TrustyMon: Practical Detection of DOM-based Cross-Site Scripting Attacks Using Trusted Types

Sunnyeo Park
KAIST
Daejeon, Republic of Korea
psnyeo88@kaist.ac.kr

Jihwan Kim
KAIST
Daejeon, Republic of Korea
jhkim@enki.co.kr

Seongho Keum
KAIST
Daejeon, Republic of Korea
keum07@kaist.ac.kr

Hyunjoon Lee
KAIST
Daejeon, Republic of Korea
hjpaul@kaist.ac.kr

Sooel Son
KAIST
Daejeon, Republic of Korea
sl.son@kaist.ac.kr

Abstract

The prevailing presence of DOM-based cross-site scripting (XSS) vulnerabilities on the Web poses a critical security threat to Internet surfers. Previous research has demonstrated the effectiveness of dynamic detection and monitoring methods for XSS attacks, demonstrating their promising potential in mitigating security threats. However, this research direction suffers from a deployment issue, necessitating browser changes and significant refactoring to web applications, which hinders their practical deployment.

To address these deployment challenges, we propose TrustyMon. The key idea is to leverage a new browser-integrated framework known as Trusted Types. We design TrustyMon to automatically collect benign signatures representing JavaScript (JS) snippets that implement website functionality. TrustyMon-protected webpages then match every JS code injected via XSS sinks among the benign JS signatures in runtime by leveraging Trusted Types in browsers. We demonstrate the efficacy of TrustyMon in accurately identifying DOM-based XSS attacks across 16 real-world web applications without false negatives. By leveraging Trusted Types, we improve the practical deployment of dynamic monitoring in client-side browsers without requiring any changes to browsers. TrustyMon introduces a negligible latency of 27.02 ms in page loading time during its deployment, exemplifying a practical and readily deployable monitoring framework for detecting DOM-based XSS attacks.

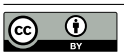
CCS Concepts

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

Keywords

DOM-based XSS, Trusted Types, Dynamic monitoring

ACM Reference Format:
Sunnyeo Park, Jihwan Kim, Seongho Keum, Hyunjoon Lee, and Sooel Son. 2025. TrustyMon: Practical Detection of DOM-based Cross-Site Scripting Attacks Using Trusted Types. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3708821.3733889



ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/25/08
https://doi.org/10.1145/3708821.3733889

1 Introduction

Cross-site scripting (XSS) is a pervasive and critical security threat on the Web. XSS has consistently ranked in the OWASP Top 10 threats [20, 42], and Invicti reported that approximately 25% of websites have XSS vulnerabilities [23]. In particular, DOM-based XSS vulnerabilities have become increasingly prevalent as web development trends shift toward single-page applications that dynamically update web content using JavaScript (JS) in response to user interactions [18, 60]. Previous studies also demonstrated that approximately 10% of the Alexa Top 10,000 websites were vulnerable to DOM-based XSS attacks [29, 54].

Previous research has focused on preventing DOM-based XSS vulnerabilities by using secure APIs and sanitizers [21, 41, 62] or by identifying vulnerabilities using static, dynamic, and hybrid analyses as well as machine learning [5, 29, 35, 36, 46, 57]. Despite these efforts, the number of reported XSS vulnerabilities has continued to increase from 470 in 2011 to 22,000 by April 2022 [15]. Addressing these identified vulnerabilities remains challenging. Stock *et al.* found that only 10.5% of DOM-based XSS vulnerabilities were patched within a month after vulnerability notifications were sent, leaving the majority of vulnerabilities unpatched [56].

Unfortunately, even after security patches are released, many vulnerabilities remain exploitable until site operators apply these patches. Approximately 52% of websites continue to operate with known security vulnerabilities [58]. Accordingly, merely identifying and patching vulnerabilities still leaves website users unprotected. It is therefore imperative to safeguard users against DOM-based XSS attacks that exploit unknown or unpatched vulnerabilities.

Dynamic detection and monitoring systems enable the real-time detection of XSS attacks. By observing unusual or malicious behaviors, one can take preventive actions when an attack occurs. For example, intrusion detection systems (IDS) monitor network traffic to detect unauthorized access or malicious activity [30]. In a similar vein, DOM-based XSS attacks can be dynamically monitored, which complements traditional vulnerability identification methods by detecting ongoing attacks that exploit unknown or unpatched vulnerabilities, thus serving as a second line of defense.

Several approaches for dynamically monitoring DOM-based XSS attacks have been proposed, including allowlist-based filtering [17, 24, 39, 59], taint analysis [55], and randomization techniques [4]. However, it is challenging to deploy these approaches on real-world environments from a practical perspective; they often

require significant modifications to server-side web applications, changes to browser implementations [4, 24, 39, 55], or introduce non-negligible execution overhead [59]. Accordingly, these challenges hinder their widespread adoption in practice.

We posit that designing a practical and widely deployable monitoring system for DOM-based XSS attacks should address the following requirements: (1) it should not require changes to browser implementations or server-side web applications; (2) it should not necessitate constant maintenance or updates as client-side browsers evolve; and (3) it should impose negligible performance overhead.

In this paper, we present TrustyMon, a dynamic monitoring framework for DOM-based XSS attacks. Unlike previous approaches [4, 17, 55, 59], TrustyMon addresses the aforementioned requirements by using *Trusted Types* [26]. Trusted Types is a browser-supported framework designed to help web developers mitigate DOM-based XSS by ensuring that sensitive injection sinks only accept trusted data. By leveraging the Trusted Types functionality, TrustyMon dynamically monitors the arguments passed to injection sinks that are able to introduce JS snippets and confirms whether the injected scripts are safe, thus monitoring real-time XSS attacks.

TrustyMon operates in two phases: extraction and monitoring. In the extraction phase, TrustyMon automatically collects a set of signatures for benign JS snippets or URLs that are passed to injection sinks by crawling a given website. In the monitoring phase, TrustyMon dynamically monitors injection sinks that can potentially cause DOM-based XSS attacks and checks whether the supplied arguments are intended values by matching their signatures against the previously collected benign signatures. For this, TrustyMon leverages Trusted Types, supported by Chromium-based browsers. Any violation triggered by Trusted Types indicates a potential DOM-based XSS attack, where injected JS snippets do not match the benign signatures. These violations are then reported to TrustyMon's server-side framework for further actions.

We note that adopting Trusted Types is known to require arduous engineering effort [25, 62, 63]. In contrast, we designed TrustyMon to be trivially deployable, minimizing the burden on website operators. TrustyMon does not require any modifications to browsers, and website operators are able to deploy TrustyMon by simply inserting a content security policy (CSP) header and loading the necessary JS libraries. This enables TrustyMon to monitor all sensitive injection sinks that Trusted Types cover without introducing side effects or compatibility issues. Moreover, TrustyMon benefits from Chromium's ongoing support for Trusted Types and its updates as web standards (e.g., ECMAScript and HTML5) evolve [6, 7], facilitating the maintenance of TrustyMon.

Previous studies [17, 39, 50, 59] have demonstrated the effectiveness of allowlist-based filtering approaches. These approaches assume that a complete list of benign signatures is available and that the collected signatures remain static over time. However, these assumptions often do not hold in practice. To address this, TrustyMon provides Report Controller that enables site operators to easily register new benign JS signatures based on violation reports, thereby minimizing the allowlist maintenance burden on site operators.

We evaluated the efficacy of TrustyMon on 16 web applications, each containing at least one DOM-based XSS vulnerability. The experimental results show that TrustyMon successfully detected all attempted attacks without false negatives. It also imposes negligible overhead, with an average increase of 27.02 ms (5.68%) in page loading time across the benchmark applications. Furthermore, when comparing TrustyMon to existing DOM-based XSS detection work, TrustyMon attained better performance in detecting XSS attacks and lower page loading latency without necessitating changes to internal browser logic or server-side web applications.

In summary, we propose a practical approach to monitoring DOM-based XSS attacks using Trusted Types. We demonstrate that TrustyMon is effective for monitoring DOM-based XSS attacks with negligible overhead and easily deployable without requiring changes to server-side web applications and client browsers.

```
<script>
  var name = decodeURIComponent(
              document.location.hash.substr(1));
  document.write(name);
</script>
// Payload: http://example.com/welcome.html#<script>
 alert(document.cookie)</script>
```

Listing 1: Example of a DOM-based XSS Vulnerability.

2 Background

2.1 DOM-based XSS

Cross-site scripting (XSS) is one of the most prevalent and critical web threats, consistently ranking first among reported vulnerability types [19, 49]. DOM-based XSS is a specific type of XSS attack that injects malicious JS code into a target webpage. It exploits JS injection sinks, such as `document.write`, `innerHtml`, and `eval`, causing these injection sinks to add malicious JS code from attacker-controlled input vectors, including `document.URL`, `location.hash`, `window.location`, and `postMessage.data`.

Listing 1 shows a JS example that has a DOM-based XSS vulnerability. Assuming that users are expected to access this webpage with the URL http://example.com/welcome.html#Alice, this webpage is supposed to print out "Alice". However, an attacker can exploit this by composing a URL like http://example.com/welcome.html#<script>alert(document.cookie)</script>, which injects the JS code in the URL fragment section into the `document.write()` function. This results in the JS code being injected into the web document, exploiting the DOM-based XSS vulnerability.

DOM-based XSS has become prevalent as web applications offer dynamic and interactive user experiences using JS. The highly dynamic and API-dependent nature of JS often leads to intricate control and data flows, making the identification of DOM-based XSS vulnerabilities challenging.

2.2 Trusted Types

Trusted Types is a built-in browser framework designed to mitigate DOM-based XSS vulnerabilities [26, 62]. Its objective is to provide a set of *Trusted Types* for HTML, URL, and JS, ensuring that browsers accept only trusted values at security-critical injection sinks, which may cause DOM-based XSS vulnerabilities.

The Trusted Types standard [26] defines a set of sensitive DOM XSS injection sinks that dynamically modify the DOM structure and insert JS snippets based on user input strings. We refer to these DOM XSS injection sinks as *injection sinks* in this paper. At runtime, browsers enforce Trusted Types by ensuring that these injection

```
1  <html><head>
2    <meta http-equiv="Content-Security-Policy"
3      content="require-trusted-types-for 'script';
4               trusted-types default">
5    <script>
6      trustedTypes.createPolicy('default', {
7       createHTML: (string) =>
8                     string.replace(/\</g, '&lt;')});
9    </script>
10 </head>
11 <body><script>
12   var name = decodeURIComponent(
13                document.location.hash.substr(1));
14   document.write(name);
15 </script></body></html>
```

Listing 2: JS example using Trusted Types.

sinks accept only trusted values as their arguments, preventing untrusted strings from being executed.

