Trust on Reload: Securing Browser-Based End-to-End Encryption

Simon Schwarz
Max Planck Institute for Informatics
Saarland Informatics Campus
Saarbrücken, Germany
sschwarz@mpi-inf.mpg.de

Florian Bauckholt
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
florian.bauckholt@cispa.de

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

Abstract

End-to-end encryption (E2EE) has become widely adopted in messaging applications, cloud storage, and password managers. Fundamentally, such applications require secrets, e.g., decryption keys, that must never leave the client to achieve their security guarantees. Native applications are typically packaged and distributed in a way that allows them to be effectively audited before installation. In contrast, client-side Web application code is distributed ephemerally by Web servers. A malicious, coerced, or compromised Web server can, at any time, deliver code that exfiltrates client-side secrets without leaving traces. Thus, without a mechanism for verifying the integrity of client-side Web applications and blocking unexpected or malicious client-side code before it is executed, true E2EE cannot be implemented in Web applications.

We survey five popular and representative Web application integrity verification tools and find that current solutions only partially address real-world threats. During our analysis, we identified several common pitfalls in existing tools that undermine their security guarantees. For each identified pitfall, we present concrete attack scenarios. Overall, we discovered and responsibly disclosed over a dozen bugs that allowed secret exfiltration across all tools.

Building on these insights, we propose a novel design for client-side code integrity verification that systematically improves the state-of-the-art. Our proof-of-concept extension exhaustively verifies a client-side Web application while blocking modified resources, thereby preserving the guarantees of E2EE. We demonstrate the viability and practicality of our approach through integration with real-world Web applications built using various frameworks.

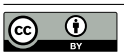
CCS Concepts

• **Security and privacy → Web application security**.

Keywords

Web Application Security, End-to-End Encryption, Code Integrity

ACM Reference Format:
Simon Schwarz, Florian Bauckholt, and Leon Trampert. 2026. Trust on Reload: Securing Browser-Based End-to-End Encryption. In *Proceedings of the ACM Web Conference 2026 (WWW '26), April 13–17, 2026, Dubai, United Arab Emirates*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3774904.3792624



WWW '26, Dubai, United Arab Emirates

ACM ISBN 979-8-4007-2307-0/2026/04
https://doi.org/10.1145/3774904.3792624

Resource Availability:
CodeSeal is available at https://doi.org/10.5281/zenodo.18314630 and https://github.com/cispa/codeseal.

1 Introduction

Over the years, the Web has evolved from a distribution system for documents to a central application platform. Using a wide variety of Web applications, including Webmail and entire office suites, users can accomplish many tasks purely from the browser. Meanwhile, end-to-end encryption (E2EE) has been widely adopted across various applications. Most prominently, E2EE is used in private messaging applications such as WhatsApp, Signal or Element, where E2EE guarantees that only the sender and recipient can read messages. More generally, E2EE offers confidentiality, integrity and authenticity guarantees even when communicating via untrusted infrastructure. E2EE has also been adopted by many other types of applications, such as password managers, e.g., Bitwarden, cloud storage, e.g., ProtonDrive, and private note-taking applications, e.g., CryptPad. Most of these applications are available on all platforms, ranging from native to Web applications accessed via the browser.

Generally, E2EE is implemented using a secret cryptographic key that never leaves the client and is inaccessible to the server [31]. In the Web application setting, this is usually achieved by storing the secret in the browser's client-side storage (e.g., local storage, IndexedDB) [10], the URL fragment [14], or by deriving a secret from user input. The client-side application code reads the secret from storage, performs the decryption, and then renders the plaintext into the page. The security guarantees of E2EE can only be achieved if the secret never leaves the client [31]. This puts trust in the client-side code, which must itself be delivered securely to uphold these guarantees.

While releases of native applications are commonly cryptographically signed [5, 42, 45] by their developers and vetted by their respective distribution platforms, e.g., Google Play Store [20], or the community, e.g., the Signal APK distribution page [46], verifying the integrity of Web applications is fundamentally more challenging. This is mainly due to the fact that a Web application is distributed ephemerally by a Web server. Although browsers may employ caching, each visit still allows the server to change delivered resources when the application's page is loaded [11]. On the Web, a server can distribute a different version of a Web application to a user without them noticing [18, 43]. In particular, a malicious server could, at any time, distribute a version of the application that exfiltrates the client secret and, thus, completely compromises the security guarantees of E2EE. Such an attempt would likely go unnoticed, as it would leave no permanent traces. Even worse, a malicious server can distribute harmful code selectively to specific users using sophisticated server- and client-side

browser fingerprinting techniques [32, 33, 52], thus enabling the server to obscure malicious behavior. This is not merely a hypothetical scenario: In 2013, Lavabit, a Webmail provider offering E2EE, was coerced by law enforcement requiring them to share their TLS private keys [48]. The operation was intended to specifically target Edward Snowden [59], a prominent whistleblower and user of the service. Without additional measures, E2EE users on the Web are at the mercy of whoever controls the server that distributes the Web application. Even worse, some applications like Bitwarden are developed by a single organization but hosted on many, potentially untrusted, instances, meaning that even if one trusts the developer, the hoster may still be untrustworthy. Thus, as of today, Web applications cannot achieve the security guarantees offered by E2EE, as opposed to their native counterparts.

To mitigate the risks associated with E2EE on the Web, there have been academic [18, 29, 38, 43], industry [40] and open-source [3, 22] proposals to verify the client-side code of Web applications. These approaches leverage browser extensions to verify the code delivered by the server on every page reload. Usually, this involves comparing the delivered code to a known-good version, e.g., by verifying a cryptographic hash or signature.

In this work, we ask the following research question: **Do current proposals fully protect the security guarantees of E2EE as implemented in real-world Web applications?**

To answer our research question, we first systematically analyze the implementation patterns of E2EE in real-world Web applications and identify all relevant locations where client-side secrets are present. The implementation patterns dictate the security guarantees and scope a verification approach must cover.

We survey the designs and implementations of existing verification approaches for their practical security guarantees given the needs of real-world applications. We find that all prior approaches have significant limitations that prevent them from effectively protecting E2EE on the Web, many of which are inherent to the chosen designs. For example, the majority of existing approaches focus on detecting code modifications and notifying the user, rather than preventing the execution of malicious code in the first place. As such, they are inherently reactive and cannot prevent secret exfiltration. Moreover, some existing approaches choose to focus e.g., only on verifying JavaScript code. This fundamentally fails to account for scriptless [23, 24, 53] and UI-based [25, 47] attacks that do not require JavaScript execution at all. Furthermore, existing approaches only focus on preventing the exfiltration of client-side secrets while largely ignoring the integrity of other resources, such as fonts. While this does not directly affect confidentiality, it breaches message integrity and authenticity, enabling a vast array of social engineering attacks. Other verification problems stem from the complexity of modern browsers and Web applications. For example, some approaches do not verify behavior-altering HTTP headers, or fail to account for persistent JavaScript service workers. In total, we have identified nine common pitfalls and reported over a dozen separate vulnerabilities in existing tools that often completely undermine their security guarantees.

