5.3. Thus, we can address this limitation by delaying the start of the animation using `animation-delay` and increasing its overall duration. Any server-side computation time is negligible.

CSS Mechanisms Used. The ability to use CSS at-rules is vital to the attack technique. In particular, we leverage `@container`, `@font-face`, and `@keyframes`. For this, we require the ability to inject top-level CSS rules since at-rules may only be used at the top level. Top-level rules can be defined via the `<link>` tag, using the at-rule `@import`, or by using inline `<style>` tags. Note that style attributes do not suffice to implement the outlined attack.

7 Case Study: Breaking Email Encryption in Thunderbird

In this section, we outline the building blocks for our attacks that break the confidentiality of end-to-end encrypted emails. As introduced in Section 4.3, we operate in a scenario where the attacker can inject arbitrary top-level CSS into the context of an encrypted email. For readability, we only focus on Mozilla Thunderbird. It merely serves as a case study to demonstrate the feasibility of our attack technique. However, the attack technique is applicable to all email clients that allow the application of untrusted stylesheets to decrypted content (cf. Table 1). As analyzed in Section 4.3, this also applies to KMail. The proof-of-concept also works in KMail, but we omit the details here for brevity. In Appendix B, we discuss the attack implementation against KMail, which only requires minor adjustments in the measurement setup.

7.1 Prerequisites

In the following, we discuss the availability and requirements of the individual building blocks of the attack.

CSS Features. While Thunderbird does not allow the use of container queries in inline stylesheets, stylesheets included via the `<link>` element allow the use of most CSS features. This includes all features relevant to our attacks: container queries (i.e., `@container`), animations (i.e., `@keyframes`) and external fonts (i.e., `@font-face`).

Remote Content. The loading of remote content is required for exfiltration. While Thunderbird aims to generally prevent the loading of remote content in email threads with encrypted messages, our investigation shows this is not the case in a mixed context, enabling exfiltration. A more detailed discussion is provided in Section 7.3.

Width Measurement. To measure the width of the decrypted content (see Section 6.1), we have to inspect Thunderbird's DOM structure of mixed-context emails, as shown in Listing 1. The decrypted content is rendered in the `<pre>` element. We propagate the element's width to its parent by setting the width of the parent `<div>` to `fit-content`. Finally, we leverage the `<body>` element as the wrapper and transform the `<table>` element into a container. The `<table>` element is adjacent to the `<div>` with the width of the decrypted content, allowing us to measure the content's width.

7.2 End-to-End Content Exfiltration

In the following, we present different end-to-end attacks of decreasing complexity that allow us to recover the content of a PGP-encrypted email in Thunderbird.

7.2.1 Full-Text Recovery. First, we perform full-text recovery on encrypted emails. We combine all techniques as outlined in Section 5 and Section 6 and proceed as follows.

We start by simplifying the setup. We apply `display: none` to all elements of the DOM that are not involved in the attack. This prevents them from interfering with our measurements that are performed using the setup described in Section 7.1. We apply `text-transform: lowercase` to the `<pre>` element, which restricts the charset to lowercase ASCII characters such that we do not have to distinguish between lower- and uppercase characters.

Next, we add a custom animation to the `<pre>` element that consists of one frame per (estimated) leakage size. As the server dynamically handles font generation and gracefully concludes extraction once no further content can be identified, we ensure flexibility and robustness in practical exploitation scenarios where we do not know the exact length of the exfiltrated text. Each frame is active for 500 ms to provide sufficient time for exfiltrating the measurements and loading new fonts, even for slower connections. Upon email decryption, the animation is applied to the decrypted content, and the leakage process begins. We set the visibility of the content to invisible to ensure that the victim does not see any visual clue, such as flickering, ensuring a stealthy attack. The first font of the animation changes the target's width to identify the first character. We leak this width to the server to compute the second font, which now uses a ligature with the first character as a prefix and is lazily loaded in the second frame. This is repeated until all fonts have been loaded, i.e., all decrypted characters are leaked.

In our proof-of-concept, for demonstration purposes, we retrieve the first 128 characters of PGP-encrypted content. Note that the exploit described is easily extensible to leak more characters, but the maximum leakage size has to be determined upfront. The leakage time grows linearly with the leakage size. We thus define $n = 128$ custom fonts using the `@font-face` directive. Each font is loaded from a different URL pointing to the attacker-controlled server. The custom animation is applied to the target element, i.e., `<pre>`. The animation iterates over its n frames and applies a new custom font to the target element. To leak the widths, we define 3328 ($n * 26$ for "a"-"z") container queries. Each query identifies one ASCII character at a specific position (cf. Section 6.2). Within a query, we load a background image for a child of the container from a URL that identifies the character and position determined by the query. Using this information, the server maintains a known prefix, which is incorporated into the ligatures of the following custom font.