Specifically, Trusted Types introduces three internal types: `TrustedHTML`, `TrustedScript`, and `TrustedScriptURL`. Trusted Type objects can only be created through user-defined policies (`TrustedTypePolicy`). These policies define a set of functions (`createHTML`, `createScript`, and `createScriptURL`) that convert strings into Trusted Type objects after applying sanitization or validation logic. Listings 7 and 8 show how Trusted Type policies are defined and how a string is converted into a Trusted Type object.

One of the policies, a *default* policy is special, which is implicitly invoked whenever an injection sink receives a string instead of a Trusted Type object. In other words, the functions in the default policy (i.e., `createXX`) act as callback functions for each Trusted Type. For instance, when `innerHTML`, which inserts HTML strings into the DOM, receives a string, the default policy's `createHTML` function is automatically invoked. Therefore, we refer to the set of functions in the default policy (i.e., `createHTML`, `createScript`, and `createScriptURL`) as *Trusted Type callbacks* in this paper.

Listing 2 shows a JS example using Trusted Types. Lines (Lns) 2-4 instruct the browser to enable Trusted Types and ensure that all DOM-based XSS injection sinks use Trusted Type values. The `require-trusted-types-for` directive enforces Trusted Types at all injection sinks. The `trusted-types` directive instructs the browser to apply the specified policies. Trusted Type policies define functions that sanitize input arguments and return a Trusted Type object. Lns 6-8 define the `default` policy, which includes a `createHTML` function that converts all left angle brackets (i.e., <) into `<`, thus preventing the injection of HTML tags. When the `default` policy is used, all injection sinks invoke the callback functions corresponding to their type. At Ln 14, `document.write` method invokes the `createHTML` function before executing any provided content. Finally, the injection sink of `document.write` is executed only if the given argument is of type `TrustedHTML`. Otherwise, it throws an error, rejecting any untrusted argument types.

As shown in the provided examples, each Trusted Type policy requires web developers to implement sanitization logic that transforms a given argument string into its sanitized version. Consequently, web developers are tasked with defining proper policies, considering the injection sinks and the contexts in which user inputs appear within a webpage. Accordingly, such tasks become challenging for large applications, demanding significant engineering effort in refactoring legacy web applications [25, 62, 63].

Chrome has supported Trusted Types since Chrome 83 [2]. Trusted Types is enabled when a CSP header includes the `require-trusted-types-for` and the `trusted-types` directives [32, 33], which outlines the Trusted Type policies that browsers enforce at runtime. 130 Google services, including Photos, Contacts, and My Activity, have deployed Trusted Types [25]. Moreover, Trusted Types is supported in popular libraries, such as Angular, React, Karma, and Webpack. The percentage of Chrome page views using Trusted Types increased from 1.8% in 2021 to 14.2% in 2024 [65].

3 Motivation

Modern web applications have become increasingly complex, rendering significant challenges in identifying web vulnerabilities before their deployments. Specifically, detecting unknown XSS vulnerabilities [5, 29, 35, 36, 44] remains a difficult task; XSS attacks have consistently ranked among the OWASP Top 10 security threats [20, 42]. Accordingly, the runtime detection of XSS attacks serves as a critical second line of defense, effectively mitigating the security risks that XSS vulnerabilities pose.

Previous studies have proposed various methods for dynamically monitoring and detecting DOM-based XSS attacks, including allowlist-based filtering [17, 24, 39, 59], taint analysis [55], and instruction set randomization [4]. While these methods have shown potential in detecting XSS attacks, they face significant deployment challenges. They often require modifications to server-side applications or user browsers [4, 24, 39, 55] and introduce considerable overhead [39, 55, 59]. Therefore, there is a clear need for a dynamic monitoring framework that is both practical and widely deployable.

3.1 Technical Challenges

Implementing a practical and widely deployable monitoring framework for DOM-based XSS detection involves addressing the following key technical challenges.

No changes to browsers. For large-scale deployment, the monitoring tool should be compatible with a wide range of users' browsers. Each browser is a potential target of DOM-based XSS attacks. However, it also simultaneously presents an opportunity for the monitoring tool to detect these attacks. Therefore, a client-side monitoring tool should require no changes to user browsers and leverage built-in browser features for widespread deployment, thus improving detection coverage for ongoing client-side XSS attacks.

Minimal changes to websites. Minimizing changes to server-side web applications and environments is essential for facilitating the deployment of a monitoring framework. Additionally, the framework should support easy maintenance as websites evolve over time. By ensuring seamless deployment and effortless maintenance, website operators are more likely to adopt the framework, thereby improving their overall security against DOM-based XSS attacks.

Negligible overhead. This additional framework should introduce negligible overhead on website users, particularly regarding webpage loading times. It is well known that visitors tend to leave a website when it takes a long time to load; according to Google Consumer Insights, 53% of website visitors leave a webpage if it takes longer than three seconds to load [67], and the bounce probability increases by 32% when page load time extends from one to three seconds [3]. Hence, a lightweight monitoring framework is crucial.

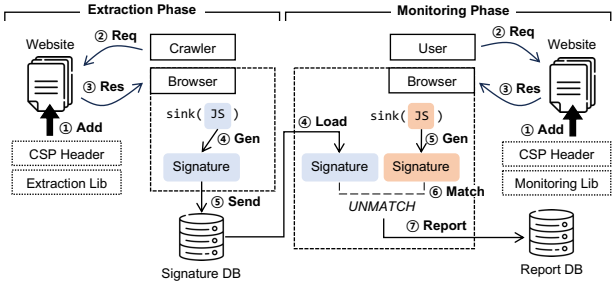


Figure 1: Overview of TrustyMon architecture.

Existing techniques, such as taint analysis and randomization, struggle to address these challenges. Taint analysis typically demands browser or website instrumentation and incurs significant overhead [55], making it unsuitable for seamless integration. Randomization techniques also face similar drawbacks regarding their deployability [4].

Our methodology. We propose TrustyMon that leverages the Trusted Types feature to address these technical challenges. By using Trusted Types, we implement an allowlist-based filtering mechanism. Since Trusted Types is a built-in browser functionality, it requires no modifications to user browsers. Moreover, to enable Trusted Types and the monitoring of XSS attacks, TrustyMon requires website operators to include a simple CSP policy and a JS library, resulting in minimal changes to websites under protection. Lastly, TrustyMon leverages the browser's built-in capability to monitor the arguments of sensitive sink functions that introduce malicious JS snippets, reducing overhead compared to previous approaches [41] that rely on customized frameworks for monitoring sensitive sinks. We also emphasize that TrustyMon supports the seamless maintenance of benign signatures (i.e., allowlists) through Report Controller, which supports the collection of new benign signatures dynamically, providing a practical monitoring framework for real-time DOM-based XSS detection.

4 Design

4.1 Overview

Figure 1 illustrates the overview of TrustyMon, which operates in two phases: *extraction* and *monitoring*.

In the extraction phase, TrustyMon extracts a set of JS signatures from a website under protection. Each signature corresponds to a benign JS snippet or URL that implements the website's functionality. To initiate this process, website operators embed a CSP header and include the TrustyMon extraction library in the target website (①). TrustyMon then crawls the website using a Chrome browser, executing existing JS snippets in the crawled webpages (②–③). Trusted Types in the crawling browser invokes the Trusted Type callbacks defined in the extraction library. These callbacks extract the JS snippets or URLs, which are dynamically injected via injection sinks, and convert them into signatures (④). These signatures are then reported to the TrustyMon signature database (⑤).

In the monitoring phase, website operators include the TrustyMon monitoring library and a CSP header in their websites to enable Trusted Types functionality (①). When a client browser requests this monitoring library, TrustyMon retrieves the set of benign JS signatures from the database and generates a JS file listing

```
Content-Security-Policy-Report-Only: require-trusted-types-for 'script';
```

Listing 3: CSP for Trusted Types used in TrustyMon.

the benign signatures (④). During this phase, the Trusted Type callbacks defined in the monitoring library inspect each dynamically inserted JS snippet prior to execution (⑤) and match its JS signature against the benign signatures (⑥). TrustyMon detects any attempts to inject JS snippets that do not have a matching signature in the collected set of benign signatures. It reports these detected attempts to a designated reporting endpoint, notifying website operators of any JS injection attempts (⑦).

Advantages. TrustyMon leverages CSP and Trusted Types which are supported by Chromium-based browsers such as Chrome, Edge, Opera, Samsung Internet, and UC Browser. This requires no modifications to client browsers, thereby addressing the first challenge (§3.1). Given that these browsers make up 76.78% of the browser market [52], we believe that TrustyMon provides substantial coverage for detecting DOM-based XSS attacks targeting general users. As more browsers adopt Trusted Types, the coverage of TrustyMon will expand; for instance, Mozilla has announced plans to implement Trusted Types in Firefox [8].

Moreover, since TrustyMon harnesses built-in browser features, it does not introduce additional overhead during webpage rendering. We demonstrate that TrustyMon incurs an average latency of 27.02 ms, imposing negligible overhead on webpage loading times (§5.4.1), which overcomes the third challenge.

Lastly, TrustyMon automatically extracts benign JS signatures from the target website and monitors dynamically inserted JS snippets using JS libraries that website operators can trivially append to their websites. This requires no changes to server-side web application logic, thus addressing the second challenge.

4.2 Extraction Phase

The extraction phase collects signatures of benign JS snippets or URLs that implement the functionalities of the target website. This phase compiles a list of benign signatures representing the JS snippets or URLs that website operators intend to run in client browsers.

Given a target website under protection, web operators place a CSP header and an extraction JS library into the website. This CSP activates the default policy of Trusted Types, and the extraction library hooks all invocations of JS injection sinks.

Listing 3 describes the CSP that TrustyMon leverages to enable a browser to check the Trusted Types of argument strings passed into injection sinks. It also allows the browser to report Trusted Type violations to a specified endpoint. The extraction library defines Trusted Type callbacks for all Trusted Types (i.e., `TrustedHTML`, `TrustedScript`, and `TrustedScriptURL`). In these callbacks, TrustyMon extracts dynamically injected JS snippets and computes their signatures.

