**Noname manuscript No.**
(will be inserted by the editor)

# XSnare: Application-specific client-side cross-site scripting protection

**José Carlos Pazos · Jean-Sébastien Légaré · Ivan Beschastnikh**

**Abstract** We present XSnare, a client-side Cross-Site Scripting (XSS) solution implemented as a Firefox extension. The client-side design of XSnare can protect users before application developers release patches and before server operators apply them.

XSnare blocks XSS attacks by using previous knowledge of a web application's HTML template content and the rich DOM context. XSnare uses a database of exploit descriptions, which are written with the help of previously recorded CVEs. It singles out injection points for exploits in the HTML and dynamically sanitizes content to prevent malicious payloads from appearing in the DOM. XSnare displays a secured version of the site, even if is exploited.

We evaluated XSnare on 81 recent CVEs related to XSS attacks, and found that it defends against 93.8% of these exploits. We compared XSnare's funcitonality and protection with two well known content filtering extensions: NoScript and uBlockOrigin. To the best of our knowledge, XSnare is the first protection mechanism for XSS that is application-specific, and based on publicly available CVE information. We show that XSnare's specificity protects users against exploits which evade other, more generic, XSS defenses.

Our performance evaluation shows that our extension's overhead on web page loading time is less than 10% for 72.6% of the sites in the Moz Top 500 list. We also show that XSnare has as a slowdown of less than 10% on 60% of the vulnerable sites that we considered. XSnare has a false positive rate of 1/4876 (0.0205%) on the Alexa top 5000 sites.

**Keywords** cross-site scripting · security · web · program analysis

## 1 Introduction

Cross-Site Scripting (XSS) is still one of the most dominant web vulnerabilities. A 2021 report showed that 25% of websites contained at least one XSS

Department of Computer Science
University of British Columbia

vulnerability [1]. Countermeasures exist, but many of them lack widespread deployment, and so web users are still mostly unprotected.

Informally, the cause of XSS is a lack of input validation: user-chosen data "escapes" from the page's template and makes its way into the JavaScript engine, or modifies the Document Object Model (DOM). Consequently, many of the XSS defenses published so far propose to fix the problem at the source, by properly separating the template from the user data on the server, or by modifying browsers [2,3,4,5,6]. There are also similar solutions that can be implemented in the front-end code of an application [7]. In all cases, these technologies must be adopted by the application software developers, otherwise users are left unprotected.

One barrier to adoption of existing XSS defenses is that developers may not have the necessary expertise, or sufficient resources, to use the approach. Luckily, users wishing to gain reassurance over the safety of the sites they visit can install browser extensions to filter malicious scripts and content. Unfortunately, some of the most popular of these extensions, like NoScript [8], achieve most of their security by disabling functionality, such as JavaScript. Doing so unfortunately impairs usability[1]. A study by Snyder et al. [10] showed that browser security can be increased by disabling some rarely used JavaScript APIs, largely retaining usability. Our work builds on this idea, retaining website usability after an exploit is disabled.

When an XSS vulnerability is disclosed, some software vendors respond with patches. If the affected software is released in the form of packages, frameworks, or libraries, and used by several web applications, there is delay before users can benefit from the patch. Most importantly, the patched software must be re-deployed by site administrators.

Unfortunately, website administrators will not, and often cannot, apply software updates immediately: one study found that 50.3% of WordPress websites were running a version with known security vulnerabilities [11]. In another report, we learn that 30.95% of Alexa's top 1 Million sites run a vulnerable version of WordPress [12].

Users are at the mercy of developers and administrators if they want to access safe, up-to-date, applications. Our solution, XSnare, helps with this problem – based on information from past disclosures, XSnare patches known page vulnerabilities *directly in the browser*.

Each layer of the web application stack (Figure 1) presents different options to defend against XSS. Note that solutions at different layers are often complementary:

1. The application logic is the first line of defence. Code safety can be enhanced with third-party vulnerability scanning solutions, and a thorough code-review process. Taint, and static code analysis tools can detect unsanitized inputs.

[1] As early as 2012, JavaScript was used by almost 100% of the Alexa top 500 sites [9]

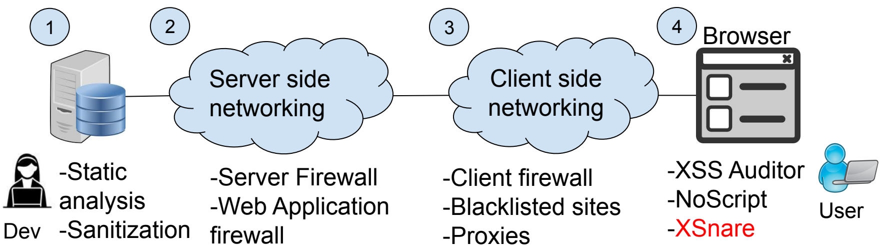


Fig. 1: Different web security solutions with XSnare on the client-side.

2. In the hosting environment, network firewalls, specifically Web Application Firewalls (WAFs) can defend against attacks such as DDoS, SQL injections and XSS.
3. In the client's environment (residential or commercial), users may install network firewalls, network content filters, and web proxies.
4. The last line of defence is the browser. Browser have built-in defences, such as Chrome's XSS Auditor [13]. Users can also install third-party extensions to block malicious requests and responses, such as NoScript [8], and XSnare.

We make two observations about existing solutions: (a) server-side solutions have to be applied independently on each server, and (b) solutions on the client are typically written as generic filters which attempt to catch everything, and consequently do not take full advantage of the specificity of the application or the vulnerability.

For example, a WAF can effectively protect the users, but users cannot realistically expect every site to be protected by a WAF. At the opposite end, in the client's environment, a user might configure a network proxy for all website traffic, with generic rules achieving maximum coverage, but this will often lead to an elevated rate of false positives (FPs).

Similarly, browser built-in defences are coarse-grained, and work on just a subset of exploits. Chrome's XSS Auditor, for example, only attempts to defend against reflected XSS. In May 2019, Google announced its intention to deprecate XSS Auditor (by September 2019, it had been removed from Chrome), for reasons including *"Bypasses abound"*, *"It prevents some legit sites from working"*, and *"Once detected, there's nothing good to do"* [14]. Stock et al. [15] propose enhancements to XSS Auditor and cover a wider range of exploits than the auditor, but are limited to DOM-based XSS. By contrast, our work covers all types of XSS.

Implementing adequate server-side protections requires time [16,17,18,19]. A 2018 study found that the average time to patch a known exploit in the form of a Common Vulnerability and Exposures (CVE), all severities combined, is 38 days, increasing to as much as 54 days for low severity CVEs, and the oldest unpatched CVE was 340 days old [20]. A more recent study found that 56% of

reported vulnerabilities were exploited within seven days of public disclosure, an 87% rise over 2020 [?].

Server-side defences also rarely protect against client-only forms of XSS, e.g., reflected XSS, or persistent client-side XSS, which use a browser's local storage or cookies as an attack vector. Steffens et al. [21] present a study of persistent client-side XSS across popular websites and find that as many as 21% of the most frequented web sites are vulnerable to these attacks. To provide users with the means to protect themselves in the absence of control over servers, we believe that a client-side solution is necessary.

A number of existing solutions in this area also suffer from high rates of false-positives and false-negatives. For example, NoScript [8] works via domain allow-listing: by default, JavaScript and other code will not execute. However, not all scripts outside of the allowlist should be assumed to be malicious. Browser-level filters like XSS Auditor use general policies and can incorrectly sanitize non-malicious content.