Based on our observations, we propose a verification approach that ensures confidentiality, integrity and authenticity while addressing the identified pitfalls. We demonstrate its viability by implementing a proof-of-concept browser extension. Our extension, CodeSeal, proactively verifies the entire delivered page, including all static remote resources, thus explicitly accounting for scriptless and UI-based attacks. CodeSeal proactively blocks unexpected content upon page load before execution. Our implementation leverages modern browser security mechanisms, such as Subresource Integrity (SRI) and a strict Content Security Policy (CSP). This drastically reduces the complexity of our proof-of-concept implementation, and significantly reduces the risk of implementation errors. To the best of our knowledge, CodeSeal is the first extension to systematically verify all client-side resources before use. We evaluate CodeSeal on various custom and real-world Web applications showing the holistic security guarantees and negligible performance overhead of our proposal.

Contributions. We summarize our contributions as follows.

- We analyze E2EE in the context of Web applications and identify all relevant threats to E2EE as implemented in popular real-world applications.
- We categorize prior verification approaches, and identify several common design and implementation pitfalls that completely undermine their security guarantees.
- We propose a proof-of-concept browser extension, CodeSeal, that systematically improves the state-of-the-art of integrity verification of client-side Web application code.

Responsible Disclosure. We have identified 18 security-relevant bugs in existing tools. All findings have been responsibly disclosed to the respective maintainers.

2 Background

In this section, we provide relevant background information.

Browser Extensions. Browser extensions can modify the behavior of a browser and consist of two main components: a background script and content scripts [9]. The background script runs in the background and has access to special browser APIs, such as the ability to modify network traffic, or to store data locally. Meanwhile, content scripts run in the context of the page and can modify the DOM, as well as access a page's JavaScript environment. Concrete extension capabilities differ by browser engine [1, 21].

Data Separation in Web Apps. Client-side Web applications are generally built using a combination of HTML, CSS, and JavaScript (JS). Together they constitute the client-side *code*. Usually, the code is the same for all users of the application. In contrast, user-specific content of a Web application, such as messages or custom images, is referred to as (dynamic) *data*. Strict separation of code and data in Web applications has several advantages [6, 17, 27]. Modern frameworks, such as React, feature a strict separation of code and data. Moreover, many Web applications are implemented as single-page applications (SPAs), which consist of a static HTML page that loads data dynamically, hence featuring data separation by design.

Client-Side Code Integrity. Several approaches have been proposed to verify client-side Web application code [3, 18, 22, 29, 38, 40, 43]. All approaches are implemented as browser extensions and require the separation of code and data. Upon page load, they compare the delivered code to a known-good version, usually using cryptographic hashes or signatures. Ultimately, they aim to prevent the execution of untrusted code on the client side. We will detail existing approaches in Section 4.1.

Browser Security Mechanisms. Modern browsers offer several security mechanisms that allow for restricting client-side code.

Subresource Integrity. Subresource Integrity (SRI) [13] allows a Web application to specify cryptographic hashes for externally loaded JS and CSS files. When the browser loads an SRI-protected resource, it verifies its integrity using the hash. Upon failure, the resource is discarded. However, SRI checks cannot be used to verify the primary (HTML) document. The `Integrity-Policy` header [12] can be used to enforce SRI for all external scripts and stylesheets of a page. When present, the browser blocks external scripts and stylesheets that do not have an SRI attribute.

Content Security Policy. The Content Security Policy (CSP) [55] allows a Web application to specify which resources are allowed to be loaded and executed. It was originally designed to mitigate Cross-Site Scripting (XSS) attacks by restricting the sources from which scripts can be loaded. It is usually declared by adding a `Content-Security-Policy` header to the HTTP response. It is universally supported and evolved to cover all remote resources, including scripts, stylesheets, images, fonts, and more. The CSP can be used to prevent the execution of inline scripts and styles, as well as the use of `eval()` and similar functions.

3 Security Model & Threats

In the following section, we outline the assumptions of this work. We define the goals of E2EE as well as concrete threats to E2EE in the Web setting. Finally, we give a basic illustrative attack scenario.

3.1 Assumptions

We assume a standard Web application setting, where a user accesses a TLS-encrypted [44] Web application via their browser. The user's system and browser are secure and have not been compromised. This also includes the absence of malicious or insecure browser extensions [1]. The user has installed a client-side code integrity verification browser extension.

The Web application implements E2EE, e.g., for messaging, password management, or note taking. Hereby, the developer is assumed to be benign and not collude with the attacker, i.e., the developer does not intentionally include backdoors or vulnerabilities in the code. We consider inadvertent vulnerabilities that would allow the attacker to exfiltrate client secrets, e.g., XSS [4] out of scope of this work. The developer provides a hash or signature that serves as the ground truth for the verification approach. They also implement the application in conformance with the requirements of the approach. This notably includes separating code and data.

The Web application is served by an untrusted server. The server operator can be malicious, compromised by a third party, or coerced by law enforcement. Even if the Web server itself is benign, its responses might be modified, for example, by a compromised CDN, or in a "man-in-the-middle" attack with a compromised SSL certificate [26]. We refer to the actor controlling the responses as "the attacker". This attacker is able to serve arbitrary responses to the user, including malicious page content, headers and other resources.

3.2 Threats to E2EE

E2EE aims to provide three main security guarantees: confidentiality, integrity and authenticity [31]. A client-side code integrity verification mechanism aims to ensure that none of these properties can be violated by the code delivered by the untrusted server. In the case of a messaging application using E2EE, this means ensuring that untrusted code cannot access client-side secrets or decrypted messages, cannot send messages on behalf of the user, and that messages are correctly displayed.

3.2.1 Secret Exfiltration. The primary threat is the exfiltration of secret data. This spans both secret keys and decrypted content. Exfiltrating the client secret keys usually breaches all security guarantees of E2EE, while leaking decrypted data usually undermines confidentiality. The secret data may reside in different locations within the client environment, each enabling unique attack vectors. We consider the following two categories for secret client data:

Script-accessible Secrets. Many client secrets are only accessible via JS. This primarily includes client-side storage mechanisms [10], the URL fragment [14], and other client-side specific JS APIs [15, 36, 37]. Most client secrets are stored in client-side storage provided via browser APIs, e.g., `localStorage`, `sessionStorage`, or `IndexedDB`. For example, WhatsApp Web and Element use `IndexedDB` to store E2EE decryption keys. In some applications, client secrets are included in the URL fragment (i.e. the part of the URL after the # symbol) to allow easy sharing of links. An example of this is CryptPad [35]. The URL fragment is not sent to the server, but accessible via JS code. As opposed to client-side storage, the URL fragment can also be influenced by HTTP headers. Finally, some client secrets are accessible via other JS APIs. For example, a local hardware token can, e.g., be accessed via the WebAuthn API [15].

DOM-accessible Secrets. While the DOM can be accessed using JS, this category also includes secrets accessible using only HTML and CSS. In some applications, such as Bitwarden, client secrets are directly provided by the user via input fields (e.g., master password that is used to derive decryption key). Finally, decrypted content is often rendered into the DOM of the Web application to display it to the user. A recent "scriptless" attack [53] demonstrated that data exfiltration from the DOM is possible without executing any JS code, by relying solely on CSS.