After the initial setting, TrustyMon crawls webpages and executes JS snippets using a Chrome browser. During this crawling process, the browser invokes a Trusted Type callback for each injection sink invocation. The invoked callback extracts a JS signature from the argument string of the invoked injection sink and reports the signature to a designated endpoint. This endpoint records the received signature, the URL where the violation occurs, and the

Algorithm 1: Trusted Type Callback

```
1  Procedure TrustedTypeCallback(sink, arg)
2      if sink.type == HTML then
3          C ← ExtractScriptOrURL(arg, [])
4      else if sink.type == Script or sink.type == ScriptURL then
5          C ← [arg]
6      signature ← []
7      foreach c ∈ C do
8          if c.type == Script then
9              signature.append(GetTypesAndValuesFromAST(c))
10         else if c.type == URL then
11             signature.append(c.scheme + c.domain + c.path)
12     signature ← Hash(signature)
13     if ExtractionPhase then
14         SendSignature(signature)
15     else if MonitoringPhase then
16         if signature ∈ allowlist then
17             TrustedType(arg)
18         else
19             SendReport(signature, sink, arg)
20 Procedure ExtractScriptOrURL(arg, C)
21     foreach e ∈ DOMParse(arg) do
22         c ← ExtractExecutableContentsFromRule(e)
23         if c.type == HTML then
24             C.append(ExtractScriptOrURL(c, C))
25         else if c.type == Script or c.type == URL then
26             C.append(c)
27     return C
```

location of the injection sink. We describe the crawler and the Trusted Type callbacks in more detail in the following sections.

4.2.1 Crawler. Given a website URL and test accounts, TrustyMon systematically explores the website. When visiting each webpage, the crawler extracts `<a>` tags and their `href` attributes to collect URLs for further exploration. Additionally, TrustyMon simulates user behavior by identifying `<form>` tags and their child elements that accept user input, such as `<input>`, `<textarea>`, and `<select>`. For each user-controllable HTML element, the crawler inputs randomly generated strings or interacts with various options, including checkboxes, radio buttons, and dropdown menus. It then submits the form by clicking the submit button. This process continues until all collected URLs have been visited or a timeout occurs. Note that deploying diverse crawlers, such as Crawljax [10] or monkey testing [31], can help expand the extraction coverage.

4.2.2 Trusted Type Callback. During the website crawling process, benign JS snippets are executed, allowing the crawling browser to invoke the defined Trusted Type callbacks with the actual argument strings used at DOM injection sinks. In these callbacks, TrustyMon extracts a JS snippet or URL from each string argument and then computes a signature based on this snippet.

Based on the type of an invoked injection sink, the corresponding Trusted Type callback defined in the Trusted Type policy is invoked. There are three types of injection sinks that render arguments as HTML (HTML), execute arguments as scripts (Script), and treat arguments as URLs for external script resources (ScriptURL), respectively. Examples of each type of injection sink supported by Trusted Types are provided in Appendix §C. For each of these injection sink types, the corresponding Trusted Type callback (i.e., `createHTML`, `createScript`, or `createScriptURL`) is defined in our Trusted Type policy in the TrustyMon extraction library. Listing 9 shows the pseudocode of TrustyMon's Trusted Types policy. Algorithm 1 describes the process of these defined callbacks.

Table 1: Matching rules for extracting executable contents.

Matching rules
#1. <script src="[URL]"></script>
#2. <script>[JS]</script>
#3.
#4. <form action="javascript:[JS]"></form>
#5. <input formAction="javascript:[JS]"></input>
#6. <button formAction="javascript:[JS]"></button>
#7. <iframe srcdoc="[HTML]"></iframe>
#8. <*element event*="[JS]"></*element*>

Script/URL extraction. The Trusted Type callback extracts a JS snippet or URL from the string passed to an injection sink (Lns 2–5). When an HTML sink is invoked, the callback parses the argument into a DOM tree (Ln 21). It then traverses the elements of this DOM tree to identify executable components that can contain JS snippets, such as HTML, JS, and URL, by matching each element against the rules specified in Table 1. If a match is found, the callback extracts the content enclosed in square brackets, such as [HTML], [JS], or [URL] (Ln 22). If the type of the extracted contents is HTML, the same procedure is applied recursively (Lns 23–24). Otherwise, the extracted components are fed into the next signature generation step. When a Script or ScriptURL sink is invoked, TrustyMon skips the process of extracting executable contents because the actual argument already contains JS code. (Lns 4–5).

Note that Table 1 excludes certain executable contents triggered by `javascript:`, which can be executed without user interaction, such as `<iframe src="javascript:[JS]"></iframe>`. Instead, these JS components are extracted when a Script sink, such as `javascript:` [27], is invoked.

Signature generation. The Trusted Type callback generates a signature using the extracted JS snippet or URL (Lns 6–12). For a JS snippet, it generates an abstract syntax tree (AST) using Acorn [1] and extracts the AST node types and the AST node values. Specifically, TrustyMon traverses the AST in a pre-order walk, capturing a string of each AST node type (e.g., Literal, Identifier, or CallExpression) and the corresponding node value (e.g., `eval`, `window`, or `document`). When extracting node values, TrustyMon only considers the values of identifier nodes, which represent built-in objects or functions. By ignoring other values, TrustyMon accommodates dynamically changing values, such as timestamps, while considering built-in objects helps protect against isomorphism attacks (§5.2). The resulting sequence of AST node types and values forms a JS signature (Lns 8–9). Similarly, for a URL, TrustyMon computes its signature by extracting the scheme, domain, and path components (Lns 10–11). All generated signatures are then hashed (Ln 12) and sent to the defined endpoint (Ln 14). The signature database stores these signatures without duplicates.

JS obfuscation. By design, TrustyMon supports signature extraction from obfuscated JS code. For example, Listing 4 shows obfuscated JS code dynamically injected into `sc.text` (Ln 2). This

```
1 const sc = document.createElement("script");
2 sc.text = "(function(){const _0xf1=function(_0x9a){
   return btoa(_0x9a);};const _0x3c=function(_0x1,_0x2)
   {return _0x1+'\x3A'+_0x2;};this['\x67\x65\x6E\x54\
   x6F\x6B']=function(_0x7b){const _0x99=Date['\x6E\x6F
   \x77']();return _0xf1(_0x3c(_0x7b,_0x99));};})();";
3 // Signature: [{"type":"Id"},{"type":"Id"},{"type":"
   Id","name":"btoa"},{"type":"Id"},{"type":"CallExpr
   ","optional":false},{"type":"ReturnStat"},{"type":"
   BlockStatement"}, ..., {"type":"ExprStat"}]
```

Listing 4: Obfuscated JS injected via Script sink.

obfuscated code is to generate an authorization token. Since this assignment invokes a Script sink, TrustyMon computes a signature of the obfuscated JS code and stores it along with its hash. Ln 3 shows an abbreviated signature string of the obfuscated code. Later, TrustyMon checks whether observed signatures in the monitoring phase match this extracted signature (§4.3).

However, TrustyMon's extraction phase can struggle with injected scripts that change their AST structure on each load. For instance, if a website injects obfuscated JS code into XSS injection sinks, and the obfuscation varies on each page load while altering its AST structure, TrustyMon generates a different signature for each obfuscated JS code, potentially generating false reports. We further discuss this limitation in §6.

4.3 Monitoring Phase

The monitoring phase aims to detect JS injection attempts at runtime. This phase begins by placing a CSP header that enables Trusted Types and a monitoring library. Website operators add these two components to the website under protection.

The monitoring JS library defines Trusted Type callbacks, which function similarly to those used in the extraction phase (§4.2.2). The key difference is that, in the monitoring phase, the callbacks check whether the current JS snippet to be inserted matches any of the benign JS signatures extracted in the previous phase (§4.2).

Specifically, each Trusted Type callback in the monitoring library extracts a JS snippet or URL from the argument string and computes its signature using the same procedure outlined in §4.2.2. It then proceeds to the matching process.

When a matching signature is found among the benign signatures, the monitoring JS library converts the argument string into a Trusted Type object (Lns 16–17), which allows the corresponding JS snippet to be inserted into the webpage. However, if there is no matching signature, the injection sink receives the untrusted string, resulting in a CSP violation (Lns 18–19). The monitoring library reports this injection attempt, which could indicate a DOM-based XSS attack. TrustyMon stores the CSP report in the report database with the timestamp and IP address where the violation occurred.

Listing 5 shows an example of a violation report. The original CSP report provides the data as listed in Lns 2–5, including the location of the invoked sink (Lns 2–4), and the first 40 bytes of the inline script that caused the violation (Ln 5). Additionally, we include further data as shown in Lns 6–10, which includes the actual argument string passed to the injection sink (Ln 6), the signature of the JS snippet (Ln 8), the injection sink, and its type (Lns 9–10).

We note that TrustyMon is designed to monitor JS execution and send violation reports to a designated endpoint, helping website

```
1  "csp-report": {
2   "line-number": 33,
3   "column-number": 8,
4   "source-file": "http://example.com/test_eval",
5   "script-sample": "eval|fetch('//attacker.com', {
     method: 'POST',",
6   "content": "fetch('//attacker.com', {method: 'POST',
      body: document.cookie})",
7   "domain": "http://example.com/test_eval",
8   "hash": "247c7c04f1ebbb5505b4113e85148d12f7a0f13285
     2d332771aa457c38cc5b8b",
9   "sink": "eval",
10  "type": 2
11 }
```

Listing 5: TrustyMon violation report example.

operators identify ongoing XSS attacks and locate XSS vulnerabilities. Since TrustyMon leverages the `Content-Security-Policy-Report-Only` header (Listing 3), it does not block any JS execution or disrupt original website functionalities.

Blocking mode. We extend TrustyMon to support a blocking mode, which prevents the execution of JS scripts whose signatures are not among the benign JS signatures. In this mode, TrustyMon actively blocks XSS attempts by enforcing a different Trusted Type policy. In this Trusted Type policy, when a signature mismatch occurs, TrustyMon not only sends a violation report but also sanitizes the arguments passed to the injection sinks, preventing further execution. For each injection sink type, TrustyMon applies tailored sanitization: for HTML, it uses DOMPurify [21] to sanitize untrusted content, while for JS and URLs, it drops untrusted strings entirely. Accordingly, sanitized arguments do not trigger JS execution, effectively blocking DOM-based XSS attempts. However, this blocking mode is also possible to block legitimate JS execution when its signature is not listed as benign. In §5.4.2, we analyze the functional breakages caused by TrustyMon in its blocking mode.