We posit that the DOM is the right place to mitigate XSS attacks as it provides a full picture of the web application. While most of the functionality we provide could be done by a network filter for the browser, we take advantage of additional browser context. Particularly, when an exploit occurs as a result of user interactions, like in response to a click, our approach benefits from knowing the initiating tab to filter the response. Previous client-side solutions have opted for detectors that were generic and site-agnostic [22,3,23]. Our work goes in the opposite direction, and tries to instead prevent precisely-defined exploits in specific applications.

If a patch for a server-side vulnerability can be "translated" into an equivalent set of operations to apply on the fully formed HTML document in the browser, then we seize the opportunity to defend *early* against exploits of that vulnerability. Our extension, which has access to the user's browsing context, can identify vulnerable pages based on a database of signatures for previous disclosures. This way, XSnare can protect users as soon as a patch is implemented and added to its database. The client-side patch will remain beneficial until *all* server operators running that software have had a chance to upgrade their deployments.

A similar philosophy is adopted by the client-side firewall-based network proxy Noxes [22]. However, due to their position in the stack, these policies do not defend against attacks invisible to the network, e.g., deleting local files.

Our system's signatures are designed to be application-specific, both in terms of exploit detection and sanitization. Application-specific signatures accurately dispose of exploits while retaining the web site's usability.

We described XSnare in an earlier conference paper [24]. In this paper we extend this prior work in the following ways:

- We detail XSnare's signature language (Subsection 2.9), and discuss key language design choices (Subsection 2.9).
- We expand the false positive study to a list of Alexa top 5000 sites [2].

[2] `https://www.cvedetails.com/cve/CVE-2018-5213/`

– We compare XSnare against two popular content filtering extensions: NoScript and uBlockOrigin [8, 25].
– We expand the performance evaluation of XSnare (Section 10), and include performance results on the set of Wordpress sites.
– We include a detailed list of XSnare limitations (Section 12).

## 2 XSnare Design

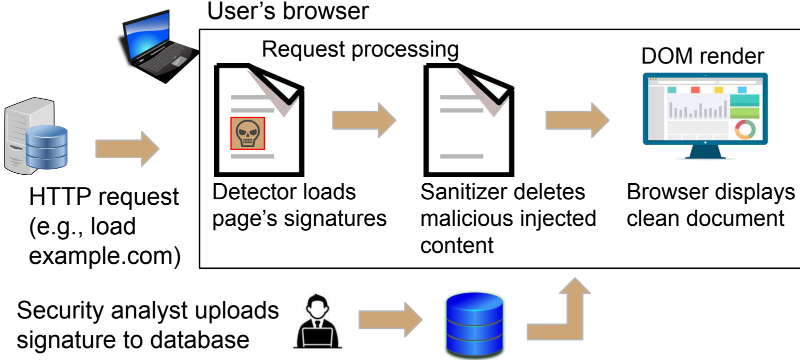


Fig. 2: XSnare's approach to protect against XSS.

We now present the design of XSnare and its components (Figure 2). We begin by reviewing our threat model.

### 2.1 Threat model

Our work makes no assumptions about the web server. In particular, the server may run out of date and vulnerable software that delivers pages to the user's browser with XSS exploits.

We trust the browser and the browser's extension mechanism to correctly execute XSnare. We also depend on the browser to disallow malicious tampering with the client-side signature database.

We trust the analyst who writes the signature definitions used by XSnare. For XSnare to be effective the signatures must be correct. However, a signature that fails to match a vulnerability, will only impact the page with longer load times.

### 2.2 Overview

We now review the high-level operation of XSnare with Figure 2. A user requests a page, *example.com*, on a browser with the XSnare extension installed. The response may or may not contain malicious XSS payloads. Before the browser renders the document, XSnare analyzes the potentially malicious document. The extension loads signatures from its local database into its detector. The detector analyzes the HTML string arriving from the network, and identifies the signatures which apply to the document. These signatures specify one or more "injection spots" in the document, which correspond, roughly speaking, to regions of the DOM where improperly sanitized content could be injected. The extension's sanitizer eliminates any malicious content and outputs a clean HTML document to the browser for rendering.

### 2.3 An example application of XSnare

To further explain our approach, we present a small example of how HTML context can be used to defend against XSS, taken from CVE 2018-10309 [26]. This is reproducible in an off-the-shelf WordPress installation running the Responsive Cookie Consent plugin, v1.7. This is a stored XSS vulnerability, and as such is not caught by some generic client-side XSS filters, including Chrome's XSS auditor.

Consider a website running PHP on the backend which stores user input from one user, and displays it later to another user, inside an **input** element.

The PHP code defines the static HTML template (in black), as well as the dynamic input (in red):

```
<input id=``rcc_settings[border-size]"
name=``rcc-settings[border-size]"
type=``text" value="<?php rcc_value('border-size'); ? >"/>
<label class=``description"
for=``rcc_settings[border-size]">
```

Normally, the **input** might have a value of "0":

```
<input id=``rcc_settings[border-size]"
 name=``rcc-settings[border-size]"
 type=``text" value="0">
<label class=``description"
 for=``rcc_settings[border-size]">
```

However, the php code is vulnerable to an injection attack:

```
border-size = ``"><script>alert('XSS')</script>
```

The browser will render this, executing the injected script:

```
<input id=``rcc_settings[border-size]"
name=``rcc-settings[border-size]"
```

```
type=‘‘text” value=“”><script>alert(’XSS’)</script>
<label class=‘‘description”
for=‘‘rcc_settings[border−size]”>
```

Note that the resulting HTML is well-formed, so a mere syntactic check will not detect the malicious injection. Let us assume a security analyst knows the original template, i.e., without injected content. If the analyst were given a filled-in document, they could (in most cases) separate the injected content from the server-side template, and get rid of the malicious script entirely, using proper sanitization.

The injected script is bounded by template elements with identifiable attributes. Assuming (for now) that there is only one such vulnerable injection point, we can search for the **input** element from the top of the document, and the **label** from the bottom to ensnare the injection points in the HTML.

This shares goals with the client/server hybrid approach of Nadji et al. [4]. They automatically tag injected DOM elements on the server-side using a taint-tracking, so that a modified browser can reliably separate template vs injected content. We do not require any server-side modifications, but rather opt for a client-side tagging solution based on exploit definitions.

The injected content, once identified, must be sanitized appropriately. The appropriate action will depend on the application setting, but assuming a patch has been written, it suffices to translate the intention in the server code's path to the client-side. This is straightforward once the fix is understood.

The developer incorrectly claimed the bug had been fixed in version 1.8 of the plugin. Other similar vulnerabilities had indeed been fixed, but not this one [27]. The built-in WordPress function sanitize_text_field needed to be applied.

XSnare does not automatically determine the actions to implement from a patch. We assign this task to a security analyst, who acts as the signature developer for an exploit. The system automates signature matching and sanitization.

### 2.4 XSnare Signatures

Our signature definitions make two assumptions: first, **an injection must have a start point and end point**, that is, an element can only be injected between a specific HTML node and its immediate sibling in the DOM tree; second, in a well-formed DOM, **the dynamic content will not be able to rearrange its location in the document before any HTML parsing in the browser** (e.g., removing and adding elements), allowing us to isolate it from the template. These assumptions are necessary for our content filter to work. Although we cannot prove these two assumptions, we also do not know of any counter-example to them.

Pages commonly contain more than one vulnerable injection point. We discuss the difficulty of supporting these pages in Subsection 2.7.

We believe CVEs are an ideal source of signature definitions. Previous client-side work does not benefit from our level of specificity; these tools often use less accurate heuristics to detect exploits. XSnare signatures are written specifically to match a known software vulnerability. Even though they cannot be generated automatically, any trusted developer can write and compose these signatures. This does not require participation from application developers.