Evaluation. We evaluate the experiment with a remote server over 20 repetitions. In each iteration, we generate a random 128-character secret consisting of lowercase ASCII letters. We successfully leak the entire secret in 64 seconds in every repetition. The demonstrated leakage rate of approximately 2 B/s is primarily limited by network round-trip latency and rendering overhead at the client side. Under local network conditions, leakage speed increases significantly (up to several tens of bytes per second), highlighting the practicality for local adversaries or low-latency attackers.

7.2.2 4-digit PIN Recovery. As a second case study, we recover 4-digit PIN codes from an encrypted email. Since the 10 000 possible combinations are below the limit on the number of glyphs for an OpenType font (approximately 65 000), we can fully recover such a PIN with a single font and, thus, without animation. For this, we create a custom font that contains a ligature for each possible PIN. Each ligature replaces the PIN with a different glyph. Each glyph has a unique width such that measuring the container's width reveals the PIN. For the exfiltration, we require one container query per possible PIN. PIN recovery has minimal requirements, as the custom font can be included in the attack email via a data URL. Thus, only the exfiltration requires the loading of remote content. In addition, we only require one font and do not leverage animations. This translates to instantaneous and error-free exfiltration.

7.2.3 Keyword Detection. As a last case study, we perform keyword detection with similar requirements to PIN recovery. We define a set of keywords and check if an email contains at least one of those keywords. For this, we leverage a font where every glyph has a width of zero, except for one glyph, which is the substitute for the keywords. Each keyword is encoded into a ligature, which replaces the word by our non-zero-width substitute. The HTML element containing the decrypted text only has a non-zero width if it includes at least one keyword. This check only requires a single container query, allowing instantaneous and error-free exfiltration. Only the exfiltration requires the loading of remote content.

7.3 Remote Content Loading

In this section, we show that remote content blocking only partially mitigates the issue and often leads to implementation inconsistencies in practice. Since successful content exfiltration requires the loading of remote content, this section further discusses how remote content loading can be triggered in email clients. While many clients allow remote content to be loaded by default, some clients aim to prevent the loading of remote content entirely or instead require a user interaction to allow it. Prior research [39] and our investigations show that blocking remote content is often challenging and may constitute an orthogonal problem in email clients. As an example, Poddebniak et al. [39] showed that simple CSS rules that load images via the `background-image` property and the `url()` function could be used to bypass remote content blocking in 11 email clients. In addition, most clients allow users to add senders to an allowlist, which allows remote content from these senders to be loaded by default. We show that remote content blocking can be bypassed by using sender spoofing [22]. Remote content loading can be triggered by sending an email from a allowlisted sender to the target user, even if remote content loading is disabled by default. Popular guides actively recommend adding senders/domains to allowlists to ensure correct email rendering. Although precise empirical statistics of how frequently these settings are modified are challenging to obtain, the widespread recommendation by popular services strongly indicates practical viability. Moreover, Thunderbird's documentation only mentions privacy implications of loading remote content, not security risks.

8 Mitigations for Email Clients

In this section, we discuss potential mitigations to the aforementioned vulnerabilities and attacks that go beyond the currently deployed spot mitigations against the original Efail attack [39]. Preventing any of its main requirements is a practical mitigation for our attack. Email clients can either isolate encrypted message contents, prevent the mixing of encrypted and plaintext content, or block remote content. Finally, we discuss attack detection.

Isolation. Our attack requires the mixing of untrusted and encrypted content. Hence, a natural mitigation is to limit the interactions between the different content parts. This can either be achieved at the parser level, or by using traditional sandboxing techniques such as `iframes` [10]. Alternatively, an email client can disallow the mix of encrypted and unencrypted content entirely. While this restricts functionality, most non-susceptible email clients choose this approach. Although it is unclear whether this was implemented in these clients for security reasons, it prevents an attacker from applying styles to the encrypted content.