4.4 Benign Signature Maintenance

TrustyMon provides a mechanism for registering new benign JS signatures using the violation reports stored in the report database. To facilitate this, we implemented Report Controller, which allows website operators to register the JS signatures from selected violation reports. Report Controller enables TrustyMon to address false reports caused by an incomplete set of benign JS signatures collected during the extraction phase, by simply adding the signature from a reported violation to the list of benign signatures. Additionally, Report Controller plays a crucial role in maintaining TrustyMon when a website under protection updates its web application or embedded JS libraries. When new code is introduced, web operators can trigger the new code using manual tests or end-to-end automatic testing. If this new code invokes injection sinks with executable JS snippets, TrustyMon generates violation reports and sends them to the designated server. Website operators then review these reports by examining the JS code included in each violation report, as shown in Listing 5. They can select the relevant JS signatures from these reports and register them as benign signatures using Report Controller.

5 Evaluation

We evaluated TrustyMon to measure its efficacy in detecting XSS attacks (§5.2) and overheads that TrustyMon imposes on server-side

website operators (§5.3) and client-side (§5.4). We also assessed its effectiveness on the Tranco Top 1,000 websites (§5.6) and explored the feasibility of extending its monitoring capability to non-Chromium browsers using a polyfill (§5.5).

5.1 Experimental Setup

Benchmarks. To evaluate TrustyMon, we collected publicly disclosed DOM-based XSS vulnerabilities. From the CVE database [9], we identified approximately 190 CVE numbers associated with the keyword "DOM XSS", reported between 2012 and 2024. We then narrowed them down to about 90 vulnerabilities discovered in open-source web applications that have received at least 100 stars on GitHub. Excluding web applications that were uninstallable or had irreproducible vulnerabilities due to unclear instructions or errors, we finally selected 19 vulnerabilities from 16 web applications.

These 16 web applications were implemented in various programming languages, including PHP, JS, Go, and Python. The scale of these applications ranged from 0.2K to 3.6M lines of code (LoC). We also considered the popularity and adoption of these applications; their GitHub repositories have received a minimum of 140 to 40.7K stars, and their plugins have been downloaded at least 10 million times. Table 5 provides detailed statistics of each benchmark application.

Attack payloads. In the subsequent evaluations, we assumed a website operator seeking to deploy TrustyMon in each of these web applications. For each vulnerability, we devised three to five attack payloads designed to exploit the vulnerability. These payloads were crafted by referencing the OWASP XSS cheat sheet, which compiles a comprehensive list of XSS attack vectors and filter evasion techniques [43]. Each payload exploits a different attack vector involving filter bypass methods or malformed tags. For the 19 vulnerabilities, we prepared a total of 75 attack payloads.

ScriptProtect. We compared TrustyMon to ScriptProtect [41], which is the only DOM-based XSS detection tool with a publicly available implementation for which we were able to reproduce their reported results. ScriptProtect instruments injection sink APIs and allows trusted first-party code. To determine whether the code is from the first party, it inspects the stack trace of the current execution thread and checks if the call was initiated by the trusted first party. If so, ScriptProtect allows the execution by passing the unaltered value to the API. If the call originated from a third-party script, the value is sanitized. We deployed ScriptProtect on our benchmark applications using the version available in its GitHub repository [40]. We only modified the `htmlSinksOnly` option to monitor script injection sinks in addition to HTML injection sinks.

Environment. Our experiments were conducted on a Linux workstation equipped with two Intel Xeon E5-2620 v4 CPUs and 128 GB of RAM. We also prepared Docker containers to host the web applications. For the browser, we employed Chrome version 126.

5.2 Effectiveness

We assessed the capabilities of TrustyMon in detecting DOM-based XSS attacks in real time. For this evaluation, we considered a scenario in which a website operator adopts the default configuration of TrustyMon without any manual effort in collecting benign signatures. By default, TrustyMon collects signatures by crawling webpages with a timeout of 60 minutes and a crawl depth of 10 levels, allowing it to crawl only pages reachable within a specified number of clicks or levels from the starting page. For each web application, we conducted the extraction phase of collecting benign signatures and then deployed the TrustyMon monitoring library to webpages under protection to detect DOM-XSS attacks. We tested both the monitoring and blocking modes of TrustyMon and found no difference in their ability to detect ongoing attacks.

Table 2: Evaluation of TrustyMon on 16 real-world applications in detecting XSS attacks.

Application	Signatures	CVE	Sink	Detection	
				TM*	SP*
Elementor	57 (0/36/21)	2021-24891	innerHTML	✓	✗
		2022-29455	setAttribute	✓	✗
Yoast SEO	4 (0/3/1)	2012-6692	innerHTML	✓	✗
MyBB	0 (0/0/0)	2020-15139	innerHTML	✓	✗
Microweber	15 (8/7/0)	2022-0698	innerHTML	✓	✗
OpenEMR	13 (1/12/0)	2022-2729	innerHTML	✓	✗
AbanteCart	2 (0/2/0)	2021-42050	innerHTML	✓	✓
Apache Answer	10 (0/0/10)	2023-0741	innerHTML	✓	✗
draw.io	16 (0/0/16)	2022-3873	innerHTML	✓	✗
URL Pages	0 (0/0/0)	2024-26468	write	✓	✗
Railroad-diagram	1 (0/1/0)	2024-26467	eval	✓	✗
Web-platform-tests	11 (0/1/10)	2024-26466	eval	✓	✗
Beep.js	1 (0/1/0)	2024-26465	eval	✓	✗
CesiumJS	342 (2/38/302)	2023-48094	innerHTML	✓	✗
LibreNMS	1 (0/1/0)	2023-5060	innerHTML	✓	✗
Cacti	11 (7/3/1)	2023-39360	href (location)	✓	✗
Pimcore	62 (0/40/22)	2023-1517	innerHTML	✓	✗
		2023-2343	innerHTML	✓	✗
		2023-2614	innerHTML	✓	✗

* TM: TrustyMon, SP: ScritpProtect

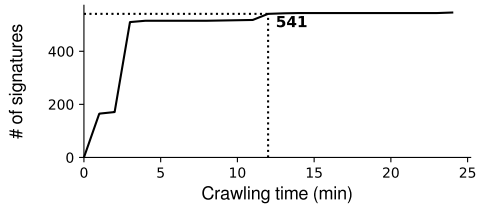


Figure 2: Cumulative number of signatures over the elapsed crawling time.

Signatures. Table 2 shows the experimental results. The "Signatures" column presents the number of signatures collected during the extraction phase of TrustyMon. The numbers in parentheses indicate the number of signatures extracted from each Trusted Type of XSS injection sink (i.e. HTML, Script, and ScriptURL).

TrustyMon collected a total of 546 signatures, including 18 HTML (3.30%), 145 Script (26.56%), and 383 ScriptURL (70.15%) signatures. Out of these, 342 (62.64%) were extracted from CesiumJS, which loads 297 third-party libraries. On average, 34.13 signatures were collected from each application.

Figure 2 illustrates the cumulative number of extracted signatures over the crawling time for all applications. Notably, 99% of the signatures were collected within the initial 12 minutes of crawling across all applications.

Attack detection. In Table 2, the third column represents the CVE identifiers of DOM-based XSS vulnerabilities. The fourth column indicates the injection sink that triggers each vulnerability, including `innerHTML`, `setAttribute`, `write`, `eval`, and `href`.

```
1  function (_bundle, _checkForLegacyModules, _mid, _amdValue) {
2    var define = function(mid, factory) {
3      define.called = 1;
4      _amdValue.result = factory || mid; },
5    require = function() {define.called = 1;};
6    try {
7      define.called = 0;
8      eval(_bundle);
9      if (define.called == 1) return _amdValue;
10     if (_checkForLegacyModules = _checkForLegacyModules(_mid))
11       return _checkForLegacyModules;
12   } catch (e) {}
13   try {return eval('(' + _bundle + ')');}
14   catch (e) {return e;}
15 }
```

Listing 6: Benign JS snippet from CesiumJS.

The last two columns describe whether TrustyMon and ScriptProtect detected XSS attacks exploiting the corresponding vulnerability in the CVE column, respectively. As the "TM" column shows, TrustyMon successfully identified all XSS attacks that exploited 19 DOM-based XSS vulnerabilities. For 13 attack payloads, TrustyMon detected them because they had different signatures from the extracted benign ones. The remaining payloads were detected because there were no benign signatures for the corresponding injection sinks, which either are not intended to accept JS code or are not covered in common use cases.

In the "SP" column, the green check mark indicates that ScriptProtect detected and blocked XSS attacks. ScriptProtect identified only one vulnerability in AbanteCart. The 13 orange X marks indicate that ScriptProtect checked the injection sinks but allowed the XSS attacks since it determined they came from first-party code. Among the five red X marks, the three vulnerabilities triggered with `eval` were not detected since ScriptProtect has limitations in instrumenting the `eval` function. The vulnerability in Elementor was not detected because ScriptProtect missed the inspection of the `setAttribute` method when setting values of `onXXX` events, such as `onload`, `onclick`, and `onerror`. For the vulnerability in Cacti, we were unable to trigger the vulnerability due to the compatibility of ScriptProtect. When ScriptProtect was applied, certain buttons in Cacti were not displayed, preventing the vulnerability from being triggered. These experimental results demonstrate the capability of TrustyMon to collect benign JS signatures without manual effort and its efficacy in identifying DOM-based XSS attacks without false negatives.

Signature isomorphism attacks. Recall that TrustyMon generates benign signatures extracted from the AST node types (e.g., CallExpression, Identifier) and AST node value for built-in objects and functions (e.g., `window`, `eval`) of JS snippets (§4.2.2). The adversary is thus able to exploit this scheme by conducting JS isomorphism attacks [12], injecting a malicious JS snippet of which the signature matches a benign one. Therefore, we evaluated the robustness of our signature scheme against JS isomorphism attacks.

Specifically, we used HideNoSeek, a generic isomorphism attack tool proposed by Fass *et al.* [11, 12]. It modifies the AST of a malicious JS snippet to mimic the syntax of a given benign one. We prepared five malicious JS snippets with distinct AST structures, varying from alerting strings to stealing cookies. We also gathered all benign JS snippets that TrustyMon collected from each application. We then searched for isomorphic subgraphs between the malicious and benign ASTs and replaced benign sub-ASTs with malicious ones using HideNoSeek.

Table 3: False violation reports when simulating website visitors using Crawljax.