In general, we do not require the existence of a publicly disclosed CVE to be able to write a signature for an exploit. CVEs have been useful to us as we did not discover the exploits. A knowledgeable analyst can write a signature without a public CVE. In fact, for security measures, many CVEs are not publicly available until the application developer has patched its software. Our system can help reduce the time between zero day attacks and patch deployment: an analyst can write a signature for a vulnerability as soon as they know the issue.

Long term, we imagine that volunteers (or entrepreneurs) would cultivate and maintain the signature database. New signatures could be contributed by a community of amateur or professional security analysts, in a manner not so different from how antispam or antivirus software is managed. The popular ad blocking extension AdBlock, for example, relies on filter rules taken from open-source filter lists [28].

The challenge of automatically deriving signatures from detailed CVEs would serve for an interesting problem to solve, albeit outside the scope of this paper.

### 2.5 Firewall Signature Language

Our signature language needs enough power of expression for the signature writer to be precise, both for determining the correct web application and to identify the affected areas in the HTML. For injection point isolation, a language based on regular expressions suffices to express precise sections of the HTML. The following is the signature that defends against the motivating example of Subsection 2.3:

Listing 1: An XSnare signature

```
url: 'wp-admin/options-general.php?page=rcc-settings',
software: 'WordPress',
softwareDetails: 'responsive-cookie-consent',
version: '1.5',
type: 'string',
typeDet: 'single-unique',
sanitizer: 'regex',
sanitizerConfig: '/^[0-9](\.[0-9]+)?$/',
endPoints:
['<input id=''rcc_settings[border-size]" name=''
    rcc_settings[border-size]" type=''text"
```

```
  value=‘‘’,
‘<label class=”description”
for=‘‘rcc_settings[border-size]”>’]
```

In summary, a signature will have the necessary information to determine whether a loaded page has a vulnerability, and specify appropriate actions for eliminating any malicious payloads.

Analysts configure their signatures with one function chosen from the static set of sanitization functions offered by XSnare (Subsection 3.2). These functions inoculate potentially malicious injections based on the DOM context surrounding the injection. The goal of signatures is to provide such sanitization, ideally without “breaking” the user experience of the page. The default function preset is DOMPurify’s [7] default configuration, which takes care of common sanitization needs [29]. However, DOMPurify’s defaults can be unnecessarily restrictive, or not restrictive enough, in which case the other sanitization methods are preferable.

We considered allowing arbitrary sanitization code in signatures. While it would open complex sanitization possibilities, we have decided against it, principally for security reasons. The minimal set of functions we settled on also sufficed to express all of the signatures defined for this paper. See Appendix A for more details.

## 2.6 Browser Extension

Our system’s main component is a browser extension which rewrites potentially infected HTML into a clean document. The extension detects exploits in the HTML by using signature definitions and maintains a local database of signatures. We leave the design of an update mechanism to future work, but in its current form, the database is bundled with each new installation of the extension.

The extension translates signature definitions into patches that rewrite incoming HTML on a per-URL basis, according to the top-down, bottom-up scan described in Subsection 2.3.

The extension’s detector acts as an in-network filter. We initially considered other designs but quickly found out that applying the patch at the network level was necessary for sanitization correctness: even before any JavaScript code runs, the browser’s built-in DOM parser can rearrange elements into an unexpected order, making our extension sanitize the wrong spot. Consider the following example, where an element inside a <tr> tag is rearranged after parsing the string:

```
<table class=‘‘wp-list-table”>
  <thead>
    <tr>
          <th></th>
          <img src=“1” onerror=“alert(1)”>
```

```
                <th>
                <form method=‘‘GET” action=‘‘”> ...
```

In this HTML, the signature developer might identify the exploit as occurring inside the given table. However, if we wait until the string has been parsed into a DOM tree to sanitize, the elements are rearranged due to <tr> not allowing an <img> as its child:

```
<img src="1" onerror="alert(1)">
<table class=‘‘wp-list-table”>
    <thead>
    <tr>
                <th></th>
                <th>
        <form method=‘‘GET” action=‘‘”> ...
```

Note that the injected <img> tag is now outside of the table, simply by virtue of the DOM parsing. The extension will not find an injection in the expected place, creating a false negative (FN). Similarly, elements rearranged inside an injection point can create false positives. This example would generate a class of circumvention techniques for our detector, so we must analyze the DOM before the browser has a chance to modify it.

### 2.7 Handling multiple injections in one page

In Listing 1, the endPoints were listed as two strings in the incoming network response. However, there are cases where arbitrarily many injection points can be generated by the application code, such as a for loop generating table rows. For these, it is hard to correctly isolate each endPoint pair, as an attacker could easily inject fake endPoints in between the original ones.

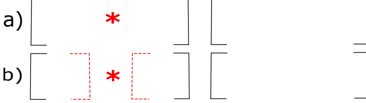


Fig. 3: Example attacker injection when multiple injection points exist in the page. a) a basic injection pattern. b) an attempt to fool the detector.

In Figure 3a, the brackets indicate a template. The content in between (i.e., the star) is an injection point , where dynamic content is injected into the template. In the case of a vulnerability, the injected content can expand to any arbitrary string. The signature separates the injection from the rest by matching for the start and end points (the endPoints), represented by the brackets. This HTML originally has two pairs of endPoint patterns.

In Figure 3b, the attacker knows these are being used as injection end points and decides to inject a fake ending point and a fake starting point (the dotted brackets), with some additional malicious content in between. If just searching for multiple pairs of end points, the detector cannot tell the difference between the solid and dotted patterns, and will not get rid of the content injected in the star. Therefore, we have to use the first starting point and the last ending point before a starting one (when searching from the bottom-up) and sanitize everything in between.

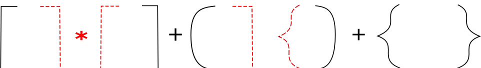

Fig. 4: Example attacker injection when multiple distinct injection points exist in the page.

Figure 4 illustrates a case when there are several injection points in one page, but each of them is distinct. Now, the filter is only looking for one pair of brackets, so the attacker can't fool the extension into leaving part of the injection unsanitized. However, they could, for example, inject an extra ending bracket after the opening parenthesis (or an extra starting brace). The extension will be tricked into sanitizing non-malicious content, the black pluses (+). Since we know the order in which the endPoints should appear, when the filter sees a closing endPoint before the next expected starting endPoint, or similarly, a starting endPoint before the next expected closing endPoint, this attack can be identified. In the diagram, the order of the solid elements characterizes the possible malformations in the end points.

In both scenarios, we have to sanitize the outermost end points. This might get rid of a substantial amount of valid HTML, so we defer to the signature developer's judgment of what behavior the detector should follow. We expand upon this further in Subsection 6.1.

Note that these complex cases do not mean that our approach is not applicable, as the extension provides a choice for blocking the page entirely if the signature writer believes a given case is too complex for our signature language.

### 2.8 Dynamic injections

The top-level documents of web pages fetch additional dynamic content via fetch or AJAX APIs. Content fetched in this way is also vulnerable to XSS, and must be filtered. An example vulnerability is CVE-2018-7747 (in WordPress Caldera Forms, which allows malicious content retrieved from the plugin's database to be injected in response to a click.

XSnare allows XMLHTTPRequests (XHR) to be filtered with XHR-type signatures. To reduce the number of signatures that need to be considered when a browser issues a request, we require that signatures for XHR be nested inside a signature for a top-level document. If a page's main content matches an existing top-level signature description, XSnare will then enable all nested XHR listeners.

Listing 2 shows an example of such a signature. The idea is extensible to scripts and other objects loaded separately from the main document (e.g., images, stylesheets, etc.).