3.2.2 Misrepresentation. A threat to the integrity and authenticity of content is misrepresentation, where the attacker modifies data displayed to the user. For example, the attacker could distribute a malicious font that displays arbitrary message content [41, 53]. Misrepresentation does not compromise confidentiality, but can be used to perform social engineering attacks, e.g., by displaying fake messages that trick the user.

3.3 Basic Attack Scenario

The following is an illustrative attack scenario that shows the inherent threats to E2EE in Web applications. Assume the end user has no integrity-checking browser extension installed. A malicious server operator can, at any time, distribute code that exfiltrates client secrets. For example, an attacker that has compromised the WhatsApp Web server could serve code that exfiltrates the decryption key, and then reloads a benign version of the Web application.

On all further visits, the attacker serves the original WhatsApp Web code, such that the user does not notice the attack. Exfiltrating the decryption key breaks all security guarantees of E2EE.

4 Common Pitfalls in Existing Approaches

In this section, we discuss existing approaches to verify client-side code in the browser and identify common pitfalls that undermine their security guarantees. We first focus on design pitfalls before presenting common implementation-level bugs in existing tools. Overall, we have identified and disclosed 18 vulnerabilities. A complete list of all reported vulnerabilities is provided in Appendix D.

4.1 Existing Approaches

We have selected the following five approaches for our survey based on their popularity (Code Verify, Signed Pages), academic impact (Accountable JS, WAIT), recency (Accountable JS, WEBCAT) and artifact availability (all). The concrete versions and excluded approaches are detailed in Appendix A.

Meta's Code Verify (CV). Meta's Code Verify [40] is available as a browser extension for Chrome and Firefox and is actively developed by Meta. It verifies the JS and CSS delivered for WhatsApp Web, Facebook and Instagram. This is achieved by collecting all JS and CSS from the DOM and comparing their cryptographic hashes to known-good values published by Meta. With over 30k installations, it constitutes the most widely used approach to verify client-side code in the browser.

Accountable JS (AJS). Accountable JS [18] is a recent academic proposal to verify client-side JS code in the browser. The corresponding paper focuses on protocol design for accountability, and formally proves security properties of the protocol. A prototype implementation is available as a browser extension for Chrome and Firefox, with a similar design as Code Verify. The focus of the project is, however, on the cryptographic protocol design. The extension is no longer maintained.

Signed Pages (SP). Signed Pages [22] is an open-source browser extension that verifies the integrity of entire Web pages. It is available for Chrome and Firefox. Contrary to the previous approaches, Signed Pages verifies the entire HTML of a Web page. In Chrome, it relies on DOM-based APIs to collect the HTML. In Firefox, it uses the `webRequest.filterResponseData` API [16] to inspect network requests. The extension is actively maintained.

WAIT. Web Application Integrity Transparency (WAIT) [38] is a Firefox-only browser extension that verifies page integrity by hashing the entire page. It also relies on Firefox's request filter API. The authors also propose a transparency log to address micro-targeting attacks and provide non-repudiation. The extension is no longer actively maintained.

WEBCAT (CAT). WEBCAT [3] is the most recent proposal and still in a prototype phase, with a developer preview published in March 2025 [2]. It is available as a Firefox extension, again relying on filtering network requests. The March 2025 WEBCAT preview provides strong integrity guarantees as well as a trust model based on Sigstore [42] and The Update Framework [45]. WEBCAT is actively developed by the Freedom of the Press Foundation and the developers have since fixed the issues raised in this work and made other major changes.

4.2 Common Design Pitfalls

In the following, we discuss common *design* pitfalls that are inherent to the proposed approaches, or have been overlooked during their design. We shortly describe each pitfall, illustrate it with a concrete attack scenario, and identify which existing tools are affected.

Table 1 summarizes the security guarantees of existing tools and highlights which of these are broken by our discovered pitfalls.

Table 1: **Comparison of existing tools: ○ = Design gap; ◐ = Implementation issue found; ● = Security guarantee achieved**

	CV	AJS	WAIT	SP	CAT
Script-based attacks	◐[P7, P8]	◐[P8]	●	◐[P8]	●
Header-based attacks	○[P3]	○[P3]	○[P3]	◐[P3]	◐[P3]
DOM-based attacks	◐[P2, P8]	○[P2]	◐[P3]	◐[P3]	●
Misrepresentation	○[P5]	○[P5]	○[P5]	○[P5]	○[P5]
Blocking	○[P1]	○[P1]	◐[P8]	○[P1]	●
Service workers	○[P4]	○[P4]	○[P4]	○[P4]	◐[P4]

4.2.1 Reactive vs. Proactive Verification. The majority of existing solutions only provide an indicator of compromise, e.g., via a red extension icon. They, by design, do not prevent the browser from executing unverified code. While this informs the user of a potential compromise, it fails to actually prevent it, as the malicious code is already executed before the user can react to the potential threat.

Furthermore, the "compromise indicator" does not persist across page reloads. This allows for trivial bypasses, where the attacker serves malicious code, and then immediately reloads a benign page. The indicator is only shown for a very short time (if at all), making it extremely hard to notice. After reloading the benign page, the indicator is reset and shows verification success. Currently, WAIT and WEBCAT are the only tools that offer proactive blocking.

Attack Scenario. The attack presented in Section 3.3 is a concrete example of this pitfall. In Code Verify, the page verification time is usually greater than the time to page reloading, resulting in no visible compromise indication at all.

P1 – Reactive Verification. The tool only detects attacks during execution, rather than preventing them proactively.

CV	AJS	WAIT	SP	WEBCAT

4.2.2 Partial Code Verification. Several existing tools only verify JS code, while ignoring other resources, such as stylesheets or the HTML structure. However, attacks can be performed by only modifying the HTML structure of a Web page. UI-based secret-exfiltration attacks are commonly applicable to user input. This includes login forms, where the user enters a master password (e.g., Bitwarden), as well as input fields for entering E2EE messages (e.g., WhatsApp Web). In the case of WhatsApp Web, a malicious initial QR code also allows linking the user's phone to an attacker-controlled session. Moreover, unverified CSS allows for scriptless attacks in which an attacker can extract arbitrary displayed information from the DOM via malicious stylesheets [23, 24, 53]. Overall, in addition to JS, HTML and CSS must be verified to guarantee the confidentiality of DOM-based secrets.

Attack Scenario. Assume the attacker wants to exfiltrate a user's Bitwarden master password entered on the login page. This page normally contains a form where the `handleLogin()` function does not expose the entered password to the server. Instead, an attacker that controls the server could send a malicious page as follows:

```
<form action="https://attacker.com/">
  <input type="password" id="masterPassword" />
</form>
<form onsubmit="handleLogin()" style="display:none">
```

The corresponding pages are visually indistinguishable and their JS is identical. However, upon submit, the malicious page exfiltrates the master password to the attacker.

P2 – Partial Code Verification. The tool only partially verifies code, rather than the entire client-side Web application code.

CV	AJS	WAIT	SP	WEBCAT

4.2.3 Missing Header Verification. HTTP headers can influence the behavior of a Web page in various ways. We have discovered that none of the existing tools verify HTTP headers that allow an attacker to compromise the integrity of delivered Web pages.