Applications	# of Requests	Elapsed time	# of False reports
Elementor	212	40s	0
Yoast SEO	424	6m 9s	0
MyBB	78	20s	0
Microweber	272	3m 52s	0
OpenEMR	1,463	60m	1 (0/0/1)
AbanteCart	375	1m 15s	0
Apache Answer	653	1m 3s	2 (0/0/2)
draw.io	23	8s	0
URL Pages	52	1m 22s	0
Railroad-diagram	6	7s	0
Web-platform-tests	13,333	57m 27s	12 (1/0/11)
Beep.js	27	8s	0
CesiumJS	598	1m 6s	2 (0/0/2)
LibreNMS	109	21s	1 (1/0/0)
Cacti	56	11s	0
Pimcore	25,732	60m	0

Out of 145 Script signatures in Table 2, HideNoSeek generated 37 isomorphic JS snippets. However, we confirmed that all 37 malicious JS snippets failed to bypass TrustyMon due to mismatched AST values since HideNoSeek considers only AST types when searching for isomorphic subgraphs, not AST values for built-in objects.

Furthermore, we manually investigated whether the 145 benign JS snippets in Table 2 could be forged to include attack payloads. In CesiumJS, we managed to identify one benign JS snippet of which the signature can be exploited to match the one of a malicious JS snippet, as shown in Listing 6. The adversary is able to change the character "(" in Line 13, the argument of `eval`. Since the character is of the literal type, TrustyMon does not consider its value. Accordingly, TrustyMon generates the same signature for the benign snippet even after modifying "(" to any string including attack payloads. However, when injecting a malicious JS snippet into the `eval`, TrustyMon also matches the signatures of this JS snippet to be executed via `eval` calls to the benign ones. Thus, we were unable to program malicious JS snippets that bypass TrustyMon.

In summary, the signature scheme of TrustyMon considering AST node types as well AST node values for built-in objects and functions increases the robustness against the isomorphism attacks, effectively mitigating advanced XSS attacks.

5.3 Server-side User Experience

We evaluated how TrustyMon facilitates maintaining benign signatures using Report Controller when false reports are generated (§5.3.1) and when web applications are updated (§5.3.2).

5.3.1 False Reports. We measured how many false alarms TrustyMon reported as DOM-based XSS attacks for each benchmark application. The root causes of false attack reports stem from incomplete sets of benign signatures that were not captured during the extraction phase. To simulate user behaviors different from those executed to collect benign signatures during the extraction phase, we used an event-driven web crawler, Crawljax [10]. For each web application, we let Crawljax explore webpages with the default configuration.

Table 3 shows the experimental results. The second and third columns represent how long and how many requests were sent. On average, Crawljax sent 2,713 requests in 15.53 minutes. The number of sent requests and the elapsed testing times varied from six to

Table 4: JS signature changes over code updates of 16 web applications.

Application	Version (T_1)	Signatures (T_1)	Version (T_2)	Signatures (T_2)	Unchanged Signatures ($T_2 \rightarrow T_1$)	Version (T_3)	Signatures (T_3)	Unchanged Signatures ($T_3 \rightarrow T_1$)	Version (T_4)	Signatures (T_4)	Unchanged Signatures ($T_4 \rightarrow T_1$)
Elementor	3.4.7	57	3.4.6	57	57 (100.0%)	3.2.0	57	57 (100.0%)	3.0.12	58	57 (98.28%)
Yoast SEO	2.1	4	2.0.1	4	4 (100.0%)	1.6.3	4	4 (100.0%)	1.5.5	4	4 (100.0%)
MyBB	1.8.22	0	1.8.21	0	-	1.8.20	0	-	1.8.19	0	-
Microweber	1.3.1	15	1.3.0	15	15 (100.0%)	1.2.9	14	10 (71.43%)	1.2.3	6	3 (50.0%)
OpenEMR	7.0.0	13	6.1.0.1	16	15 (93.75%)	6.0.0.3	16	15 (93.75%)	6.0.0.2	16	15 (93.75%)
AbanteCart	1.3.1	2	1.3.0	2	2 (100.0%)	1.2.13	2	2 (100.0%)	1.2.12	2	2 (100.0%)
Apache Answer	1.0.2	10	1.0.1	10	9 (90.0%)	1.0.0	10	7 (70.0%)	-	-	-
draw.io	20.5.0	16	20.4.2	16	16 (100.0%)	17.4.3	14	9 (64.29%)	15.5.8	11	8 (72.73%)
URL Pages	035b647	0	0d82cb9	0	-	6fb7c47	0	-	52780d6	0	-
Railroad-diagram	ea9a123	1	928618	1	1 (100.0%)	c3a16b9	1	1 (100.0%)	eab712c	1	1 (100.0%)
Web-platform-tests	938e843	11	02823cf	11	11 (100.0%)	3933453	11	11 (100.0%)	e7a8ee5	11	11 (100.0%)
Beep.js	ef22ad7	1	cf4f9fd	1	1 (100.0%)	795215	0	-	-	-	-
CesiumJS	1.111	342	1.110.1	342	342 (100.0%)	1.105	339	338 (99.71%)	1.99	339	338 (99.71%)
LibreNMS	23.9.0	1	23.8.2	1	1 (100.0%)	23.2.0	1	1 (100.0%)	22.8.0	1	1 (100.0%)
Cacti	1.2.24	11	1.2.23	10	7 (70.0%)	1.2.22	16	10 (62.50%)	1.2.19	15	4 (26.67%)
Pimcore	10.5.18	62	10.5.17	62	62 (100.0%)	10.5.4	62	62 (100.0%)	10.3.1	62	62 (100.0%)
Average		34.13		34.25	33.94 (99.09%)		34.19	32.94 (96.34%)		37.57	36.14 (96.20%)

25,732 and from seven seconds to 60 minutes, depending on the target web applications.

The number of false reports in the last column indicates how many false reports TrustyMon generated. For 11 applications, TrustyMon generated no false positives, while for five applications, TrustyMon generated 18 false reports. On average, TrustyMon produced 1.13 false reports. The false reports were produced due to the limitations of the crawler. Although TrustyMon could collect more signatures when increasing the timeout of the crawler or using additional web crawlers, it is still challenging to visit all webpages and trigger available events to extract all benign signatures. To address this challenge, TrustyMon provides Report Controller to assist website operators in adding benign signatures that might not have been covered by the crawler.

Report Controller provides a web interface for maintaining signatures. After examining each report, if it is determined to be a false alarm, web operators can easily add this signature to the set of benign signatures by clicking a button, as shown in Figure 4 (Appendix D). For the above 18 false reports, we manually investigated each report and determined all to be benign signatures, which took about ten minutes for one author. It demonstrates that website operators can easily enhance their benign signatures when faced with false alarms using Report Controller.

5.3.2 Application Updates. When a target server-side application is updated, the extraction phase should be conducted again. Otherwise, TrustyMon might produce false alarms. Fortunately, the release cycle of most popular JS libraries is at least six months [22, 39]. We further validated how many signatures changed after updating the web applications in the benchmarks.

For each application released at T_1, we prepared three additional versions of the application; the immediate previous version, the version released more than six months prior, and the version released more than twelve months prior. We denoted these selected times as T_2, T_3, and T_4 in descending order. We measured how many signatures had changed over each of these time periods.

Table 4 summarizes the evaluation results. The "Version" columns represent the selected versions of web applications corresponding to each time period. For Apache Answer and Beep.js, there were no versions suitable for T_3. Thus, we selected the oldest release as T_3. For AbanteCart, there was a three-year gap between T_2 and its previous version. Thus, we set T_3 as the release prior to T_2, and T_4 as the release prior of T_3.

The "Signatures" columns represent the number of signatures collected in each version of the application. The "Unchanged Signatures" columns represent the number of signatures unchanged from T_n to T_1 and the number in parentheses indicates the corresponding percentage. For example, in the sixth column, we analyzed which signatures collected at T_2 remained in the updated application released at T_1.

On average, about 99.09%, 96.34%, and 96.20% of signatures remained unchanged from T_2, T_3, and T_4 to T_1 updates, respectively. It shows that the majority of the signatures remained unchanged. Additionally, we found that the average number of newly added signatures over a year was approximately 1.64. We expect that web operators can incorporate such minor changes using Report Controller, which reduces the burden of regenerating benign signatures.

5.4 Client-side User Experience

5.4.1 Performance Overhead. We evaluated the latency overhead that TrustyMon imposes from the perspective of clients visiting TrustyMon-protected websites. For this, we collected webpages containing injection sinks that add executable JS components for each web application. We then loaded each webpage ten times without browser caching, measuring page loading times with and without TrustyMon. Finally, we computed the average latency across webpages from each application. We found no significant latency differences between the monitoring and blocking modes of TrustyMon in detecting attacks. We thus report the latency of TrustyMon with its default monitoring mode.

Figure 3 depicts the experimental results. The gray bars represent the average page loading time in a vanilla browser (i.e., without TrustyMon or ScriptProtect). The left red and right black bars stacked on the gray bars indicate the latency introduced by TrustyMon and ScriptProtect, respectively.

Figure 3 shows that TrustyMon incurs an average latency of approximately 27.02 ms (5.68%) across the benchmark applications.

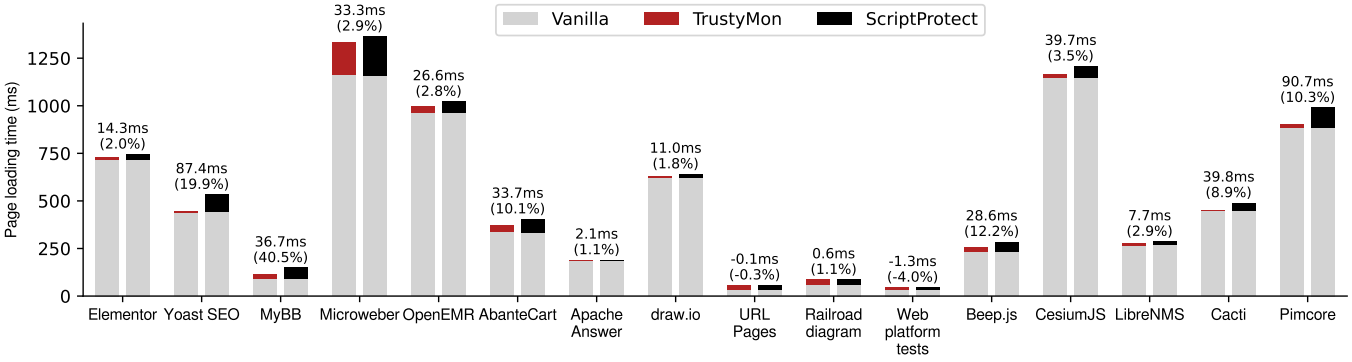


Figure 3: Latency overheads of TrustyMon and ScriptProtect.