Blocking Remote Content. While the ability to apply styles to an encrypted message is sufficient to undermine its integrity [33], it is not necessarily enough to exfiltrate the content, which requires the ability to load remote resources. Thus, blocking the loading of remote content prevents exfiltration. As discussed in Section 7.3, this has to be implemented correctly. Instead of blocking remote content, clients could unconditionally fetch all remote resources of an email and directly include them via data URLs [55]. This way, the attacker does not receive requests from the victim.

Attack Detection. Due to the ability of using external stylesheets, static detection of our attack is infeasible. The loading of such external stylesheets can be deferred until after successful decryption or fingerprinting [55]. However, dynamic attack detection during the exfiltration phase is feasible. Here, the email client could monitor the loading of remote resources and styles. A high number of remote resources loaded over time, or the evaluation of a large number of container queries could indicate an ongoing attack.

9 Applicability to the Web

In this section, we show that, unsurprisingly, our new scriptless attack can also be used on the web. We introduce the threat model for web attackers (Section 9.1) and demonstrate that our attack breaks the security guarantees of Meta's Code Verify (Section 9.2), showing a gap in their threat model. In response to our research, Meta has extended the Code Verify threat model to account for scriptless attacks. Additionally, we show that popular sanitization libraries do not account for scriptless attacks (Section 9.3).

9.1 Threat Model

In the web scenario, an attacker aims to recover arbitrary text content on a website. The attacker exploits a vulnerability in the website that allows stylesheet injection, which is still possible in several settings where script-based attacks are prevented.

9.1.1 XSS Mitigations. Scriptless attacks from an alternative to XSS [40]. While they are more limited, they can circumvent security measures tailored towards detecting malicious scripts [36].

HTML Sanitizers. While most HTML sanitization libraries are highly customizable, they commonly provide default configurations. However, our investigation shows that some libraries do not account for scriptless attacks in their threat model but only focus on XSS. Both *DOMPurify* and the *HTML Sanitizer API* implementation of Firefox do not filter `<style>` tags, thus allowing scriptless attacks. The same applies to the Trusted Types API, which enforces type safety for DOM manipulation if used with such a library.

Content Security Policy (CSP). While the sources of images and stylesheets can be defined by a CSP, they are often overlooked, especially on sites that deploy policies hardened against XSS. This is underlined by the findings of prior work [47, 54] that established three main use cases of CSP in the wild: framing control (i.e., *frame-ancestors*), TLS enforcement (i.e., *block-all-mixed-content* and *upgrade-insecure-requests*) and script content restriction.

9.1.2 Script-restricting Clients. Clients can block scripting entirely [4, 32] or restrict access to certain features, e.g., the *NoScript* extension [16]. Additionally, Accountable JavaScript, the concept of auditing client-side code before execution, is increasingly gaining importance in academic research [14] and industry practices [27]. Client-side secrets could be exfiltrated by delivering malicious JavaScript at any time due to the ephemeral nature of web applications. The

Meta *Code Verify* extension shows a user if the page's scripts do not match the expected scripts [27, 28]. This can, e.g., indicate a compromised server. We argue that not considering CSS is a flaw in Meta's threat model, as it undermines all security guarantees. Our technique that allows for fully recovering text using only CSS goes unnoticed and bypasses the extension's security guarantees.

9.2 Case Study: Code Verify

In this section, we show how our scriptless attack bypasses the protection of the Meta Code Verify extension. Although Meta's Code Verify extension currently serves a niche community, its explicit threat model oversight–auditing JavaScript but ignoring CSS–is indicative of a broader, systematic gap in modern web security audits. Our CSS-based attack clearly demonstrates the inadequacy of JavaScript-only audits in protecting end-to-end encrypted content, also acknowledged by Meta and thus fixed in the current version.

Extension Design. The Code Verify extension provides a transparent audit of the client-side code of a web application [27]. It verifies the integrity of the code served to the end user. This enables the detection of parties that modify, add, or remove scripts that could exfiltrate client-side secrets. An example of such secrets is end-to-end encrypted messages in WhatsApp Web. As such, the threat model accounts for browser extensions that inject their code and a malicious server that serves code different from the regular operation. Meta has released a high-level description of their implementation of Accountable JavaScript [27]. Furthermore, the actual implementation as a browser extension is open-source and available for Chrome, Firefox, and Safari [28]. Code Verify expands on the concept of subresource integrity [12], a browser security feature that detects manipulation of resources. The extension calculates cryptographic hashes for all scripts of the site. These hashes are compared against the expected fingerprint of the code a trusted third party maintains. In the case of Meta, the trusted third party is Cloudflare. The site must deploy a CSP that prevents the use of inline scripts and eval functions and also restricts the possible sources of Web Workers. If the hashes do not match or the site has no restrictive CSP, the user is notified.