Listing 2: An example dynamic request signature. This patches CVE-2018-7747.

```
...
listenerData: [{
  listenerType: 'xhr', listenerMethod: 'POST',
  sanitizer: 'escape', type: 'string',
  requestUrl: 'wp-admin/admin-ajax.php',
  typeDet: 'single-unique',
  endPoints: ['<p><strong>', '[AltBody]']
}]
```

### 2.9 Signature Language Description

We now further detail XSnare's signature language. A signature is specified as a JavaScript object. Each of the fields in this object captures an aspect of the signature's behaviour, including website identification, type of request (see Subsection 2.8), injection endpoints, and sanitization methods. Some fields are optional, as they are either used for added signature loading precision, or only need to be defined based on the value of another field. Required fields are used to identify the type of request and the injection point details. Additionally, some fields, like the sanitization method have default options. In this section we focus on the design behind the language. Appendix A goes into more detail about the implementation and application of each field. Our extension includes several pre-packaged signatures, so a signature developer can refer to a large set of example signatures.

While our signature language has a strong basis on the inner workings of HTML and XSS, we not only wanted to make a language strong enough to cover a wide variety of vulnerabilities, but also one that would feel intuitive for an analyst and would not overwhelm them with unnecessary details. In this sense, we've had to be rather restrictive in the constructs we allow.

For most of the signature fields, we've taken careful consideration into its design:

- **url**: While our evaluation has focused mostly on WordPress sites, and in general in CMSs, this does not mean our extension only applies to those. In

fact, it would be easier for us to detect vulnerabilities in some other pages, for example, by URL. This way, we can skip the probing and even the versioning. Even in the case of WordPress sites running vulnerable plugins, we found that many of them manifested themselves only in a subpage of the site. A signature developer can specify this page so as to not affect other pages which would also trigger the signature.

- **software**: Probes are an important component of our software identification functionality. While we could have offloaded software identification to the signature writer, in practice we found that much of the same software is prone to having vulnerabilities. Naturally, this is bound to occur in an open development environment, like in many CMS systems. Thus, we believe that incorporating the probes directly in our extension alleviates the workload for the developer.
- **softwareDetails**: This field is for providing more details about the 'software', and follows a similar design choice: software specific probes are able to identify software details, for example, plugins in WordPress, as these are most often identified in the same way in the HTML.
- **version**: We discuss the details and complications that arise from versioning in 3.1.
- **sanitizer**: As discussed in Subsection 3.2, we provide three different methods: DOMPurify, regex and escape. While DOMPurify is the most powerful out of the three, we found it to be unnecesarily restrictive in many cases in practice. We consider it the default option, but a developer might deem the other methods better suited for a particular scenario.
- **sanitizerConfig**: Even though we consider DOMPurify as the default sanitization method, it also provides much more fine-grained control over its sanitization [3]. The developer can take advantage of this to tune the signature. For escaping, the developer can specify a pattern for escaping, which will use JavaScript's *replace* method on the injection point. For regex matching, this field specifies which pattern to match. If the pattern matches the injection point text, it will leave it as is. Otherwise, it will remove the content entirely. These different behaviours correspond to distinct requirements: DOMPurify is a more powerful, but potentially less precise sanitizer, escaping is for when an injection point should be allowed certain string content that can be sanitized by taking out special characters (e.g., a string without "<, >, [, ], etc."), and regex is for when a restrictive pattern should be met (e.g., only alphanumeric characters).
- **typeDet**: We went through several iterations to determine the most concise way to specify the cases where one page might have several injections. We eventually decided that it could be reduced to four fundamental cases, dictated by two binary options. That is, the same injection can occur once or several times throughout the HTML, its 'occurrence' (e.g., in the case of a table with several elements), and there can be several distinct injections

[3] `https://github.com/cure53/DOMPurify/tree/main/demos#what-is-this`

in the same page (e.g., one page suffering from several different vulnerabilities).
- **endPoints**: As our injection points are just a StartPoint and EndPoint couple, we decided that these should be specified as list of tuples (2-element arrays). Developers need to take into account that a small mistake could result in a missed vulnerability.
- **endPointsPositions**: Even similar strings might not be subject to the same vulnerabilities. For example, an <input class="some-input"> element might only be subject to an exploit if it occurs inside of another element. An analyst familiarized with HTML/CSS might find it easier to specify these locations with Selectors. However, tracking these positions is best done in a parsed HTML site, not a text string. Unfortunately, we cannot do this because it might result in element rearrangement. Therefore, we decided to specify these positions as arrays of indices. This is tedious, but fortunately, we believe this will occur in a small subset of vulnerabilities. In our studied sites, we used this field in 7 signatures (out of 64 we wrote).
- **listenerDetails**: This field is for dealing with dynamic requests, and is further described in Appendix A

## 3 Implementation

We implemented our system as an extension in Firefox 69.0. Our signatures are stored in a local JavaScript file in the extension package. We decided on an extension implementation for several reasons. (1) *Privileged execution environment.* The extension's logic lies in a separate environment from the web application code. This guarantees that malicious code in the application cannot affect the extension. (2) *Web application context.* Our solution requires knowledge of the application's context. The extension naturally retains this context. (3) *Interposition abilities.* As it lies within the browser, the extension can run both at the network level, e.g., rewrite an incoming response; and at the web application level, e.g., interpose on the application's JavaScript execution.

### 3.1 Filtering process

Algorithm 1 describes our network filtering process: once a request's response comes in through the network, we process it and sanitize it if necessary.

**Loading signatures.** Our detector loads signatures and finds injection points in the document. However, not all signatures need to be loaded for a specific website, since not all sites run the same frameworks. When loading signatures, we proceed in a manner similar to a decision tree. The detector first probes the page (line 3) to identify the underlying framework (the **software** in our signature language). We currently provide a number of static probes. However, as more applications are required to be included, we believe it would

**Algorithm 1:** Network filter algorithm

```
1  //global DBSignatures
2  procedure verifyResponse (responseString, url)
3      loadedProbes = runProbes(responseString, url)
4      signaturesToCheck ← []
5      for probe in loadedProbes do
6          signaturesToCheck.append(DBSignatures[probe])
7      end
8      filteredSignatures ← []
9      for signature in signaturesToCheck do
10         if responseString and url match signature then
11             filteredSignatures.push(signature)
12     end
13     versionInfo ← loadVersions(url, loadedProbes)
14     endPoints ← []
15     for signature in filteredSignatures do
16         if (signature,signature.version) ∈ versionInfo then
17             endPoints.push(signature.endPointPairs)
18     end
19     indices ← []
20     for endPointPair in endPoints do
21         indices.push(findIndices(responseString, endPointPair))
22     end
23     if discrepancies exist in indices then
24         Block page load and return
25     for endPointPair in endPoints do
26         sanitize(responseString,indices)
27     end
28 end
```

be better to cover this task in the signature definitions. The widely popular network mapping tool Nmap [30] uses probes in a similar manner, kept in a modifiable file. As mentioned in Section 4, we currently only have signatures for Content Management Systems (CMSs) applications. Our probes use specific identifiers related to the application, as well as the particular site that is affected by the exploit. WordPress pages, for example, have several elements in the page that identify it as a WordPress page. While this might seem easier for CMS style pages, and we acknowledge that application fingerprinting is a hard problem in general, we believe other web apps will also have similar identifying information, like headers, element ID's, script/CSS sources, classes, etc. Previous work has shown that DOM element boundaries can be effectively identified given some previous knowledge of the DOM structure [31].

After running these probes, the detector loads corresponding frameworks' signatures and filters out checks whether the information of each loaded signature matches the page (lines 5-12).