Microweber exhibited the largest overhead, with an average page loading time increase of approximately 171.81 ms when TrustyMon was active. This is because Microweber invokes many injection sinks and the JS code passed to these sinks is relatively large, resulting in additional overhead for parsing the JS code and computing signatures. Additionally, the blocking mode incurs an average latency of approximately 26.57 ms (5.58%), which is comparable to the monitoring mode.

In Figure 3, for each application, the number between the two bars represents the loading time difference in milliseconds, and the following percentage indicates the proportion of this difference relative to the original loading time without TrustyMon. For instance, in Pimcore, the loading time difference between TrustyMon and ScriptProtect is 90.7 ms, accounting for 10.3% of the original loading time of the web application.

Overall, ScriptProtect incurred an average latency of approximately 55.19 ms (11.60%), while TrustyMon consistently produced less overhead across most applications, except for URL Pages and Web platform tests. For these two applications, the differences were minimal, with values of just 0.1 ms and 1.3 ms, respectively. TrustyMon leverages browser features to monitor injection sinks, unlike ScriptProtect which instruments sink APIs, resulting in largely lowering its overhead.

5.4.2 Functionality Breakage. TrustyMon does not cause any functional breakage in its default monitoring mode. However, in the blocking mode (§4.3), TrustyMon affects website functionalities, potentially disrupting legitimate operations. To evaluate the impact of the TrustyMon blocking mode on the functionality of target websites, we compared user experiences before and after deploying TrustyMon. Specifically, we examined differences in the rendered webpage, DOM structure, HTTP responses, and JS console logs while exploiting the vulnerabilities listed in Table 2. For 19 vulnerabilities, TrustyMon successfully blocked all attack attempts. It effectively prevented malicious payloads injection without introducing any visible functionality breakages on the rendered webpages.

While TrustyMon did not interfere with the legitimate webpage functionalities when blocking potential attacks, false alarms can introduce usability issues. To assess this impact, we conducted the same experiments described in §5.3.1. Across five applications, TrustyMon generated 18 false reports, leading to blocked legitimate JS execution and partial functionality breakages on three webpages. These breakages included incomplete message displays, slight UI rendering differences, and failed script loading within an iframe. However, by leveraging Report Controller, we registered 18 JS signatures, preventing functionality breakages. This process only took 10 minutes for one author.

5.5 Extended Browser Support via Polyfill

We assess the feasibility of deploying TrustyMon on non-Chromium browsers. The W3C provides a polyfill implementation to support Trusted Types in non-Chromium browsers, such as Firefox and Safari [61]. It is straightforward to deploy TrustyMon using this polyfill (referred to as TrustyMon$_{ex}$ in the following). Instead of inserting a CSP header, TrustyMon$_{ex}$ activates Trusted Types by including the polyfill library (trustedtypes.js) with the `data-csp` attribute as follows:

```
<script src="trustedtypes.js" data-csp="trusted-types
default; require-trusted-types-for `script`"></script>
```

Using the same implementation of Trusted Types policies as TrustyMon, we evaluated the effectiveness of TrustyMon$_{ex}$ in Firefox. During the extraction phase, TrustyMon$_{ex}$ collected 235 signatures. Some signatures could not be collected due to the polyfill's lack of support for certain injection sinks, such as `eval`, `Function()`, or `Worker()`.

In the monitor phase, TrustyMon$_{ex}$ successfully detected 16 vulnerabilities among 19. It failed to detect three vulnerabilities since the polyfill does not support `eval`. Additionally, the polyfill does not report the line and column numbers where violations occur. Despite these limitations, TrustyMon$_{ex}$ demonstrates the feasibility of deploying TrustyMon on non-Chromium browsers.

5.6 Real-world Deployment

To evaluate the efficacy of TrustyMon on real-world websites, we expanded our benchmark to include the Tranco Top 1,000 websites [28]. For each website, we artificially injected a DOM-based XSS vulnerability using a proxy server [38]. Specifically, we modified the HTTP response intercepted via the proxy to include DOM injection sinks such as `innerHTML` or `eval`.

To deploy TrustyMon for protecting these websites, we injected a CSP header and the TrustyMon libraries via the proxy. We collected benign signatures by crawling each website with a timeout of 10 minutes. We then compiled and deployed the monitoring library to detect DOM-based XSS attacks.

Out of the 1,000 websites, 273 websites blocked TrustyMon during the signature generation phase: 230 websites were inaccessible due to client-side errors (4xx), server-side errors (5xx), or unresolvable hostnames, while 40 websites were excluded for exceeding a loading time of 30 seconds. Additionally, three websites had already deployed their CSP policies that blocked the execution of the injected vulnerabilities. Note that these cases do not indicate an issue with TrustyMon itself. Previous work by Jannis *et al.* also reported a similar failure rate when crawling Tranco top websites [47], since many of these websites require specific URL paths and parameters to be accessed, such as those hosted on CDN servers.

For the remaining 727 websites, we collected a total of 19,667 signatures, comprising 465 HTML (2.36%), 6,558 Script (33.35%), and 12,644 ScriptURL (64.29%) signatures.

After loading the monitoring library, we conducted three DOM-based XSS attacks involving same-origin CSRF attacks, secret token harvesting, and cookie exfiltration. TrustyMon successfully detected all 2,181 XSS attacks without false negatives. During the attack detection phase, we measured the latency introduced by TrustyMon. On average, TrustyMon imposed an overhead of 334.68 ms (9.07%) across these websites. Since we deployed TrustyMon on real-world websites using a proxy, it incurred higher overhead compared to our benchmark evaluations.

We also evaluated how many false alarms TrustyMon reported while traversing websites using Crawljax for a maximum 10 minutes. Out of 481 websites, TrustyMon generated no false positives, while for 246 applications, it generated a total of 750 false reports. On average, TrustyMon produced 1.03 false reports. We highlight that these minor false alarms can be easily addressed by website operators by enhancing their benign signatures using Report Controller, which took us less than one minute per report.

Additionally, we analyzed how signatures evolved over time. Three months after the initial benign signature collection, we recollected benign signatures and compared them. On average, about 10 new signatures were added per website during this period. These experimental results demonstrate the efficacy of TrustyMon in detecting DOM-based XSS attacks with minimal deployment effort.

Ethics consideration. For real-world websites, we set the crawling time to 10 minutes to minimize the load on the servers. We also did not directly attack the websites. Instead, we artificially injected DOM injection sinks via a proxy to simulate attack scenarios.

6 Discussion

Comparison with CSP and Trusted Types. The disparities between these security mechanisms stem from differences in their policy granularity. CSP relies on origins to define allowed sources for fetching web resources. Accordingly, DOM-based XSS prevention is enforced based on the origin sources of these resources, which introduces difficulties in composing finer-grained policies. For example, a CSP may allow a third-party script to execute arbitrary JS snippets that contain a DOM-based XSS vulnerability. If so, the adversary is able to certainly exploit this vulnerability even with the presence of CSP [53, 57].

In contrast, TrustyMon and Trusted Types are able to mitigate this threat. TrustyMon offers a finer-grained policy framework based on JS and URL signatures, enabling it to detect script injection attempts even from sources permitted by a CSP. Trusted Types takes this a step further by allowing the use of programmed types to specify permitted JS execution. The difference between the difficulty and engineering costs of composing a CSP policy versus a TrustyMon policy is arguable. Nonetheless, both are relatively easier than deploying Trusted Types because implementing a Trusted Types policy and enforcing this policy in web applications necessitates significant engineering effort in making modifications to existing JS code [63].

CSP compatibility. TrustyMon is compatible with existing CSPs. TrustyMon requires an existing CSP to only include the `require-trusted-types-for` directive, which enables website operators to easily extend their security protections. Also, TrustyMon is capable of detecting JSONP XSS and XSS attacks that exploit vulnerabilities introduced by trusted third-party JS snippets [64], thereby addressing security gaps that existing CSPs may leave exposed.

Browser coverage. Safari and Firefox currently do not support Trusted Types. However, Mozilla has announced plans to implement Trusted Types in Firefox [8]. Meanwhile, TrustyMon is able to extend its support to non-Chromium browsers by using a polyfill that implements Trusted Types [61], as demonstrated in §5.5. When deploying TrustyMon with the polyfill, it is still able to detect DOM-based XSS attacks while it loses the ability to monitor JS injection attempts exploiting the `eval` function and to report the exact line and column numbers where violations occur. Additionally, we investigated the impact of the lack of support for `eval`. While collecting 19,667 signatures from real-world websites (§5.5), we found that the `eval` function was invoked in 191 scripts. However, we believe that deploying TrustyMon with the polyfill remains a promising approach for monitoring DOM-based XSS attacks, considering that ScriptProtect also lacks support for `eval`. Furthermore, even without the polyfill, TrustyMon still covers 77% of browser users (i.e., Chromium-based browser users).

Polluting benign signatures. We considered a scenario in which an abusive third-party library attempts to change the benign JS signatures that TrustyMon leverages. To mitigate this threat, we encompassed the monitoring library within JS closures and leveraged Immediately Invoked Function Expression (IIFE) [34] to access TrustyMon objects. Moreover, we froze the JS object holding benign JS signatures to make it immutable in runtime.

Limitations in URL Signatures. TrustyMon uses the URL argument of ScriptURL injection sinks as its signature, without inspecting the content of the scripts. This approach introduces two limitations. First, false reports of DOM-based XSS attacks can occur when the URL changes each time the website loads, particularly if the URL contains random or hash values. For example, URLs like `ondemand.s.25603eca.js` and `ondemand.s.4e5a585a.js` may trigger false reports. To prevent this, web operators are encouraged to define patterns for trusted URLs.

Second, certain attack scenarios could bypass this approach by modifying the content of the script without changing the URL. This could occur if the server hosting the script is compromised or if an attacker injects malicious code through vulnerabilities in the script-serving process. In the first scenario, we assume that all servers hosting scripts for the target application are trusted during the signature extraction phase, and thus, attacks involving server

compromises are considered outside the scope of our system. In the second scenario, scripts loaded over HTTPS are protected from man-in-the-middle (MITM) attacks, ensuring that their content cannot be altered during transit. However, scripts delivered over HTTP remain vulnerable to interception and modification before reaching the target application. To mitigate these risks, we recommend that all server hosting scripts be configured to use HTTPS.