The `Link` header can be used to include additional stylesheets that are not referenced in the HTML document. Including a malicious stylesheet via the `Link` header can be used to exfiltrate secrets from the DOM [23, 24, 53]. Moreover, the `Location` header can be used to redirect the browser to a different URL. This redirect preserves the URL fragment, which contains secret keys in the context of E2EE cloud storage (ProtonDrive) and note-taking (CryptPad, Dropshare) applications. Thus, by setting the `Location` header, an attacker can exfiltrate those secrets from the URL fragment. Both examples highlight the importance of HTTP header verification. In Section 5.2.3, we detail more headers that can influence the behavior of a Web page and should be verified.

Attack Scenario. Assume that law enforcement has coerced CryptPad to provide access to an encrypted file. Now, when the encrypted blob is requested from CryptPad, the server's behavior is changed. Instead of replying with the normal application, the server now responds with a `302` redirect status code, pointing to `https://attacker.com/intercept`. The response body, apart from the Location header, is left unmodified. Hence, this change is invisible to any client-side integrity verification tool that does not consider HTTP headers, even if the entire HTML is verified. The victim's browser now follows the redirect. According to RFC 7231 [19, Section 7.1.2], the browser appends the URL fragment to the new request. On the attacker-controlled page, JS code can now extract the URL fragment, leaking the decryption key.

P3 – Missing Header Verification. The tool insufficiently verifies HTTP headers which can alter a Web page's behavior.

CV	AJS	WAIT	SP	WEBCAT

4.2.4 Missing Service Worker Verification. Service workers are a mechanism that allows Web pages to persist JS code through page reloads and even browser restarts. This JS code is not part of the Web page itself, but runs in the background. Still, it can access some client-side APIs (e.g., IndexedDB) or intercept and modify network requests. As such, service workers can be used to exfiltrate many script-accessible secrets, even if the Web page itself is benign.

Almost all existing tools fail to account for already installed service workers. As such, an attacker that is able to install a malicious service worker once (e.g., before the validation extension is installed, or with a fast reload as in Section 4.2.1) can persistently exfiltrate secrets, even if all future Web pages are benign. WEBCAT is the only tool that tries to remove installed service workers. However, due to an implementation bug, this fails under specific conditions (see Appendix D, Issue 6, 7).

Attack Scenario. Assume the attacker wants to exfiltrate the WhatsApp Web E2EE decryption key stored in `IndexedDB`. We assume the user is not yet logged in to WhatsApp Web, such that the secret key is not yet stored on the client. Upon visiting WhatsApp Web, the malicious server first serves a page that installs a service worker. The user aborts the login process, leaves the page and installs Meta's Code Verify extension. Afterwards, the user again browses to WhatsApp Web and logs in. This time, the server responds with a benign page, which is successfully verified by the extension. This service worker can now read the secret key from `IndexedDB` and send it to the attacker's server.

P4 – Missing Service Worker Verification. The tool does not properly verify or remove previously installed service workers.

CV	AJS	WAIT	SP	WEBCAT

4.2.5 Missing Resource Verification. Non-code remote resources, such as fonts and images loaded from remote URLs, can heavily influence the appearance of a Web page. In particular, fonts can be used to replace the displayed text with arbitrary other text, thus undermining the integrity and authenticity of E2EE messages [41, 53]. This can be used to perform social engineering attacks, e.g., by displaying fake messages that trick the user into performing unwanted actions. None of the existing tools verify remote resources, thus leaving this attack vector unmitigated.

Attack Scenario. In our attack scenario, law enforcement wants a user to deanonymize themselves in a messaging application with E2EE. The attacker serves a benign page, but replaces the remote font with a malicious one. This font replaces an existing text message from a trusted sender with a message that tricks the user into performing unwanted actions, e.g., "Please add Jane Doe to the group." This effectively undermines the integrity of E2EE messages, allowing the attacker to perform social engineering attacks.

P5 – Missing Resource Verification. The tool does not verify resources which may alter the appearance of a Web page.

CV	AJS	WAIT	SP	WEBCAT

4.2.6 Developer Misuse. In our threat model, Section 3.1, we assume a benign Web application developer that does not intentionally include backdoors or vulnerabilities. However, they may still inadvertently introduce insecure patterns into their code that allow the execution of unverified code. For example, a developer that signs the following Web page can still be attacked:

```
fetch("/api/secret.jsonp").then(resp => eval(resp));
```

As the data from the dynamic API endpoint is not signed, the server can respond with arbitrary JS code that is then executed via `eval()`. Similar patterns exist in real-world Web applications, like the nowadays out-of-style JSONP pattern [58].

In some tools, developer behavior that undermines the guarantees of a signature can be more subtle. For example, Signed Pages and WAIT put the "requirement that all referenced scripts and styles have an SRI integrity checksum" [38] on the developer. However, if the developer inadvertently publishes code that does not include SRI on some resource, their security guarantees are broken entirely. However, there is no warning to either the developer or user that this has occurred.

Similarly, Code Verify and Accountable JS only verify JS and CSS that is directly included in the HTML document. However, both approaches do not account for recursively imported JS modules, or stylesheets that are imported via `@import` statements in CSS. The contents of those recursive imports are not verified and thus are attacker-controlled. Thus, as soon as the developer inadvertently includes a single recursive import, all security guarantees are compromised.

Attack Scenario. The developer has signed the main HTML document, and all top-level scripts are verified via SRI. However, one of those scripts loads another script using the `import` directive. There is no signature or SRI for the import, which allows the attacker to control the contents of the imported module, where unverified code can be executed.

P6 – Developer Misuse. The tool allows publishing and trusting code that undermines the guarantees of a signature.

CV	AJS	WAIT	SP	WEBCAT

4.3 Common Implementation Pitfalls

The implementation of the verification logic is crucial to the security guarantees of the proposed approaches. In most cases, the verification logic is complex, often spanning thousands of lines of code. This complexity often leads to bugs that allow bypassing the verification logic entirely. In the following, we discuss common implementation-level pitfalls that we have identified in existing tools. A full breakdown of all findings can be found in Appendix D.

4.3.1 Parsing Complexities. Often, verification approaches rely on some parser to offer their security guarantees. In our analysis, we have encountered two main types of parsers.

First, some approaches, like Meta's Code Verify, aim to work on code that is in parts dynamic. To verify such code, they have to ignore dynamic parts of the code when comparing it to the ground truth. At first, Code Verify has simply removed parts of the code between two well-defined separators (i.e., `string.split()`). This has, however, allowed an attacker to hide code between the separators, such that this approach was ultimately replaced by a more robust approach. Currently, Code Verify parses the entire JS code of a Web application into an AST using *acorn.js*. Then, parts of the code are removed based on the AST. While this is more robust than the initial approach, it now delegates the security guarantees of the extension to a third-party parser. In particular, this parser must behave the same as the browser's parser, as this would otherwise allow simple bypasses. Prior research has, however, shown that parser differentials are common and often directly lead to vulnerabilities [30].

Second, security headers, such as the CSP, are commonly parsed and checked for disallowed directives. While parsing a CSP is fairly simple due to its strict grammar [55], there are still edge cases, especially due to the interplay of multiple CSPs. Furthermore, uncommon CSP directives can be overlooked.