Scenario. As a proof of concept, we add our own site to the list of sites on which the extension can operate and add a script that starts the audit on our site. This script is analogous to the one used on `instagram.com`, except that the user does not have to be authenticated. We replace the trusted third party with a custom domain since there is currently no way of registering an application with Cloudflare for audits. We perform an audit of a site that has no scripts and deploys a sufficiently restrictive CSP. The site contains a secret, similar to the scenario in Section 9.3. When deploying stylesheets on the site that are not present during the initial audit, a user is still shown that the site matches the expectations of the trusted third party. We verify this by recovering the secret from the site using our scriptless attack. The user is presented with a message that the audit was successful.

9.3 Case Study: DOMPurify Bypass

In this section, we outline an end-to-end attack that allows for recovering the secret from an example web application that leverages the default configuration of DOMPurify to prevent XSS attacks. We use DOMPurify as instructed by the official documentation of the project. As our study shows, DOMPurify does not remove `<style>` tags from the input. This enables all building blocks for our attack technique. In our scenario, an attacker wants to leak a secret placed in a `<p>` tag. For a successful attack, we must first identify a part of the DOM that matches the layout described in Section 6.1. In general, we only need two adjacent elements where the container element has some child element. We provide a more sophisticated real-world example in Section 7. The script of the site takes attacker-controlled input, sanitizes it using DOMPurify, and adds it to the DOM. By default, DOMPurify aims to mitigate all script injections, such as `<script>` tags or event listeners. Thus, the application is not susceptible to XSS but still provides means to dynamically add HTML to the DOM. While a malicious actor can only insert benign HTML tags, this includes `<style>` tags that can add arbitrary styles to any element of the DOM. The actual exploitation is analogous to Thunderbird, as described in Section 7.2.1. An evaluation with a remote server over 20 repetitions successfully recovers the secret ($n = 128$) in Chromium, Firefox, and Safari.

Other HTML Sanitizers. We analyzed the top 10 most popular HTML sanitizers on GitHub. Popularity is determined by the number of GitHub stars since prior research has shown that the metric correlates with deployment metrics in the wild [24]. The selection of libraries is shown in Table 2 in Appendix C. Firefox's implementation of the HTML Sanitizer API allows both `<style>` tags and even the inclusion of remote stylesheets via the `<link>` element. We find that DOMPurify and Firefox do not mitigate our attack in their default configuration. This is likely due to the fact that both libraries are primarily designed to prevent XSS attacks. The maintainers acknowledged our attack and confirmed that CSS injections are not part of their default threat model.

9.4 Mitigations

CSS injection vulnerabilities are inherently related to XSS. Thus, many existing solutions for mitigating XSS vulnerabilities also apply here. However, as showcased by our study of HTML sanitization libraries, not all solutions may account for CSS injections by default.

Sanitization. Naturally, the first step in preventing the injection of malicious code is using appropriate sanitization mechanisms [20]. However, current practices are biased towards JavaScript, often ignoring stylesheets that enable scriptless attacks. While all investigated HTML sanitization libraries provide means to remove stylesheets from untrusted input, not all of them do so in their default configuration. Thus, developers must expand on the default configurations to account for scriptless attacks.

Isolation. If feature-rich stylesheets are supposed to be controlled by users or third parties, they can be isolated using different methods. First, user-controlled stylesheets can be isolated by leveraging subdocuments (e.g., `iframes`) [10]. Second, *namespacing* is a technique usually employed to avoid conflicts between stylesheets [11, 46], where identifiers are prefixed such that they do not collide with those of existing stylesheets [46]. Further, at-rules and selectors may only be used in top-level stylesheets [6, 8], such that only allowing `style` attributes prevents most known scriptless attacks.

Content Security Policy. As a second line of defense, websites can deploy a CSP [56]. A CSP defines an allowlist of resources a user agent can load for a site. A policy with the directives `default-src` or `style-src` can restrict the loading stylesheets. A policy that prevents the loading of remote resources also prevents exfiltration.

10 Related Work

In this section, we discuss prior work on non-cryptographic attacks on email encryption and scriptless attacks on the web.