Maintenance of TrustyMon. By leveraging the browser's built-in feature, Trusted Types, TrustyMon reduces the burden on website operators to maintain TrustyMon as browsers evolve. Specifically, when new methods that could serve as DOM-based XSS injection sinks become available, Chrome updates Trusted Types to support these new injection sinks. For example, Trusted Types can monitor methods like `Element.setHTMLUnsafe()`, which were introduced in Chrome in April 2024 [7].

However, since TrustyMon is based on an allowlist of signatures, website operators should maintain a complete and up-to-date signature allowlist to avoid false reports as websites evolve or when uncovered signatures are found. To simplify allowlist management, TrustyMon provides Report Controller, making it easy for operators to add new signatures from violation reports and increase its coverage on benign signatures.

Additionally, when browsers introduce new elements or attributes that can contain JS components, or when new attack vectors emerge for injecting JS snippets, the matching rules in Table 1 should cover these new attack vectors to provide comprehensive monitoring of DOM-based XSS attack attempts. We believe that updating matching rules is straightforward, as it only demands adding new element tags or attribute names.

Impact of obfuscated code. From the Tranco 1,000 websites, we identified seven websites where obfuscated JS code was injected into XSS sinks. TrustyMon remains robust against obfuscation as long as the obfuscated JS code remains consistent between the extraction and monitoring phases. During extraction, TrustyMon records signatures of obfuscated JS code passed as actual arguments to XSS injection sinks. In the monitoring phase, it matches the executed obfuscated JS code against these registered signatures. However, TrustyMon may generate false reports if a website (1) dynamically generates obfuscated JS code that changes its AST structure on each load and (2) executes this code via XSS injection sinks. Among the Tranco Top 1,000 websites, we observed only one instance where dynamically injected JS code exhibited variable obfuscation in an XSS injection sink. This suggests that such dynamic injection of variably obfuscated JS is relatively rare in practice. We emphasize that this limitation arises from the absence of consistent JS signatures rather than JS obfuscation itself.

7 Related Work

DOM-based XSS vulnerability detection. Prior research has focused on preventing DOM-based XSS vulnerabilities by mitigating the occurrences of attacks in the first place [21, 41, 62] or identifying the vulnerabilities before deployment [5, 29, 35, 36, 44, 46]. DOMPurify sanitizes a given input string by removing dangerous HTML code that does not appear in the allowlist [21]. Musch *et al.* [41] and Wang *et al.* [62] introduced secure APIs to replace native APIs that are vulnerable to DOM-based XSS. They instrumented native APIs to prevent the injection of untrusted scripts. Several studies proposed approaches that detect DOM-based vulnerabilities by identifying potential vulnerable candidates and generating attack payloads using dynamic taint analyses [5, 29, 35] and hybrid analyses [44]. Instead of generating exploits, DexterJS generates patches for the detected DOM-based XSS vulnerabilities using dynamic taint analysis [46]. Melicher *et al.* trained machine learning classifiers to analyze JS functions and predict whether they are vulnerable to DOM-based XSS attacks [36]. However, these approaches still leave users unprotected from unknown vulnerabilities and new attack vectors, which TrustyMon is able to mitigate.

DOM-based XSS attack monitoring. Dynamic monitoring approaches can serve as the second line of defense by detecting ongoing XSS attacks in real time [4, 16, 17, 24, 39, 55, 59]. Previous methods have employed allowlist-based filtering methods to dynamically detect XSS attacks. These approaches collect an allowlist before deployment and then check whether the script contents to be executed exist in the allowlist by using a proxy [59], instrumenting the JS engine [39], or monitoring HTTP traffic [16, 17]. Stock *et al.* identified attacker-controllable data using character-level taint tracking and checked whether the tainted data changed the execution flow [55]. xJS detects XSS attacks using randomization [4]. For JS snippets, the modified web server applies XOR operation with an isolation key and Base64 encoding. Conversely, the modified web browser decodes all scripts using Base64 and applies XOR operation with a de-isolation key. However, these approaches require significant modifications to server-side web applications or browser changes, or introduce non-negligible execution overhead, making them challenging to deploy on real-world websites.

Additionally, CSP can detect XSS attacks by specifying which content is allowed to load and execute on each page [37, 51]. However, crafting and maintaining accurate CSP policies is challenging, as websites become more complex and the CSP specification evolves [48, 64, 66]. To facilitate deploying CSP, researchers have proposed approaches for automatically generating CSP [13, 45].

8 Conclusion

We presented TrustyMon, a dynamic monitoring framework capable of accurately detecting DOM-based XSS injection attacks. TrustyMon leverages Trusted Types to dynamically monitor any DOM-based XSS injection attempts at client-side user browsers. TrustyMon checks for the existence of the signatures of injected scripts to be executed among the benign JS signatures and reports any violations. Our evaluation results demonstrate that TrustyMon is capable of accurately detecting DOM-based XSS attacks while imposing negligible performance without requiring any changes to existing server-side web applications or web browsers. TrustyMon is readily deployable on any websites to enable the detection of DOM-based XSS attacks while imposing only light deployment costs and yielding huge security advantages.

Acknowledgments

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT). (No. RS-2023-NR076965)

References

[1] AcornJS. 2024. Acorn: A small, fast, JavaScript-based JavaScript parser. https://github.com/acornjs/acorn.

[2] Alexis Deveria. 2024. Can I use: Trusted Types for DOM manipulation. https://caniuse.com/trusted-types.

[3] Daniel An. 2018. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks.

[4] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P Markatos, and Thomas Karagiannis. 2010. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the USENIX Conference on Web Application Development*.

[5] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. 2021. Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis. In *Proceedings of the European Workshop on Systems Security*. 27–33.

[6] Chromium. 2019. Commit: Make trustedTypes() available on ExecutionContext. https://github.com/chromium/chromium/commit/a9f3bdb3d19326f510923daa08bd05e9e3e5fb7a.

[7] Chromium. 2024. Commit: Protect new setHTMLUnsafe and parseHTMLUnsafe methods with trusted types. https://github.com/chromium/chromium/commit/5f9c98130587c76019cb2692d0c850e689b4b3d2.

[8] Thomas Claburn. 2023. Mozilla decides Trusted Types is a worthy security feature. https://www.theregister.com/2023/12/21/mozilla_decides_trusted_types_is.

[9] The MITRE Corporation. 2023. CVE. https://cve.mitre.org.

[10] Crawljax. 2023. Crawljax. https://github.com/crawljax/crawljax.

[11] Aurore Fass. 2020. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. https://github.com/Aurore54F/HideNoSeek.

[12] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1899–1913.

[13] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. 2015. AutoCSP: Automatically retrofitting CSP to web applications. In *Proceedings of the International Conference on Software Engineering*. 336–346.

[14] Google. 2024. The Chromium Projects. https://source.chromium.org/chromium/chromium/src/.

[15] Charu Gupta, Rakesh Kumar Singh, and Amar Kumar Mohapatra. 2022. GeneMiner: a classification approach for detection of XSS attacks on web services. *Computational Intelligence and Neuroscience* (2022).

[16] Shashank Gupta and Brij Bhooshan Gupta. 2016. XSS-immune: a Google chrome extension-based XSS defensive framework for contemporary platforms of web applications. *Security and Communication Networks* 9, 17 (2016), 3966–3986.

[17] Shashank Gupta, Brij Bhooshan Gupta, and Pooja Chaudhary. 2018. Hunting for DOM-Based XSS vulnerabilities in mobile cloud-based online social network. *Future Generation Computer Systems* 79 (2018), 319–336.

[18] Guy Podjarny (Snyk). 2017. XSS Attacks: The Next Wave. https://snyk.io/blog/xss-attacks-the-next-wave.

[19] HackerOne. 2024. The HackerOne Top 10 Vulnerability Types. https://www.hackerone.com/top-ten-vulnerabilities.

[20] HAHWUL. 2021. History of OWASP TOP 10. https://www.hahwul.com/cullinan/history-of-owasp-top-10.

[21] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. Dompurify: Client-side protection against xss and markup injection. In *Proceedings of the European Symposium on Research in Computer Security*. 116–134.

[22] Akinori Ihara, Daiki Fujibayashi, Hirohiko Suwa, Raula Gaikovina Kula, and Kenichi Matsumoto. 2017. Understanding when to adopt a library: A case study on ASF projects. In *Proceedings of the IFIP International Conference on Open Source Systems*. 128–138.

[23] Invicti. 2021. The Invicti AppSec Indicator Spring 2021 Edition: Acunetix Web Vulnerability Report. https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021.

[24] Junaid Iqbal, Ratinder Kaur, and Natalia Stakhanova. 2019. PoliDOM: Mitigation of DOM-XSS by detection and prevention of unauthorized DOM tampering. In *Proceedings of the International Conference on Availability, Reliability and Security*. 1–10.

[25] Krzysztof Kotowicz. 2021. Trusted Types - mid 2021 report. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/2cbfffc0943dabf34c499f786080ffa2cda9cb4c.pdf.

[26] Krzysztof Kotowicz (W3C). 2024. Trusted Types. https://w3c.github.io/trusted-types/dist/spec.

[27] Krzysztof Kotowicz (W3C). 2024. Trusted Types - DOM XSS injection sinks. https://w3c.github.io/trusted-types/dist/spec/#dom-xss-injection-sinks.

[28] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the Network and Distributed System Security Symposium*.

[29] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1193–1204.

[30] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. 2013. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications* 36, 1 (2013), 16–24.

[31] Marmelab. 2022. gremlins.js. https://github.com/marmelab/gremlins.js.

[32] MDN. 2023. CSP: require-trusted-types-for. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for.

[33] MDN. 2023. CSP: trusted-types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types.

[34] MDN. 2024. IIFE. https://developer.mozilla.org/en-US/docs/Glossary/IIFE.

[35] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *Proceedings of the Network and Distributed System Security Symposium*.

[36] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. 2021. Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning. In *Proceedings of the Web Conference*. 2684–2695.

[37] Mike West, Antonio Sartori (W3C). 2024. Content Security Policy Level 3. https://w3c.github.io/webappsec-csp.

[38] Mitmproxy Project. 2024. mitmproxy. https://mitmproxy.org.

[39] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D Keromytis. 2016. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security (TOPS)* 19, 1 (2016), 1–31.

[40] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. ScriptProtect. https://github.com/scriptprotect/scriptprotect.

[41] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. 391–402.

[42] OWASP. 2021. OWASP Top Ten. https://owasp.org/www-project-top-ten4.

[43] OWASP. 2024. XSS Filter Evasion Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html.

[44] Jinkun Pan and Xiaoguang Mao. 2017. Detecting dom-sourced cross-site scripting in browser extensions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 24–34.