P7 – Parser Vulnerabilities. The tool parses complex inputs, which is error-prone and often leads to vulnerabilities.

CV	AJS	WAIT	SP	WEBCAT

4.3.2 DOM API Misuse. Tools that operate on DOM-based APIs to collect Web application code are prone to oversights. In particular, modern valid DOMs are extremely complex, making it difficult to capture all relevant parts. For example, Code Verify collected all relevant JS as follows:

```
document.querySelectorAll('script');
```

However, this fails to find nested DOM elements, such as:

```
<iframe srcdoc='<script>...</script>'></iframe>
```

Furthermore, DOM-based APIs are not suitable for preventing the execution of malicious code, as they usually operate after the code has already been executed. For example, WAIT tries to ensure script integrity via a DOM-based API by removing script elements that lack integrity attributes. However, this removal happens only after execution of the script itself.

P8 – DOM-API-based Verification. The tool mis-uses DOM-based APIs to verify the Web page.

CV	AJS	WAIT	SP	WEBCAT

5 Our Design: CodeSeal

In this section, we present the design of our browser extension CodeSeal. We base our design on the pitfalls identified in Section 4 to avoid them entirely.

5.1 Core Design Principles

CodeSeal follows the design principles outlined in the following:
Proactive Blocking. Our design actively blocks unverified content before it can be executed. We are interacting with the network layer, and non-verifying responses are blocked before they are further processed by the browser. In particular, this prevents the execution of malicious JS code.
Holistic Verification. We verify the entire delivered Web page. This includes the response body (i.e. all HTML, JS, CSS), as well as behavior-changing HTTP headers. In addition, we expand our validation to include remote resources such as fonts and images. This prevents a malicious actor from misrepresenting the content of a validated page. We also explicitly verify (or uninstall) all registered service workers using a content script.
Simple Design. We avoid interacting with the DOM as well as parsing CSPs or HTML, which has proven to be error-prone in previous approaches. Instead, we rely on robust network-layer APIs that allow us to inspect and modify network requests before rendering.

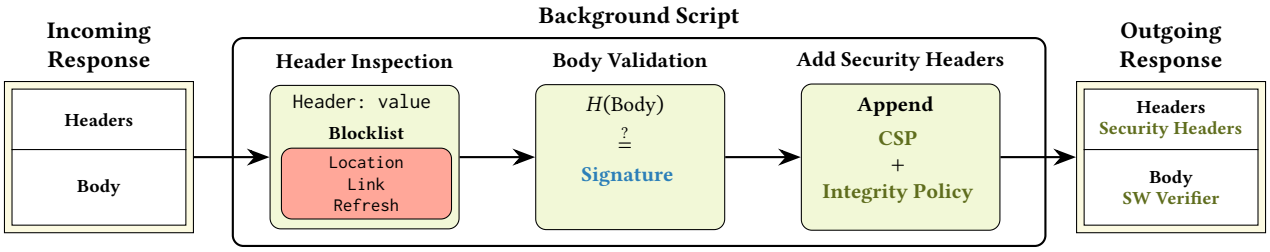


Figure 1: **Page response filtering design of CodeSeal. A response is checked for disallowed headers, then the signature of the body is verified. For enforcement, we append a CSP and Integrity Policy. Finally, we remove service workers of the page.**

Once the browser executes the client-side code, all security guarantees are enforced by the browser. Delegating the enforcement to the browser significantly reduces code complexity.

Signed = Secure. We avoid "foot-guns", i.e., a naïve but benign developer should not be able to break security guarantees inadvertently. To this end, we enforce Subresource Integrity (SRI) for all included scripts and stylesheets, using the recently introduced Integrity Policy [12] header. This header also prevents the recursive inclusion of scripts and stylesheets without SRI, which other tools have failed to verify. Moreover, we disallow the use of `unsafe-eval` in the CSP. Overall, this prevents the scenarios discussed in Section 4.2.6.

5.2 Implementation

Figure 1 provides an overview of our Firefox extension's implementation. It consists of a background script that implements the response verification and adds security headers, and a content script that verifies service workers. Fundamentally, we leverage the `webRequest.filterResponseData` API [16] in a background script to inspect incoming responses to requests. If the signature is not valid for the delivered content, the response is blocked and a warning page is displayed instead. Finally, the response is passed to the browser engine.

5.2.1 Added Security Headers. We leverage the browser's security mechanisms for resource integrity by adding headers to each response for enrolled pages. Before passing a response to the browser engine, our extension sets `Integrity-Policy` to enforce Subresource Integrity (SRI) for all included scripts and stylesheets. Additionally, we add a `Content-Security-Policy` that only allows resources that are either (1) embedded in the main page, or (2) are checked via SRI through the Integrity Policy header, or (3) are included inline via a data or blob URI. This allows us to verify full Web applications by only checking the signature for the main page, as all other resources are either inlined or verified via SRI.

Note that an attacker can control all original response headers. However, it cannot remove or modify headers that are added by the extension itself. For both CSP and Integrity-Policy, *all present* headers are enforced conjunctively [12, 55]. Thus, it is not possible to relax the policies by adding additional headers.

5.2.2 Response Body Verification. We intercept and verify the entire response body for Web pages (i.e., the main HTML document). The added security headers (Integrity Policy and CSP) guarantee the integrity of any included sub-resource, either directly (if the resource was inline) or via SRI.

Our prototype uses a configurable JSON Web Key [28] for enrolled sites, and verifies the signature against a user-configured public key through the Web Crypto API [57]. The signature for a Web page is delivered via a custom HTTP header. The (automatic) distribution of public keys to end-users is out of scope for our prototype and is further discussed in Section 6.2. Requests to enrolled pages without a valid signature are blocked by our extension and are replaced with a warning page.

5.2.3 Verifying HTTP Headers. HTTP headers can substantially change the way a Web page works. As detailed in Section 4.2.3, this can lead to unintended leakage of secrets, even if the page body remains unmodified. Thus, our extension verifies the absence of headers that can influence page behavior in a security-sensitive manner. We have manually reviewed all standardized HTTP headers and identified the `Location`, `Refresh` and `Link` headers as security-sensitive. Thus, we verify the absence of these headers in all responses to enrolled pages.

5.2.4 Service Workers. All newly installed service workers are verified by intercepting the corresponding network requests. Every service worker is verified separately from the main application, i.e., the developer must sign all service workers in addition to the main HTML page. The browser provides no information about previously (i.e., before installing CodeSeal) installed service workers. On the first load of an enrolled page with CodeSeal active, all service workers are unregistered. Afterwards, new, verified versions are installed. Unregistered service workers may continue running, but are guaranteed to be removed after a browser restart. Thus, if previously registered service workers are found and removed, the user is asked to restart the browser.

5.3 Web Page Compliance

In the following, we outline the requirements for a Web application to be verified using CodeSeal. A guide for developers and an overview of the compliance of popular web applications is in Appendix B.

Separation of Code and Data. The client-side code cannot directly contain any dynamic content; it must be static. Dynamic content must instead be loaded from dedicated endpoints, e.g., via XHR. Modern Web applications usually already implement data separation. Data separation is a general requirement for all client-side code verification tools.