**Version identification.** We then apply version identification (lines 13-16). Our objective for versioning is to prevent signatures from triggering false positives on websites running patched software. We found this to be one of the harder aspects of signature loading. In many CMSs, for example, file names

are not updated with the latest version, and versioning information is often unavailable on the client-side.

We have observed that even if we load a signature when the application has already been patched on the server, it can preserve the page's functionality. Motivated by this observation, our mechanism follows a series of increasingly accurate but less precise version identifiers. If versioning is unavailable in the HTML, the patch is applied as we cannot be sure the page is running patched software.

**Injection point search and sanitization.** Once we have the correct signatures, we find the indices for the endpoints using our top-down, bottom-up scan, and need to check for potential malformations in the injection points (lines 19-24), as described in Subsection 2.7. If this occurs, the page load is blocked and a message is returned to the user, or if the signature developer specifies so, sanitization proceeds on the new endpoints. Finally, if all **end-Point** pairs are in the expected order, we sanitize each injection point (lines 25-27).

### 3.2 Sanitization methods

We provide different types of sanitization: "DOMPurify", "escape", and "regex". DOMPurify works well as an out-of-the-box solution (although it has more precise functionality as well through its API). Escaping can be useful when only a few characters need to be filtered. Regex Pattern matching can be particularly effective when the expected value has a simple representation (e.g., a field for only numbers).

## 4 Evaluation methodology

To verify the applicability of our detector and signature language, we tested the system by looking at several recent CVEs related to XSS. We have three objectives: to verify that our signature language provides the necessary functionality to express an exploit and its patch, to test our detector against existing exploits, and to show that composing signatures takes a reasonable amount of time.

We study recent CVEs related to WordPress plugins. We focus on WordPress for two reasons:

1. WordPress powers 34.7% of all websites according to a recent survey [32] [33]. The same study states that 30.3% of the Alexa top 1000 sites use WordPress. Thus, we can be confident that our study results will hold true for the average user.
2. WordPress plugins are popular among developers (there are currently more than 55,000 plugins [34]). Due to its user popularity, WordPress is also heavily analyzed by security experts. A search for WordPress CVEs on the Mitre CVE database [35] gives 2310 results. Plugins, specifically, are

an important part of this issue, 52% of the vulnerabilities reported by WPScan are caused by WordPress plugins [36]. In particular, a search for WordPress XSS CVEs returns 1124 as of the time of the study.

We used a CVE database, CVE Details [37] to find the 100 most recent WordPress XSS CVEs, as of October 2018. For each CVE, we set up a Docker container with a clean installation of WordPress 5.2 and installed the vulnerable plugin's version. For CVEs that depended on a particular WordPress version, we installed the appropriate version. Of the CVEs we looked at, only one occurred in WordPress core. We believe it would be harder to precisely sanitize injection points in WordPress core, as many of the plugins have particular settings pages where the exploits occur, and the HTML is more identifiable. WordPress core, on the other hand, can be heavily altered by the use of themes and the user's own changes. However, as evidenced by our investigation, the vast majority of exploits occur in plugins.

Next, we reproduced the exploit in the CVE and we analyzed the vulnerable page and wrote a signature to patch the exploit. The signature validation process was done manually by one of the researchers. It consisted of validating that the PoC exploit was defended against.

This researcher's background is 5 years of JavaScript/HTML/CSS experience, with an MSc in Software Security. The researcher did not have experience in penetration testing or automated exploit detection tools (a CVE writer would most likely have at least some experience in these).

## 5 Signature effectiveness on CVEs

| Plugin | Installations |
|---|---|
| WooCommerce | 5+ million |
| Duplicator | 1+ million |
| Loginizer | 900,000+ |
| WP Statistics | 500,000+ |
| Caldera Forms | 200,000+ |

Table 1: Top 5 most popular WordPress plugins included in our study.

| Type-0 (DOM Based) | Type I (Reflected) | Type II (Stored) |
|---|---|---|
| 22 | 18 | 36 |

Table 2: Types of XSS exploits studied.

Of the initial 100 CVEs, we were able to analyze 76 across 44 affected pages. We dropped 24 CVEs due to reproducibility issues: some of the de-

scriptions did not include a PoC, making it difficult for us to reproduce; or, the plugin code was no longer available. In some cases, it had been removed from the WordPress repository due to "security issues", which emphasizes the importance of being able to defend against these attacks.

Table 2 shows the number of CVEs for each type of XSS vulnerability we found. While stored attacks were most frequent, we were able to validate signatures against all three types of XSS. Since all XSS manifests itself in the DOM at some point, this does not necessarily improve the representativeness in terms of attacks found in the wild. We discuss this further in Section 12.

The plugins we studied averaged 489,927 installations: Table 1 shows the number of installations for the 5 most popular plugins studied. For the vulnerabilities, 27 (35.5%) could be exploited by an unauthenticated user; 56 (73.7%) targeted a high-privilege user as the victim, 7 (9.2%) had a low-privilege user as the victim; the rest affected all users.

Many of the studied CVEs included attacks for which there are known and widely deployed defenses. For example, many were cases of Reflected XSS, where the URL revealed the existence of an attack, e.g.,: `http://<target> &page-uri=<script>alert("XSS")</script>` While Chrome's built-in XSS auditor blocked this request, Firefox did not, and so we still wrote signatures for such attacks. In practice, we found several cases where Firefox did not block a reflected XSS. We list these cases in Section 9.

We wrote 59 WordPress signatures in total, which got rid of the PoC exploit when sanitized with one of our three methods. Note that while a PoC is often the most simple form of an attack, our sanitization methods can get rid of complex injections as well. We were able to include several CVEs in some PoCs because they occurred in the same page and affected the same plugin. Overall, these signatures represent 71 (93.4%) signed CVEs. The 5 we were not able to sign were due to lack of identifiers in the HTML, which would result in potentially large chunks of the document being replaced[4].

After manual testing, the majority of the 71 signatures maintained the same layout and core functionality of the webpage. However, 12 signatures caused some elements to be rearranged. One caused a table showing user information to render as blank. Most of the responsibility of maintaining functionality is left to the signature developer. We found that being precise is key to retaining functionality. Furthermore, even if the layout of the page is affected, we believe that applying the signature is preferable to allowing an exploit. And, unlike the complete blocking approach commonly used by malware detection software, our approach opens the option to let the user access the page.

While our goal is to retain as much information of the page as possible after sanitization, we believe that even if a part of the page becomes unusable, this does not impact the user's experience as much, since many of the exploits

[4] In these cases, the signature developer can weigh the trade-offs and decide whether the added cost is worth it.

occur in small sections of the HTML. A usability study is out of scope for this paper and we leave it to future work.

### 5.1 Applicability beyond WordPress

To test the applicability of our approach to other frameworks, we analyzed 5 additional CVEs, 2 related to Joomla!, 2 for LimeSurvey, and 1 for Bolt CMS. We chose Joomla! because it is another popular CMS. Unfortunately, we only found 2 CVEs that we were able to reproduce, as the software for its extensions is often not available. For fairness, we looked for the most recent CVEs we could reproduce listed in the Exploit Database [38], since these have recorded proof of concepts (PoCs). We carried out the same procedure as with the WordPress CVEs, and were able to patch all of the 5 exploits. This brought our CVE coverage rate up to 93.8%.

## 6 Writing Signatures

We expect a signature developer to have a solid understanding of the principles behind XSS, as well as web applications, HTML, CSS and JavaScript, so they can identify precise injection points. In this section, we aim to show that minor effort is required from an analyst when writing a signature.

### 6.1 Case Study: CVE-2018-10309

Going back to our example in Subsection 2.3, we describe the process for writing a signature using one of our studied CVEs.