10.1 Non-cryptographic Attacks on Emails

Prior research on OpenPGP- and S/MIME-compliant email clients investigated the exfiltration of encrypted content [35, 39] and the misrepresentation of signed content [33]. Poddebniak et al. [39] found that various email clients do not isolate multiple MIME parts of an email but instead render them in the same HTML document. Their attack, "Efail", leveraged that an encrypted message wrapped in two adjacent HTML parts would lead to the decrypted content being treated as part of the same HTML document. This enabled direct exfiltration of the entire text to an attacker server by placing the decrypted content in the place of a `src` attribute of an `<img>` tag. Their research additionally highlighted ways of loading remote content without consent. Their work led to several mitigations, from blocking remote content to proper isolation. We show that there are still shortcomings of existing mitigations in post-Efail clients. In particular, while direct exfiltration as induced by the parser is mitigated, decrypted content may still be mixed with untrusted stylesheets and thus be subject to our attack.

Müller et al. [33] found that several OpenPGP- and S/MIME-compliant clients allowed the application of untrusted stylesheets to signed content, thus providing means to spoof signed messages. In addition, they showed how users could be tricked into signing responses to emails where the content was misrepresented using stylesheets [34]. Furthermore, Müller et al. [35] showcased critical flaws in the implementation of OpenPGP- and S/MIME-compliant email clients that allowed the remote deployment of keys to a communciation partner or the exfiltration of a communication partner's key. They additionally showed that some email clients could be tricked into signing or decrypting arbitrary messages to the drafts folder of the victim's IMAP server via malicious `mailto` links combined with auto save.

10.2 Scriptless Attacks on the Web

Existing "Blind CSS Exfiltration" [17, 18, 21] can exfiltrate the value of HTML *attributes* using attribute selectors but not an element's content. Heiderich et al. [18, 19] introduced a scriptless attack to detect the occurrence of a set of words but not for generic text recovery. They leverage iframe-based width measurements combined with ligatures to perform dictionary attacks. However, the techniques are not widely available since, e.g., iframes are generally unavailable in email clients [55]. Similarly, scrollbar selectors are only available in WebKit-based user agents, such as Chromium or Safari [7, 58]. Crucially, the described techniques cannot be leveraged to recover arbitrary content due to the limits on the number of ligatures that hinder dictionary attacks. Building on this, Bentkowski [26, 29] published a method to exfiltrate arbitrary text using the technique by Heiderich et al. [18, 19] with repeated injections. The method maintains a prefix of known text as a ligature between injections. However, the requirement for repeated injections and the dependency on non-standard features makes the attack inapplicable to many real-world scenarios, such as attacks on email clients. Using the `unicode-range` property, fonts can be loaded on demand if a character matching that range is present in the text [23]. This allows for leaking the charset of the text but not the text itself. In particular, the technique does not preserve the order of the characters or their frequency. Another technique detected the presence of text via the Chrome feature "Scroll to Text Fragment", which enables automatic scrolling to and highlighting of text defined in the URL fragment [38, 48]. The presence of text can be determined by applying styles to the highlight effect. The feature does, however, not provide regex-like functionality, making it infeasible to recover arbitrary text. In addition, the feature requires the user to interact with the page [48]. Scriptless attacks have also been used to perform privacy-infringing attacks from the field of XS-Leaks [15, 52]. Shusterman et al. [50] demonstrated microarchitectural attacks via CSS, and Trampert et al. [53] demonstrated them using fonts.

11 Conclusion

Our paper introduced a novel scriptless attack that extracts complete PGP-encrypted plaintext using only standard-compliant CSS, without JavaScript, visual artifacts, or complex user interaction. We reveal that multiple widely used PGP-enabled email clients fail to isolate encrypted content from untrusted styles, leaving them vulnerable to rendering-based exfiltration. Our attack leverages three benign CSS features: container queries, lazy-loaded web fonts, and contextual font ligatures. It circumvents the limitations of prior scriptless attacks, being able to exfiltrate arbitrary text fully, and is universally applicable to all modern rendering engines. In Mozilla Thunderbird and KMail, we demonstrated the effectiveness of our attack by presenting end-to-end proof-of-concept exploits for recovering the plaintext of PGP-encrypted emails. With an investigation of the most prominent HTML sanitization libraries and Meta's Code Verify auditing mechanism, we showed that current security practices are biased towards JavaScript and ignore the increasing capabilities of HTML and CSS, as demonstrated by our attack. In particular, we showed that the default configurations of popular HTML sanitization libraries do not account for scriptless attacks, allowing attackers to exfiltrate arbitrary text using our technique. Our work highlights the underestimated potency of scriptless attacks and the resulting need for broader mitigation awareness.