Subresource Integrity. To link included scripts and styles to the page's trusted signature, SRI needs to be set for all resources. Since fonts and images do not support SRI, pages need to use Data URIs or Blob URIs to ensure integrity. Similar to prior work, we advocate extending the concept of SRI to all subresources [41].
Signature Delivery. The signature for the main Web page and all service workers needs to be included either in the HTTP response headers, or in a comment in the respective file.

6 Discussion

In this section, we provide an evaluation of CodeSeal on various Web applications. We comment on how signatures and keys for the integrity verification may be distributed, and discuss limitations.

6.1 Evaluation

We evaluate CodeSeal's practicality, security guarantees, and performance overhead on four real-world E2EE Web applications.

6.1.1 Practicality. To demonstrate the practicality of our approach, we implemented an example Web application for storing and reading E2EE notes similar to `CryptPad`, with implementations in both vanilla JS and React. The development overhead for CodeSeal compliance (Section 5.3) is minimal: We only require the use of SRI for all external scripts and stylesheets.

We also evaluate CodeSeal's practicality with popular real-world applications. For this, we choose the password manager Bitwarden, the E2EE Matrix client Element Web (both built on Webpack [8]), another Matrix client Cinny (built on Vite [7]), and the HIBP Password Checker (plain HTML). These applications require only minor build setup modifications to be compliant with CodeSeal: For Bitwarden, we use Webpack's `asset/inline` type and the `webpack-subresource-integrity` plugin to add Data URIs and SRI attributes. With Vite, we use `assetsInlineLimit` and `vite-plugin-sri-gen` respectively. Finally, we sign the main HTML document using our signing script. No code changes are required, showing that applications can be easily adapted to CodeSeal.

6.1.2 Security. We manually verified that CodeSeal prevents all attack scenarios outlined in Section 4. To the best of our knowledge, CodeSeal is the only tool that systematically analyzes all resources that can influence the behavior of a Web page and verifies their integrity or absence. For this, we have implemented three versions of our test application that store client side secrets in `localStorage`, the URL fragment, or user input, respectively. This covers all threats explored in Section 3.2.1. We have verified that CodeSeal successfully prevents secret exfiltration in all cases, while each prior tool fails in at least one case (Section 4).

6.1.3 Performance. CodeSeal's performance overhead is negligible in real-world applications. On average, the time to Largest Contentful Paint [56] increases by 1.76 % (5.96 ms) with Element Web and 1.30 % (2.50 ms) with Cinny (See Appendix C).

6.2 Establishing Trust

Our tool currently verifies a cryptographic signature of the client-side Web application. In our extension, end-users can configure public keys for enrolled sites, but key distribution is outside the scope of this paper. Prior research has proposed various methods to distribute such trust anchors in a decentralized manner, e.g., via Web of Trust [54] or Key Transparency [39]. Beyond classical key distribution, modern cryptographic approaches [18, 45, 50] can guarantee authenticity, integrity and freshness [5, 45] of the delivered code. Moreover, they can provide accountability [18] and transparency [49], e.g., by appending all signed versions to a public, append-only log. Such mechanisms ensure that even a targeted attack on a single user would be detectable through its traces in the public log [49]. For native software, production-ready distribution mechanisms such as The Update Framework (TUF) [5, 45] and in-toto [50] have already been adopted by major projects.

6.3 Dangerous Permissions

Permissions to dangerous browser APIs [51] can be reused by an attacker, e.g., via domain re-registration [34]. CodeSeal mitigates this by shifting the trust-model of client-side code.

6.4 Limitations

Our implementation uses the `webRequest.filterResponseData` API [16], which is currently only available in Firefox [21]. This API is necessary to prevent the rendering and execution of untrusted content (see Section 4.2.1). Both other blocking tools (WEBCAT, WAIT) use the same API, and are also only compatible with Firefox. All other features used by CodeSeal are universally-available.

In place of the extension, a local TLS-terminating proxy could also verify responses. While this approach is browser-independent, it has several limitations: It needs a separate native process running alongside the browser, and requires the user to install a custom TLS certificate, which poses risks of social-engineering abuse. Further, directly managing service workers is not possible.

Fundamentally, the Web application needs to be compliant with our approach (see Section 5.3). This is usually achievable with small changes, as shown in Section 6.1. By design, our approach is incompatible with customizing the code of a Web application.

7 Conclusion

In this paper, we have systematically analyzed five Web application integrity verification tools and identified common design pitfalls and implementation vulnerabilities that undermine their security guarantees. We responsibly disclosed over a dozen critical bugs to the respective developers, many of which allowed for secret exfiltration attacks that violate the fundamental goals of E2EE.

We addressed these shortcomings in CodeSeal, an extension implementation that systematically avoids the identified pitfalls and provides strong client-side code integrity guarantees. Our work improves the state-of-the-art in Web application integrity verification and provides a foundation for securing E2EE Web applications.

Acknowledgments

We want to thank our anonymous reviewers for their comments and suggestions. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. We also thank Christian Rossow and Michael Schwarz for their valuable feedback on the paper. All authors carried out this work as members of the Saarbrücken Graduate School of Computer Science at Saarland University.

References

[1] Shubham Agarwal. 2022. Helping or Hindering? How Browser Extensions Undermine Security. In *CCS*.

[2] Giulio Berra. 2025. Introducing WEBCAT: Web-based Code Assurance and Transparency. https://securedrop.org/news/introducing-webcat-web-based-code-assurance-and-transparency/

[3] Giulio Berra. 2025. WEBCAT: Web-based Code Assurance and Transparency. Cryptology ePrint Archive, Paper 2025/797. https://eprint.iacr.org/2025/797

[4] Stefano Calzavara, Samuele Casarin, and Riccardo Focardi. 2025. Dynamic Security Analysis of JavaScript: Are We There Yet?. In *WWW*.

[5] Justin Cappos, Trishank Karthik Kuppusamy, Joshua Lock, Marina Moore, and Lukas Pühringer. 2025. The Update Framework Specification v1.0.33. https://theupdateframework.github.io/specification/v1.0.33/

[6] Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich. 2010. Separating Web Applications from User Data Storage with BSTORE. In *USENIX Conference on Web Application Development (WebApps)*.

[7] Vite Contributors. 2025. Vite. https://vite.dev/

[8] Webpack Contributors. 2025. Webpack. https://webpack.js.org/

[9] MDN Web Docs. 2025. Browser extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions

[10] MDN Web Docs. 2025. Client-side storage. https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Client-side_storage

[11] MDN Web Docs. 2025. HTTP caching. https://developer.mozilla.org/de/docs/Web/HTTP/Guides/Caching

[12] MDN Web Docs. 2025. Integrity-Policy Header. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Integrity-Policy

[13] MDN Web Docs. 2025. Subresource Integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

[14] MDN Web Docs. 2025. URI fragment. https://developer.mozilla.org/en-US/docs/Web/URI/Reference/Fragment

[15] MDN Web Docs. 2025. Web Authentication API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API

[16] MDN Web Docs. 2025. webRequest.filterResponseData API. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData

[17] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. 2013. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *CCS*. doi:10.1145/2508859.2516708

[18] Ilkan Esiyok, Pascal Berrang, Katriel Cohn-Gordon, and Robert Künnemann. 2023. Accountable Javascript Code Delivery. In *NDSS*.