**Identifying the exploit.** An entry in Exploit Database [39] describes a persistent XSS vulnerability in the WordPress plugin Responsive Cookie Consent for versions 1.7/1.6/1.5. This entry describes the Cookie Bar Border Bottom Size parameter as vulnerable. We run a local WordPress installation with this plugin.

**Establishing the separation between dynamic and static content.** We insert the string "><script>alert('XSS')</script> in the Cookie Bar Border Bottom Size (rcc_settings[border-size] in the HTML) input field as a PoC. This results in an alert box popping up in the page.

In general, the analyst can find the vulnerable HTML from the server-side code without reproducing the exploit. Since we did not discover the exploit, we had to do this extra step.

In the example, the **input** element is the injection starting point, and the **label** tag is the end point. Identification of correct endpoints is extremely important, and in particular, when a page has multiple injection points, the signature developer must ensure the elements do not overlap with other innocuous ones. In some cases, the developer might think it best to stop the page

from loading due to the complexity of the injection points. We believe that if sanitization is impractical, compromising usability for security is preferable.

**Collecting other required page information and writing the signature.** The next step is to gather the remaining information to determine whether the signature applies to the page loaded. The full signature for this example was previously shown in Listing 1. The page's HTML includes a link to a stylesheet with href "http://localhost:8080/wp-content/plugins/responsive-cookie-consent...", "wp-content/plugins/plugin-name" is the standard way of identifying that a WordPress page is running a certain plugin, in this case, "responsive-cookie-consent". Since the exploit only occurs in this specific spot in the HTML, the **typeDet** is listed as "single-unique". Since the vulnerable parameter is a border-size, the **sanitizer** applied is "regex", further restricting the pattern to only numbers in **config**. We list the **endPoints** as taken from the HTML.

**Testing the signature**. Finally, we run the extension. We expect to not have an alert box pop up, and we manually look at the HTML to verify correct sanitization. If the exploit is not properly sanitized, the developer is able to use the debugging tools provided by the browser to check the incoming network response information seen by the extension's background page and make sure it matches the signature values.

### 6.2 Signature writing times

Figure 5 plots a histogram of the times it took one of the authors to compose each of the signatures. Each time measurement includes the time it took to check the HTML injection points, write the signature and to debug it. We do not include the time taken to discover and carry out an exploit, as we assume a vulnerability has been discovered already. The median time is 3.89 minutes, and the standard deviation is 4.18 minutes. 72% of signatures were written in under 5 minutes. We believe this to be a reasonable amount of time considering the security granted by our extension.

The signature which took the longest time to write (25 minutes) corresponds to an exploit with 12 HTML injection points. Additionally, testing this signature proved difficult, as some of the injections were a result of a script inserting elements in the DOM after the page had loaded. This caused the initial HTML to look innocuous, but with exploits still occurring after sanitization. As this script was part of the initial request, we eventually got to the root of the problem. We believe a more experienced exploit analyst might be able to detect this kind of behaviour more easily.

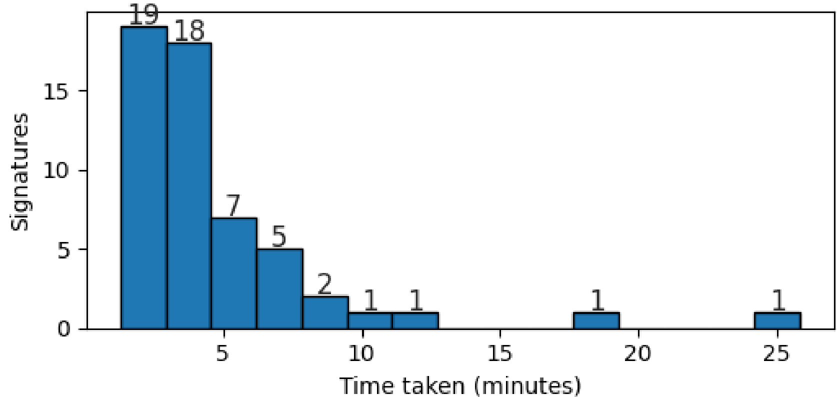


Fig. 5: Histogram of time taken to write signatures.

## 7 XSnare's false positives rate on the web

To evaluate the false positive rate of XSnare, we carried out a study with the Alexa top 5,000 sites[5]. For each site we recorded the number of loaded signatures. We had consistent issues loading 124 of these sites, due to 404 errors, timeouts, broken SSL and broken redirects. Overall, we found a false positive rate of 1/4876 or 0.0205%.

The one false positive that we found was a particular site that loaded one signature and which was running WordPress with the Simple Download Monitor plugin. When cross-referencing with our list of signatures, we found that this loading was triggered by a protection measure against CVE-2018-5213 [6]. In particular, this signature does not specify a path for the page where the exploit might be triggered. This means that the signature is loaded on any page that is running this plugin.

Furthermore, since this exploit is not limited to the admin site of WordPress, it is harder to identify the correct version. In this particular signature, we did not specify a version, as the HTML of the user site did not contain accurate version information. In general, this could be detrimental to a user's experience, as this means the signatures will be loaded on sites that are running updated versions of this plugin. Indeed, we verified that this plugin has been patched for the XSS exploit.

Not, however, that even though the signature was loaded on all pages of this web site, we checked the site functionality manually and found no failures on any of the pages. In fact, even though the signature was, no sanitization was done, as none of these pages had the HTML elements targeted by the

[5] `https://gist.github.com/chilts/7229605`

[6] `https://www.cvedetails.com/cve/CVE-2018-5213/`

signature. Thus, even loading the signature incorrectly had no downside for the user in this case. We also compared the functionality of this site when JavaScript was disabled. We noticed several differences in the page, not only visually but also in terms of functionality. We expand upon this in the next section.

From our results, we can infer with confidence that the rate of falsely loaded signatures during an average user's web browsing is low. This rate could potentially go up as we expand XSnare to cover more frameworks. Even in this extended set, many of these sites are not running any of our covered frameworks and are very popular and therefore more prone to fixing their vulnerabilities. However, since the most popular sites account for a great deal of the average user's browsing time, we can conclude with greater confidence that false positives will be as low or even lower in practice.

## 8 Comparison to other solutions

| | **Deployment** | **Type-0** | **Type I** | **Type II** | **Functionality** |
|---|---|---|---|---|---|
| **XSnare** | Extension | ✓ | ✓ | ✓ | Maintained |
| NoScript | Extension | ✓ | ✓ | ✓ | Disabled |
| uBlockOrigin | Extension | ✓ | ✓ | ✓ | Configurable |
| Browser Filters | Built-in | ✗ | ✓ | ✗ | Configurable |

Table 3: Comparison with existing popular solutions. Deployment is either browser extension or built into the browser. Different types of XSS can be covered by different configurations of the solution. Functionality refers to maintaining the page's functionality.

XSnare has a unique design: it requires no browser modifications and no server-side requirements. Thus, many XSS defences are complementary to XSnare and would not make for a fair comparison.

Table 3 shows a comparison between different existing solutions with a similar deployment scenario as XSnare. While at first glance it may seem NoScript and uBlockOrigin [8, 25], both popular content blocking extensions, are able to defend against all types of XSS, this comes at a cost. Both need to disable JavaScript functionality, or, in the case of uBlockOrigin, can additionally be kept with heavy user customization. XSnare, on the other hand, is designed to maintain as much of the page's functionality as possible, and requires no configuration from the user's point of view. By their nature, browser filters are only able to defend against reflected XSS and, depending on the configuration, can attempt to sanitize the page or block it entirely. Many of these have already been deprecated or removed, as is the case for Chrome's XSS Auditor and Microsoft Edge's XSS Filter.