Acknowledgments

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 Lukas Gerlach and Simon Schwarz for helping with some experiments and Ben Stock for his valuable feedback on the paper. We further thank the Saarbrücken Graduate School of Computer Science for their funding and support.

References

[1] Adobe. 2024. Syntax for OpenType features in CSS. https://helpx.adobe.com/fonts/using/open-type-syntax.html Retrieved 2024-04-24.
[2] Apple. 2024. TrueType Reference Manual. https://developer.apple.com/fonts/TrueType-Reference-Manual/ Retrieved 2024-04-24.
[3] Derek Atkins, William Stallings, and Philip Zimmermann. 1996. RFC1991: PGP message exchange formats. https://datatracker.ietf.org/doc/html/rfc1991
[4] Chrome for Developers. 2019. Disable JavaScript. https://developer.chrome.com/docs/devtools/javascript/disable
[5] World Wide Web Consortium. 2022. CSS Containment Module Level 3. https://www.w3.org/TR/css-contain-3/
[6] MDN Web Docs. 2023. CSS at-rules. https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule
[7] MDN Web Docs. 2024. ::-webkit-scrollbar. https://developer.mozilla.org/en-US/docs/Web/CSS/::-webkit-scrollbar
[8] MDN Web Docs. 2024. CSS selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_selectors
[9] MDN Web Docs. 2024. CSS specificity. https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity
[10] MDN Web Docs. 2024. <iframe>: The Inline Frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe
[11] MDN Web Docs. 2024. Namespace. https://developer.mozilla.org/en-US/docs/Glossary/Namespace
[12] MDN Web Docs. 2025. Subresource Integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
[13] Electronic Frontier Foundation (EFF). 2018. Announcing STARTTLS Everywhere: Securing Hop-to-Hop Email Delivery. https://www.eff.org/deeplinks/2018/06/announcing-starttls-everywhere-securing-hop-hop-email-delivery
[14] Ilkan Esiyok, Pascal Berrang, Katriel Cohn-Gordon, and Robert Künnemann. 2023. Accountable Javascript Code Delivery. In *NDSS*.
[15] Nethanel Gelernter and Amir Herzberg. 2015. Cross-Site Search Attacks. In *CCS*.
[16] Giorgio Maone. 2017. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! https://noscript.net
[17] HackTricks. 2024. CSS Injection. https://book.hacktricks.xyz/pentesting-web/xs-search/css-injection
[18] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless attacks: stealing the pie without touching the sill. In *CCS'12*.
[19] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2014. Scriptless attacks: Stealing more pie without touching the sill. *Journal of Computer Security* (2014).
[20] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. Dompurify: Client-side protection against xss and markup injection. In *ESORICS*.
[21] Heyes, Gareth. 2023. Blind CSS Exfiltration: exfiltrate unknown web pages. https://portswigger.net/research/blind-css-exfiltration
[22] Hang Hu and Gang Wang. 2018. End-to-End Measurements of Email Spoofing Attacks. In *USENIX*.
[23] huli.tw. 2022. Stealing Data with CSS - CSS Injection (Part 2). https://blog.huli.tw/2022/09/29/en/css-injection-2/
[24] Simon Koch, David Klein, and Martin Johns. 2024. The Fault in Our Stars: An Analysis of GitHub Stars as an Importance Metric for Web Source Code. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*.
[25] Xu Lin, Frederico Araujo, Teryl Taylor, Jiyong Jang, and Jason Polakis. 2023. Fashion Faux Pas: Implicit Stylistic Fingerprints for Bypassing Browsers' Anti-Fingerprinting Defenses. In *IEEE S&P*.
[26] Masato Kinugawa. 2021. Data Exfiltration via CSS + SVG Font. https://mksben.l0.cm/2021/11/css-exfiltration-svg-font.html
[27] Meta. 2022. Code Verify: An open source browser extension for verifying code authenticity on the web. https://engineering.fb.com/2022/03/10/security/code-verify/
[28] Meta. 2022. Code Verify on GitHub. https://github.com/facebookincubator/meta-code-verify