[19] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. doi:10.17487/RFC7231

[20] Google. 2025. Google Play Store. https://play.google.com/

[21] Google Developers. 2024. Migrate to Manifest V3. https://developer.chrome.com/docs/extensions/develop/migrate

[22] Tom Hacohen. 2020. Signed Pages: A browser extension to verify the authenticity (PGP signature) of web pages. https://github.com/tasn/webext-signed-pages

[23] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *CCS*.

[24] 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).

[25] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. 2012. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*.

[26] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. 2014. Analyzing forged SSL certificates in the wild. In *IEEE Symposium on Security and Privacy (SP)*.

[27] Martin Johns and Christian Beyerlein. 2007. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of the ACM symposium on Applied computing*.

[28] Michael B. Jones. 2015. JSON Web Key (JWK). RFC 7517. doi:10.17487/RFC7517

[29] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE Symposium on Security and Privacy (SP)*. doi:10.1109/SP.2016.58

[30] David Klein and Martin Johns. 2024. Parse Me, Baby, One More Time: Bypassing HTML Sanitizer via Parsing Differentials. In *IEEE Symposium on Security and Privacy (SP)*.

[31] Mallory Knodel, Sofia Celi, Olaf Kolkman, and Gurshabad Grover. 2024. Definition of End-to-end Encryption. *Cryptology ePrint Archive* (2024).

[32] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser Fingerprinting: A Survey. In *ACM Transactions on the Web*.

[33] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *USENIX Security Symposium*.

[34] Chaz Lever, Robert Walls, Yacin Nadji, David Dagon, Patrick McDaniel, and Manos Antonakakis. 2016. Domain-Z: 28 registrations later measuring the exploitation of residual trust in domains. In *IEEE Symposium on Security and Privacy (SP)*.

[35] Aaron MacSween, Caleb James Delisle, Paul Libbrecht, and Yann Flory. 2018. Private Document Editing with some Trust. In *Proceedings of the ACM Symposium on Document Engineering* (Halifax, NS, Canada) *(DocEng '18)*. doi:10.1145/3209280.3209535

[36] MDN Web Docs. 2023. WebHID. https://developer.mozilla.org/en-US/docs/Web/API/WebHID_API

[37] MDN Web Docs. 2023. WebSerial. https://developer.mozilla.org/en-US/docs/Web/API/USB

[38] Echo Meißner, Frank Kargl, and Benjamin Erb. 2021. WAIT: Protecting the Integrity of Web Applications with Binary-Equivalent Transparency. In *Proceedings of the Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. 4 pages. doi:10.1145/3412841.3442143

[39] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *USENIX Security Symposium*.

[40] 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/

[41] Tobias Mueller, Daniel Klotzsche, Dominik Herrmann, and Hannes Federrath. 2019. Dangers and Prevalence of Unprotected Web Fonts. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*.

[42] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *CCS*.

[43] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. 2014. Building web applications on top of encrypted data using Mylar. In *USENIX Symposium on Networked Systems Design and Implementation*.

[44] Eric Rescorla and Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. doi:10.17487/RFC5246

[45] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable key compromise in software update systems. In *CCS*.

[46] Signal Technology Foundation. 2025. Signal Android APK distribution page. https://signal.org/android/apk/

[47] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. 2014. Password Managers: Attacks and Defenses. In *USENIX Security Symposium*.

[48] The Guardian. 2013. Lavabit email service abruptly shut down citing government interference. https://www.theguardian.com/technology/2013/aug/08/lavabit-email-shut-down-edward-snowden

[49] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. 2019. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

[50] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *USENIX Security Symposium*.

[51] Leon Trampert, Lorenz Hetterich, Lukas Gerlach, Mona Schappert, Christian Rossow, and Michael Schwarz. 2025. Peripheral Instinct: How External Devices Breach Browser Sandboxes. In *WWW*.

[52] 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*.

[53] Leon Trampert, Daniel Weber, Christian Rossow, and Michael Schwarz. 2025. Styled to Steal: The Overlooked Attack Surface in Email Clients. In *CCS*.

[54] Alexander Ulrich, Ralph Holz, Peter Hauck, and Georg Carle. 2011. Investigating the OpenPGP Web of Trust. In *European Symposium on Research in Computer Security*.

[55] W3C. 2024. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/

[56] W3C. 2025. Largest Contentful Paint. https://www.w3.org/TR/largest-contentful-paint/

[57] W3C. 2025. Web Cryptography Level 2. https://www.w3.org/TR/webcrypto-2/

[58] Wikipedia. 2025. JSONP. https://en.wikipedia.org/wiki/JSONP

[59] WIRED. 2016. A Government Error Just Revealed Snowden Was the Target in the Lavabit Case. https://www.wired.com/2016/03/government-error-just-revealed-snowden-target-lavabit-case/

A Surveyed Code Integrity Tools

Table 2 lists the tools that verify the integrity of client-side Web application code we include in our survey (see Section 4). It includes the respective repositories and versions used for testing and evaluating their security guarantees on the needs of real-world E2EE Web applications. Note that the most recent versions of the tools may have already updated their implementation and improved their designs after our responsible disclosure.

Table 2: **Repositories and Versions of the Extensions Surveyed**

Tool	Repository	Revision
Code Verify	facebookincubator/meta-code-verify	Multiple
Accountable JS	iesiyok/accountable-js	6cccdf3
Signed Pages	tasn/webext-signed-pages	a5e9a64
WAIT	vs-uulm/wait-prototype	d7d558e
WEBCAT	freedomofpress/webcat	d44b1ef

Meanwhile, Table 3 contains tools that we decided to exclude from our survey, even though they share the same goal of verifying the integrity of client-side Web application code. Some of them are referenced in previous work [3].

We decided to exclude Mylar [43] since the last update to the tool was over 10 years ago in 2014. Naturally, it is also no longer compatible with recent browser versions. Verena [29] was excluded due to the unavailability of the tool. To the best of our knowledge, the publication provides no artifact. Furthermore, it is also likely no longer compatible with recent browser versions. The Page Integrity project was excluded as it has been discontinued and is no longer available online.

Table 3: **Extensions Excluded from the Survey**

Tool	Link	Reason
Mylar [43]	css.csail.mit.edu/mylar/	Last update 2014 & Incompatibility
Verena [29]	-	Unavailable
Page Integrity	pageintegrity.net (Archive)	Discontinued

B Web Application Compliance

This section provides concrete action items for Web developers to make their applications compliant with CodeSeal. Based on our evaluation of real-world applications (Section 6.1), most Web applications implementing E2EE can be adapted with minor changes.

B.1 Developer Guide

Making a Web Application compliant with CodeSeal requires the following steps. Note that this omits the final step, signature generation and delivery, as it is beyond the scope of this paper.

Separation of Code and Data. First, ensure that the client-side code is static and does not contain dynamic content. All dynamic content (e.g., user data, messages, decrypted content) must be loaded from API endpoints via XHR or the Fetch API.