In the rest of this section, we compare XSnare against NoScript and uBlock-Origin. Currently, the Firefox Addons site specifies the number of users of these extensions as 357,725 and 5,228,853 respectively. Thus, uBlockOrigin is by far the most popular one. We decided to choose these two mainly on popularity due to the current lack of widespread XSS client-side solutions. Not only do they make for a fair comparison in terms of use-cases (users trying to defend themselves against attacks), but also in terms of deployment scenarios (client-side as a web extension).

The website for NoScript claims to have the *"the most powerful anti-XSS and anti-Clickjacking protection ever available in a browser"*. It can certainly get rid of several XSS issues. NoScript allows 4 settings for security: Default, Temp. TRUSTED, and Untrusted. Each of these settings comes with predetermined permissions for different web features: script, object, media, frame, font, webgl, fetch, ping, noscript, unrestricted CSS, other. The Untrusted setting blocks all of these, while the TRUSTED one allows all of them. The Default setting blocks most functionality, including scripts. All of these settings can be modified, and the addon also provides a CUSTOM setting, allowing the user to set all permissions by themselves. NoScript also provides an option to "Sanitize cross-site suspicious requests". However, there do not appear to be any other ways to customize XSS filtering functionality.

The site for uBlockOrigin does not mention XSS explicitly. However, it does mention blocking scripts, especially ads. This extension is more customizable than NoScript, as it lets a user block particular HTML elements from a page. It also allows for more fine-grained blocking control. For example, blocking 3rd party scripts, 1st party scripts, inline scripts, frames, and to perform dynamic content filtering.

**Example 1: CVE-2018-10309**. This CVE refers to a WordPress site running the Responsive Cookie Consent Plugin, vulnerable to Stored XSS. Note that in our setup we only have the vulnerable plugin installed. Real-life websites would most likely have several plugins (at least for WordPress) and might therefore be subject to additional failures due to disabling of JavaScript.

**Experimental settings**:

1. NoScript default configuration.
2. NoScript custom configuration: all elements allowed except for scripts.
3. uBlockOrigin blocking inline scripts.
4. uBlockOrigin specialized filters.

**Results**:

1. The exploit is correctly disabled. However, there are some downsides to this. Figure 6a shows the side menu on the regular page (this is also what is displayed with XSnare running). Figure 6b shows the side menu with the default NoScript configuration. Not only are the icons missing, but some of these menu items show a submenu when hovering over them. With NoScript, this functionality was broken.
2. The icons showed up correctly, but the submenu functionality was still broken. This particular page has 30 script elements. While not all of them

might be vital to the page either visually or functionally, there are several extra scripts being disabled.

3. Disables the submenu functionality, but properly gets rid of the exploit. We applied uBlockOrigin's block for inline scripts, as this is the only filter that gets rid of the exploit. The submenus were still non-functional. However, as this is more fine-grained than NoScript, several of the 30 scripts where still working. In general, this would be a better compromise for most XSS, but still not as precise as we would like it to be. uBlockOrigin also offers "element zapping" functionality. A user can specify a particular HTML element to disable from the page. Unfortunately, this doesn't seem to work on elements that are not native to the page, like those that are injected. Therefore, we resorted to using uBlockOrigin's filtering language.
4. This extension provides a language to identify HTML elements correctly in a page. One potential weakness of this extension's approach is that this filtering is done after the response has already been received by the browser parsing engine. Thus, the filtering might occur after an HTML rearrangement, as discussed in Subsection 2.6. This would cause the filter to potentially miss a dangerous element in the document. For the purposes of this particular page, correct use of the filter results in a similar behaviour as with XSnare running. However, it is much more tedious to write, as script elements are not the only ones that can trigger malicious behaviour, and is the reason why using DOMPurify makes our sanitizer so effective.

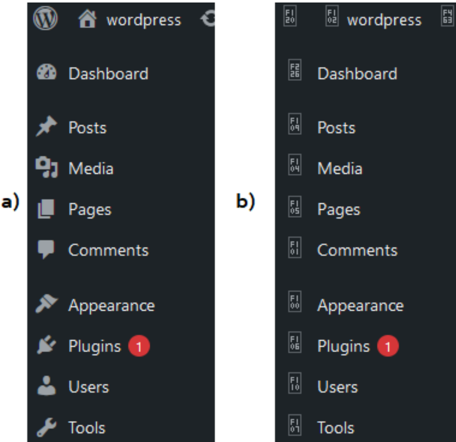


Fig. 6: WordPress site running a) XSnare b) NoScript default configuration.

**Example 2: CVE-2018-17947**

This CVE refers to a WordPress site running the Snazzy Maps Plugin, vulnerable to Reflected XSS. NoScript states that it is particularly good at detecting Reflected XSS.

**Experimental settings**:

1. NoScript default configuration.
2. NoScript TRUSTED configuration.
3. uBlockOrigin specialized filters.

**Results**:

1. This leads to some functional issues with the page.
2. NoScript will show a popup indicating that it has detected a potential XSS attack. However, the popup only has a few options: Blocking the request, always block, allow the request, and always allow. While the user experience should be enhanced, as they are aware of a potential issue; in practice, this isn't much better than disabling JavaScript, as a user has to decide whether they want to risk loading the page with a potential exploit or not loading it at all.
3. Works more precisely than NoScript when using their specialized filters. As Reflected XSS is not any less vulnerable to different elements being injected, this extension suffers from the same drawbacks as before in this example.

**Example 3: www.on24.com**

XSnare loaded a signature on a live website. This was caused by a vulnerable plugin being detected on the site. As discussed in Section 7, this is likely a false positive. It is nevertheless instructive to compare the behaviour of the site when running these three extensions. None of the pages we browsed were affected by XSnare.

**Experimental settings**:

1. NoScript default configuration.
2. uBlockOrigin disabling third-party scripts (no default configuration on this page.)
3. uBlockOrigin disabling inline scripts.

**Results**:

1. We noticed several issues with the site's functionality. Figure 7 shows an example of a section of the website with this configuration. As can be seen in Figure 8, NoScript causes some visual issues. Furthermore, the top banner did not open any submenus when hovering over it. In fact, clicking on the banner did not redirect the page. Other issues included some notice banners not showing up, a contact form not showing at all, amongst others.
2. This results in similar behaviour to NoScript.
3. Breaks the site even further. Of course, as this particular site is likely not vulnerable, there is not much benefit in disabling JavaScript in terms of XSS. However, if the site was running a vulnerable version of the plugin, uBlockOrigin would not detect it by default, unless a user specified a rule for this particular page. This is one of the major downsides of this extension (and NoScript) when compared to XSnare. Our extension's application detection is domain agnostic. Of course, it's much easier to correctly load

signatures on a unique site, as we don't even have to worry about versioning. Furthermore, even when loading a false positive, the page still worked as expected. We consider this to be a net gain for the user, as they are protected in the worse case, and will often not be affected even if signatures are loaded.

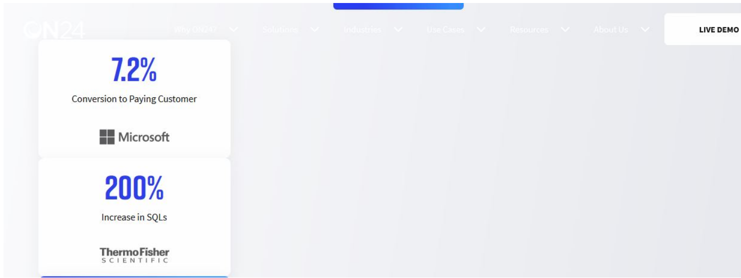


Fig. 7: on24.com running NoScript default configuration.