[29] Michał Bentkowski. 2017. Stealing Data in Great style - How to Use CSS to Attack Web Application. https://research.securitum.com/stealing-data-in-great-style-how-to-use-css-to-attack-web-application/
[30] Microsoft. 2024. OpenType Font Specification. https://learn.microsoft.com/en-us/typography/opentype/spec/ Retrieved 2024-04-24.
[31] Microsoft. 2024. OpenType Overview. https://learn.microsoft.com/en-us/typography/opentype/ Retrieved 2024-04-24.
[32] Mozilla. 2024. JavaScript settings and preferences for interactive web pages. https://support.mozilla.org/en-US/kb/javascript-settings-for-interactive-web-pages
[33] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Hanno Bock, Sebastian Schinzel, Juraj Somorovsky, and Jörg Schwenk. 2019. Johnny you are fired!"–spoofing OpenPGP and S/MIME signatures in Emails. In *USENIX*.
[34] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel, and Jörg Schwenk. 2019. Re: What's Up Johnny? Covert Content Attacks on Email End-to-End Encryption. In *ACNS* .
[35] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel, and Jörg Schwenk. 2020. Mailto: Me your secrets. on bugs and features in email end-to-end encryption. In *IEEE Conference on Communications and Network Security (CNS)*.
[36] OWASP. 2024. XSS Filter Evasion Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
[37] Pepe Vila. 2024. Charset Leakage Demo. https://demo.vwzq.net/css2.html
[38] Maciej Piechota. 2022. New technique of stealing data using CSS and Scroll-to-Text Fragment feature. https://www.secforce.com/blog/new-technique-of-stealing-data-using-css-and-scroll-to-text-fragment-feature/
[39] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. 2018. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security*.
[40] PortSwigger. 2024. CSS injection (reflected). https://portswigger.net/kb/issues/00501300_css-injection-reflected
[41] PortSwigger. 2024. Path-relative style sheet import. https://portswigger.net/kb/issues/00200328_path-relative-style-sheet-import
[42] Proton. 2024. What are PGP/MIME and PGP/Inline? https://proton.me/support/pgp-mime-pgp-inline
[43] Python Package Index (pypi). 2024. Flask. https://pypi.org/project/Flask/
[44] Python Package Index (pypi). 2024. fonttools. https://pypi.org/project/fonttools/
[45] Pete Resnick. 2008. RFC5322: Internet Message Format. https://datatracker.ietf.org/doc/html/rfc5322
[46] Harry Roberts. 2015. More Transparent UI Code with Namespaces. https://csswizardry.com/2015/03/more-transparent-ui-code-with-namespaces/
[47] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*.
[48] Matthew Savage. 2020. PlaidCTF 2020: Catalog Writeup. https://dttw.tech/posts/B19RXWzYL
[49] Jim Schaad, Blake C. Ramsdell, and Sean Turner. 2019. RFC8551: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification. https://datatracker.ietf.org/doc/html/rfc8551
[50] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*.
[51] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.
[52] terjanq. 2023. exploit.js - CTF Challenge Solution using CSS-based XS-Search Attack. https://gist.github.com/terjanq/33bbb8828839994c848c3b76c1ac67b1
[53] Leon Trampert and Michael Schwarz. 2025. Hidden in Plain Sight: Scriptless Microarchitectural Attacks via TrueType Font Hinting. In *uASC*.
[54] Leon Trampert, Ben Stock, and Sebastian Roth. 2023. Honey, I Cached our Security Tokens - Re-usage of Security Tokens in the Wild. In *RAID*.
[55] Leon Trampert, Daniel Weber, Lukas Gerlach, Christian Rossow, and Michael Schwarz. 2025. Cascading Spy Sheets: Exploiting the Complexity of Modern CSS for Email and Browser Fingerprinting. In *NDSS*.
[56] W3C. 2024. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/
[57] W3C Arabic Script Language Enablement Community. 2024. Arabic and Persian Layout Requirements. https://www.w3.org/TR/alreq/
[58] W3C CSS Working Group. 2024. CSS Scrollbars Styling Module Level 1. https://drafts.csswg.org/css-scrollbars/