Table 4: **Compliance status of popular E2EE Web applications with CodeSeal requirements. ● indicates compliance, ◐ indicates partial compliance, and ○ indicates non-compliance.**

Application	Code/Data Separation	SRI Attrs	Data URIs for Assets
Bitwarden	●	○	○
(adapted)	●	●	●
Cinny	●	○	○
(adapted)	●	●	●
Element	●	○	○
(adapted)	●	●	●
Have I Been Pwned	●	◐	n/a
(adapted)	●	●	n/a
WhatsApp Web	◐	○	○
CryptPad	●	○	○
Jitsi Meet	●	○	○
PrivateBin	●	◐	○

SRI Attributes. All external scripts and stylesheets must include subresource integrity attributes. This can generally be done automatically during the build process by using existing bundler plugins, like `webpack-subresource-integrity` for Webpack or `vite-plugin-sri-gen` for Vite.

Data URIs for Fonts and Images. Since fonts and images do not support SRI, they must be embedded using Data URIs or Blob URIs. This can also often be automated by the bundler during the build process, e.g., using `asset/inline` modules in Webpack.

B.2 Compliance Status of Popular Applications

Table 4 shows the compliance status of popular Web applications with respect to CodeSeal's requirements.

The table shows that most popular E2EE applications already implement code and data separation, which is a fundamental requirement for E2EE applications. However, SRI attributes are often missing or only partially implemented (for some but not all resources), and none of the surveyed applications currently use Data URIs for all assets. However, compared to achieving code and data separation, adopting SRI and Data URIs is often straightforward using existing build tools and plugins. We have demonstrated this with Bitwarden, Cinny, Element and Have I Been Pwned, which we successfully adapted to be fully compliant with only minor changes (Section 6.1).

C Performance Evaluation

Figure 2 shows the performance impact of request verification in CodeSeal. We measure the Largest Contentful Paint (LCP) on an adapted and signed version of the Bitwarden Web client, Cinny Matrix client, and Element Web client, and Have I Been Pwned Application with and without verification. Page verification with CodeSeal shows mean overheads between 1.30 % and 7.98 % (1.38 ms to 25.56 ms), as shown in Table 5.

We also evaluate the memory usage of the browser with and without verification active and find no statistically significant difference.

C.1 Signature Verification and Hashing

The inherent additional cost stems from hashing the main page body and verifying the embedded signature. We measure this operation specifically and log how many bytes were verified: For Cinny, this is only 3875 bytes of main page HTML, taking around 1ms. For Bitwarden, CodeSeal verifies the main page HTML and a 2.1 MB WebAssembly module in around 7ms.

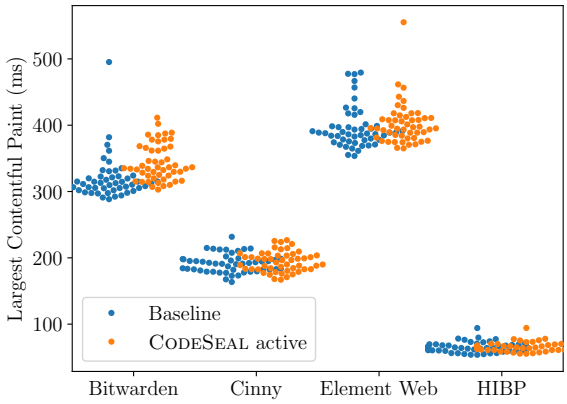


Figure 2: **Comparison of the Largest Contentful Paint overhead on popular E2EE Web applications with and without CodeSeal verification.**

Table 5: **Performance statistics for LCP measurements**

Target	Kind	Samples	LCP (ms)
Bitwarden	Baseline	50	318.96 ± 31.98
	Active		344.52 ± 26.95
	Overhead		+25.56 (+7.98 %)
Cinny	Baseline	50	191.80 ± 13.73
	Active		194.30 ± 14.83
	Overhead		+2.50 (+1.30 %)
Element	Baseline	50	395.78 ± 31.47
	Active		401.74 ± 30.31
	Overhead		+5.96 (+1.76 %)
Have I Been Pwned	Baseline	50	64.98 ± 7.66
	Active		66.36 ± 5.45
	Overhead		+1.38 (+2.12 %)

D Discovered Vulnerabilities

In Table 6 we provide more details on the vulnerabilities we have discovered across all different integrity verification tools that we include in our survey (see Section 4). For each vulnerability, we state which tools were affected, provide a short description of the vulnerability, and map them to their respective pitfalls as categorized in Section 4, if applicable. We have responsibly disclosed all vulnerabilities to the respective maintainers.

Table 6: **Vulnerabilities discovered in the code integrity tools surveyed in Section 4. For each vulnerability, we also state the respective pitfall, if applicable.**

#	Tool	Pitfall	Description
1	Code Verify Accountable JS Signed Pages	P1	The tool does not block the execution of unverified code.
2	Code Verify Accountable JS	P2	The tool does not verify the response HTML. The tool verifies only JS, not CSS and HTML.
3	Code Verify Accountable JS Signed Pages WAIT	P3	The tool does not verify stylesheets imported via the `Link` header.
4	Accountable JS Signed Pages WAIT WEBCAT	P3	The tool does not verify the HTTP `Location` header. It directly allows leaking the URL fragment.
5	Code Verify Accountable JS Signed Pages WAIT	P4	The tool does not verify or remove service workers which have been installed prior to the tool.
6	WEBCAT	P4	The tool fails to remove service workers under specific conditions: To remove service workers, it injects custom JS "hooks" into all externally sourced JS. Those hooks are not executed if all JS is delivered in-line.
7	WEBCAT	P4	The tool de-registers Service Workers on all page visits. Due to the asynchronous Service Worker lifecycle, this does not guarantee that the Service Worker is already removed when the user visits the page.
8	*All tools*	P5	The tools do not verify remote resources, such as fonts and images.
9	Code Verify	P7	Due to multiple parser errors, the tool allows untrusted code in parts of the JS that are assumed to contain only data. This data is explicitly excluded from the verification. *(Found and reported as three separate bugs)*
10	Accountable JS	P7	The tool employs a regular expression to parse hostnames from URLs. This regex fails to normalize hostnames before checking for equality.
11	Code Verify Accountable JS WAIT	P8	The tool does not verify DOM elements nested via `<iframe srcdoc>`.
12	Code Verify	P8	The tool does not verify JS within `<math>` tags.
13	Accountable JS	P8	The tool does not verify all inline event handlers.
14	Signed Pages	P8	In Chrome, the tool uses the DOM API to verify the overall DOM after the `<body>` DOM element is completed. With chunked requests, a `<script>` tag after the `<body>` can be injected, which is not verified.
15	WAIT	P8	The tool finds and removes all JS and CSS resources that do not have a SRI attribute set. However, the deleted JS is executed prior to removal.
16	WAIT	Other	The tool incorrectly verifies the number of present signatures, allowing to re-use the same signature multiple times.
17	Code Verify	Other	The tool has a time-of-check to time-of-use (TOCTOU) vulnerability while verifying the integrity of remote resources. It first fetches a resource and verifies it. Then the browser fetches the resource again and executes it.
18	Signed Pages	Other	In Firefox, the tool only verifies the first chunk of a chunked HTTP response.