[45] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the ACM Conference on Computer and Communications Security*. 653–665.

[46] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. Auto-patching DOM-based XSS at scale. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 272–283.

[47] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The leaky web: Automated discovery of cross-site information leaks in browsers and the web. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2744–2760.

[48] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the Network and Distributed System Security Symposium*.

[49] Snyk. 2020. The State of Open Source Security 2020. https://snyk.io/series/open-source-security/report-2020.

[50] Pratik Soni, Enrico Budianto, and Prateek Saxena. 2015. The sicilian defense: Signature-based whitelisting of web javascript. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1542–1557.

[51] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the Web Conference*. 921–930.

[52] StatCounter. 2024. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share.

[53] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. 2021. Who's hosting the block party? studying third-party blockage of csp and sri. In *Proceedings of the Network and Distributed System Security Symposium*.

[54] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In) Security. In *Proceedings of the USENIX Security Symposium*. 971–987.

[55] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against DOM-based cross-site scripting. In *Proceedings of the USENIX Security Symposium*. 655–670.

[56] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *Proceedings of the USENIX Security Symposium*. 1015–1032.

[57] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1419–1430.

[58] Sucuri's Research. 2021. 2021 Website Threat Research Report. https://sucuri.net/wp-content/uploads/2022/04/sucuri-2021-hacked-report.pdf.

[59] Smitha Sundareswaran and Anna Cinzia Squicciarini. 2012. XSS-Dec: A hybrid solution to mitigate cross-site scripting attacks. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*. 223–238.
[60] Vimal Tarsariya (Vasundhara Infotech). 2023. Latest 13 Website Development Trends To Expect In 2023. https://vasundhara.io/blogs/web-development-trends.
[61] W3C. 2024. trusted-types. https://github.com/w3c/trusted-types.
[62] Pei Wang, Julian Bangert, and Christoph Kern. 2021. If it's not secure, it should not compile: Preventing DOM-based XSS in large-scale web development with API hardening. In *Proceedings of the International Conference on Software Engineering*. 1360–1372.
[63] Pei Wang, Bjarki Ágúst Guðmundsson, and Krzysztof Kotowicz. 2021. Adopting Trusted Types in ProductionWeb Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study. In *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*. 60–73.
[64] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1376–1387.
[65] Mike West. 2024. Web Mitigation Metrics - Trusted Types. https://mitigation.supply/#trusted-types.
[66] Seongil Wi, Trung Tin Nguyen, Jiwhan Kim, Ben Stock, and Sooel Son. 2023. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In *Proceedings of the Network and Distributed System Security Symposium*.
[67] Think with Google. 2016. Mobile Site Load Time Statistics. https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics.

A Benchmarks

To evaluate TrustyMon, we prepared 16 web applications that had at least one DOM-based XSS vulnerability. Table 5 describes statistics for each application. The first and second columns represent the name and version of a benchmark application. The following three columns describe the main programming language used, the total lines of code (LoC), and the number of stars on its GitHub repository or the number of user downloads, respectively.

We selected these benchmarks based on their popularity and diversity. These applications were implemented in various languages, including PHP, JS, Go, and Python, with LoCs ranging from 229 to 3.6M. Their GitHub repositories have received a minimum of 140 to 40.7K stars. The number of user downloads of WordPress plugins is over 10M, as reported on their respective download pages.

Table 5: Benchmark web applications (WP: WordPress).

Applications	Version	Language	LoC	GitHub Stars
Elementor (w/ WP)	3.4.7 (5.8.1)	PHP 7.4	316,863	10M+* (19.3k)
Yoast SEO (w/ WP)	2.1 (4.2.1)	PHP 7.4	158,171	10M+* (19.3k)
MyBB	1.8.22	PHP 7.4	131,467	1.1k
Microweber	1.3.1	PHP 8.1	1,870,564	3.1k
OpenEMR	7.0.0	PHP 7.4	2,070,073	3.1k
AbanteCart	1.3.1	PHP 7.4	263,831	140
Apache Answer	1.0.2	Go	31,800	12.6k
draw.io	20.5.0	JS	281,973	40.7k
URL Pages	commit 035b647	JS	229	1.4k
Railroad-diagram	commit ea9a123	Python	2,446	1.6k
Web-platform-tests	commit 938e843	JS	376,691	4.9k
Beep.js	commit ef22ad7	JS	1,428	1.4k
CesiumJS	1.111	JS	3,671,809	12.8k
LibreNMS	23.9.0	PHP 8.1	866,182	3.8k
Cacti	1.2.24	PHP 7.4	161,750	1.6k
Pimcore	10.5.18	PHP 7.4	2,706,124	3.3k

* User downloads

```
1  export class SafeXMLUtils {
2    constructor(xmlString) {
3      xmlPolicy = trustedTypes.createPolicy('xml-policy', {
4        createHTML: (s) => (s),
5      });
6      const xml = xmlPolicy.createHTML(xmlString);
7      const parser = new DOMParser();
8      this.xmlDoc = parser.parseFromString(xml, 'text/xml');
9    }
10 }
```

Listing 7: JS example using Trusted Types in Chrome.

```
1  function triggerDocsCanvasAnnotationMode() {
2    const extId = document.currentScript.src.split('/')[2];
3    const scriptContents = `
4      window['_docs_annotate_canvas_by_ext'] = "${extId}";
5    `;
6    const policy = trustedTypes.createPolicy('gdocsPolicy', {
7      createScript: (text) => text,
8    });
9    const sanitized = policy.createScript(scriptContents);
10   eval(sanitized);
11 }
```

Listing 8: JS example using Trusted Types in Chrome.

B Comparison with Trusted Types and TrustyMon

Trusted Types. Trusted Types can be enforced through user-defined, immutable policies that define how strings are converted into Trusted Type objects.

Listings 7 and 8 show JS snippets demonstrating how Trusted Types are deployed in Chromium [14]. Each example defines a Trusted Types policy using `createPolicy` in Lines 3–5 and 6–8, respectively. In both cases, the policy returns the input as is, without modifying or sanitizing the argument passed to the injection sink (Lines 4 and 7). Using this policy, the `createHTML` and `createScript` functions convert a dynamic or constant string into a Trusted Type object (Lines 6 and 9). Finally, the created Trusted Type object is passed to the injection sink (Lines 8 and 10).

As demonstrated in these examples, adopting Trusted Types requires web developers to define appropriate policies and make substantial modifications to their applications. This involves assigning Trusted Types to variables that affect webpage output and incorporating sanitization logic to ensure that only validated values are used.

TrustyMon. Unlike Trusted Types, TrustyMon does not require developers to define custom policies. To enable TrustyMon, website operators only need to inject a CSP header and include the TrustyMon library. The library includes predefined Trusted Types policies, which work across different applications without requiring any modifications.

Listing 9 presents the pseudocode for TrustyMon's Trusted Types policy. It defines a default policy that is automatically invoked whenever an injection sink receives a string instead of a Trusted Type object. In TrustyMon, we implemented Algorithm 1 in the default policy. For each type, TrustyMon parses the received string and generates its signature. In the extraction phase, the signature is stored in the database. During the monitoring phase, the signature is compared against registered signatures to determine whether execution should be allowed.

```
trustedTypes.createPolicy('default', {
  createHTML: function(htmlString) {
    parsedScript = parseDOM(htmlString);
    ast = parseAST(parsedScript);
    sig = generateSignature(ast);

    // Extraction phase
    saveSignature(sig, htmlString);

    // Monitoring phase
    if (matchSignature(sig)) return htmlString;
  },
  createScript: function(script) {
    ast = parseAST(script);
    sig = generateSignature(ast);

    // Extraction phase
    saveSignature(sig, script);

    // Monitoring phase
    if (matchSignature(sig)) return script;
  },
  createScriptURL: function(scriptURL) {
    sig = generateSignature(scriptURL);

    // Extraction phase
    saveSignature(sig, scriptURL);

    // Monitoring phase
    if (matchSignature(sig)) return scriptURL;
  }
});
```

Listing 9: Pseudocode of TrustyMon's Trusted Type policy.

In summary, defining a Trusted Types policy and implementing sanitization logic requires significant development effort. In contrast, TrustyMon leverages a list of registered benign JS signatures, reducing deployment costs by simply injecting a CSP header and including the TrustyMon library.

C DOM-based XSS Injection Sinks

- HTML
  - Document: `write()`, `writeln()`, `execCommand()`, `parseHTMLUnsafe()`
  - DOMParser: `parseFromString()`
  - Element: `innerHTML`, `outerHTML`, `setAttribute()`, `setAttributeNS()`, `insertAdjacentHTML()`, `setHTMLUnsafe()`
  - HTMLIFrameElement: `srcdoc`
  - Range: `createContextualFragment()`
  - ShadowRoot: `innerHTML`, `setHTMLUnsafe()`
- Script
  - Element: `setAttribute()`, `setAttributeNS()`
  - HTMLScriptElement: `text`, `innerText`
  - Node: `textContent`
  - Window: `setInterval()`, `setTimeout()`
  - `eval()`
  - `Function()`, `AsyncFunction()`, `GeneratorFunction()`, `AsyncGeneratorFunction()`
  - `javascript:`
- ScriptURL
  - Element: `setAttribute()`, `setAttributeNS()`
  - HTMLEmbedElement: `src`
  - HTMLObjectElement: `data`
  - HTMLScriptElement: `src`
  - Worker: `Worker()`
  - SharedWorker: `SharedWorker()`
  - WorkerGlobalScope: `importScripts()`

D Report Controller

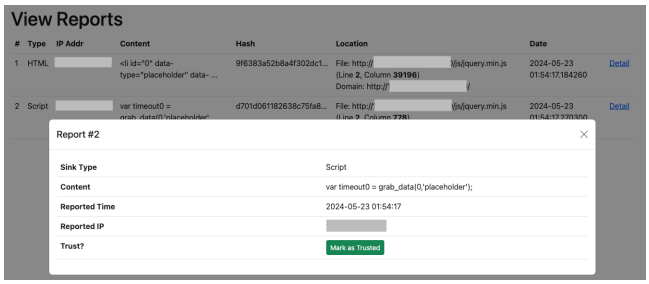


Figure 4: A web interface of Report Controller.

Report Controller allows website operators to register JS signatures from selected violation reports, enabling TrustyMon to handle false positives caused by missing benign signatures during the extraction phase. Figure 4 shows the Report Controller web interface. It shows a violation report along with a button to register the reported signature as benign.