A Minimal Example: Container Queries

Listing 4 shows the measurement setup that was ommited from the example in Listing 3. It shows the animation that can be leveraged to leak two characters of unknown text using our technique. If we restrict ourselves to the characters “0” and “1”, we only require four container queries to identify the next character of the unknown text. In the first frame of the animation, we apply a font that either triggers the first or second container query. This information is transmitted to the server via the loading of the corresponding background image and leveraged in the font that is applied in the second frame of the animation. Here, the ligatures of the font are prefixed with the leaked character. Finally, the font is applied, and the width of the target matches either the third or fourth container query. Note that our queries check for width ranges, since our investigation has shown that exact floating point width comparison behaves inconsistently across user agents.

```
.wrapper { width: fit-content; }
#target {
  width: fit-content;
  font-size: 160px;
}
.container { container-type: inline-size; }

@container (width > 0px) {
* { background-image: url("/leak/0?i=0"); }
}
@container (0.4px < width) and (width < 0.8px) {
* { background-image: url("/leak/1?i=0"); }
}
@container (0.8px < width) and (width < 1.1px) {
* { background-image: url("/leak/0?i=1"); }
}
@container (width > 1.1px) {
* { background-image: url("/leak/1?i=1"); }
}
```

Listing 4: The measurement setup using the technique by prior work [55], which completes the example in Listing 3. The charset is restricted to "0" and "1" for illustrative purposes, such that we require only four container queries.

B Case Study: KMail

The attack implementation against KMail is analoguous to the one against Thunderbird discussed in Section 7.2.1, except for the measurement setup. We confirmed that the exploit works in KMail 6.0.2 which was the latest version at the time of writing. Listing 5 shows a simplified part of the DOM used by the KMail client when rendering a mixed-context email for inline PGP. We can construct the width measurement setup for container queries as discussed in Section 6.1. For this, we transform the `<tbody>` element into the wrapper. We propagate the width of the decrypted content to the `<tr>` with class `encrB` using `width: fit-content`. Finally, we transform an adjacent `<tr>` element into a container and can measure the width of the decrypted content via container queries.

```
<div>
  <table class="encr">
    <tbody>
      <tr class="encrH">...</tr>
      <tr class="encrB">
        <td><div><div>DECRYPTED CONTENT</div></div></td>
      </tr>
      <tr class="encrH">...</tr>
    </tbody>
  </table>
</div>
```

Listing 5: A simplified part of the DOM as rendered by the KMail email client in a mixed context. Attacker-controlled stylesheets are included above the document.

C HTML Sanitization Libraries

Table 2 shows versions and usage statistics as provided by GitHub for the HTML sanitization libraries in our study (Section 4).

Table 2: **The versions of the HTML sanitization libraries used in our study and their usage stats as provided by GitHub.**

Library	Version	Stars	Used By
DOMPurify	3.0.11	12,700	292,000
XSS	1.0.15	5,100	n/a
bluemonday	1.0.26	3,000	12,300
Bleach	6.1.0	2,600	306,000
sanitize	6.1.0	2,000	10,000
HtmlSanitizer	8.0.843	1,500	3,100
loofah	2.22.0	920	1,700,000
OWASP Java HTML Sanitizer	20240325.1	813	3,000
insane	2.6.2	438	6,300
html-sanitizer	1.5.0	388	n/a
HTML Sanitizer API	124.0.2	-	-

D PGP Email Clients

Table 3 shows the versions of the PGP-compliant email clients used in our study (Section 4). It additionally shows the versions of the plugins that enable the PGP functionality. All clients of Table 3 that are not listed in Table 1 are **not** susceptible to our attack. Note that we had to exclude some clients listed on `openpgp.org` due to severe functionality issues or unavailability.

Table 3: **The versions of the PGP-compliant email clients used in our study featuring their respective PGP plugins.**

Type	Client	Version	Plugin
Windows	eM Client	9.2.2157	-
	The Bat!	11.1	-
	Outlook	2404 (Classic)	gpg4o
	Outlook	2404 (Classic)	gpg4win
	Postbox	7.0.60	Enigmail
Linux	Claws Mail	3.17.5	-
	Thunderbird	115.9	-
	Mutt	9.4.0	-
	Evolution	3.44.4-0ubuntu2	-
	KMail	6.0.2 (24.02.2)	-
macOS	Apple Mail	16.0 (3774.300.61.1.2)	GPGSuite 2.0 (1827)
	Canary Mail	4.48 (1612)	-
Android	FairEmail	1.2168a	OpenKeychain 6.0.4
	K-9 Mail	6.802	OpenKeychain 6.0.4
iOS	Canary Mail	4.47 (1506)	-
	FlowCrypt	0.6.0	-
Browser Extension	FlowCrypt	8.5.4 (Chrome)	-
	Mailvelope	5.1.2 (Chrome)	-
	Psono	3.0.9 (Chrome)	-
Webmail	ProtonMail	Webmail	