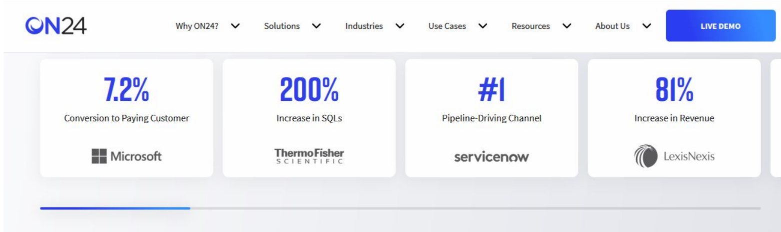


Fig. 8: on24.com running XSnare.

## 9 Exploits unblocked by Firefox

In this section, we list the 10 reflected XSS cases which were blocked by the Chrome XSS Auditor but were not blocked by Firefox during our evaluation: CVE-2018-1000556, CVE-2018-5316, CVE-2018-5293, CVE-2018-5292, CVE-2018-5288, CVE-2018-5286, CVE-2018-18460, CVE-2018-6002, CVE-2018-6001 and CVE-2018-9034. Recall that the Chrome XSS Auditor has now been removed from the Chrome browser, so this serves as anecdotical evidence, as no CVEs would currently be blocked by a default Chrome installation. However, this does show that existing popular solutions have false negatives.

## 10 Load time performance on top websites

XSnare's performance goal is to provide its security guarantees without impacting the user's browsing experience. We now report on XSnare's impact on top website load times, representing the expected behaviour of a user's average web browsing experience.

While our extension's functionality only applies at the network level, there is potential slowdown at the DOM processing level due to the optimization techniques the browser applies throughout several levels of the web page load pipeline. Figure 9 shows the different timestamps provided by the Navigation Timing API [40], as well as a high-level description of the browser processing model. Since our filter listens on the onBeforeRequest event, none of the previous steps before the Request are affected. In this section, we refer to the difference in time between responseEnd and requestStart as the "network filter time".

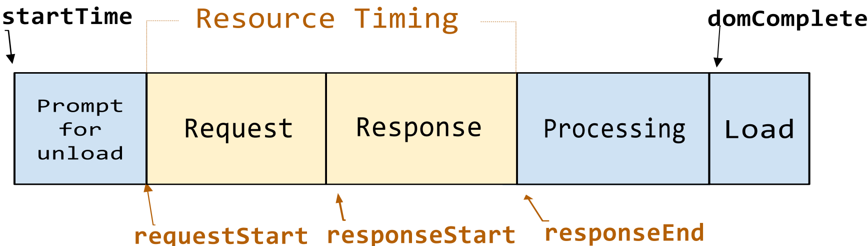


Fig. 9: The Navigation Timing API's timestamps[7]

In our setup, we used a headless version of Firefox 69.0, and Selenium WebDriver for NodeJS, with GeckoDriver. All experiments were run on one machine with an Intel Xeon CPU E5-2407 2.40GHz processor, 32 GB DRAM, and our university's 1GiB connection.

In our tests we used the top 500 websites as reported by Moz.com [41]. For each website, we loaded it 25 times (with a 25 second timeout) and recorded the following values: requestStart, responseEnd, domComplete, and decodedBodySize. From the initial set of 500 sites, we only report values for 441: the other 59 had consistent issues with timeouts, insecure certificates, and network errors. We believe these to have been caused by the Selenium web driver, as our extension runs after a response has been delivered to the browser. We manually loaded each page on a personal computer with our extension running successfully and were not able to reproduce the issues. We ran four test suites:

[7] This image was taken from the w3 spec: `https://www.w3.org/TR/navigation-timing-2/`

- **No extension cold cache**: Firefox is loaded without the extension installed and the web driver is re-instantiated for every page load.
- **Extension cold cache**: As before, but Firefox is loaded with the extension installed.
- **No extension warm cache**: Firefox is loaded without the extension installed and the same web driver is used for the page's 25 loads.
- **Extension warm cache**: As before, but Firefox is launched with the extension installed.

For each set of tests, we reduced the recorded values to two comparisons: network filter time, and page ready (domComplete - responseStart). The first analyzes the time spent by the network filter, while the second determines the time spent until the whole document has loaded. We calculate the medians for each website for each of these measures as well as the decodedByteSize.

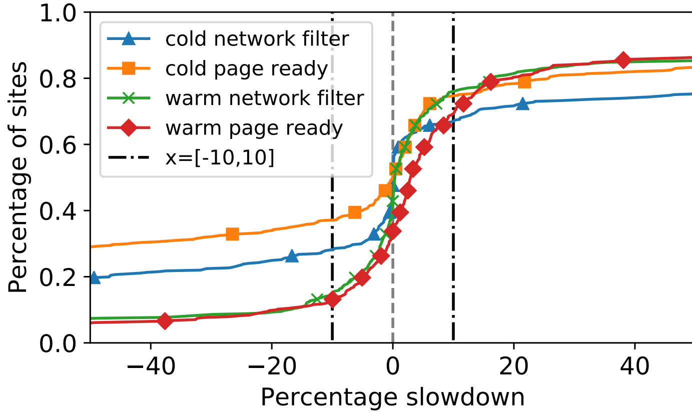


Fig. 10: Cumulative distribution of relative percentage slowdown with extension installed for top sites.

We compare the load times with/without the extension by calculating the relative slowdown with the extension installed according to the following formula:

$$100 * \frac{\tilde{x}_{with} - \tilde{x}_{without}}{\tilde{x}_{without}}$$

where $\tilde{x}$ is the median with/without the extension running.

Figure 10 plots the results. We can see a slowdown of less than 10% for 72.6% of sites, and less than 50% for 82% of sites when the extension is running. Note that these values are recorded as percentages, and while some are as high as 50%, the absolute values are in 77% of cases less than a second. This overhead should not alter the user's experience significantly.

The slowdown increases by at most 5% when we take caching into account. This is likely because the network filter causes the browser to use less caching, especially for the DOM component, as it might have to process it from scratch every time. While it may seem counter-intuitive that some pages have shorter loading times with the extension, there are several variables at play that can affect these measurements (local network, server-side load, internal scheduling, etc). For example, for one of the external pages, load times between different trials varied between 664ms and 1363ms without the extension running. With the extension running it varied between 697ms and 1582ms. While the ranges show that it would be slower with the extension running, depending on the time taken for a given trial, the average could end up being longer without the extension.

We manually checked the websites for which values were higher than |40%| and verified that our extension did not change the page's contents, a possible cause of faster load times. We also checked the timings for the page as reported by the browser and noted a high variance even within small time windows. The time spent by our verification function was less than 10ms for 87.6% of sites (Figure 11). This corroborates our findings that the slowdown is mostly negligible.

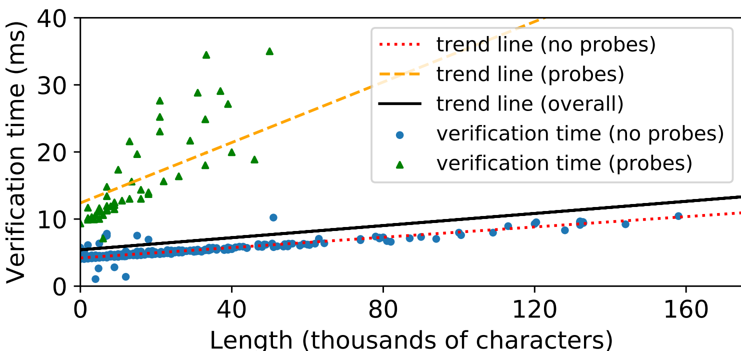


Fig. 11: Scatter plot of network filter time as a function of character length for top sites.

Figure 11 shows the time spent by the call to our string verification function in the network filter as a function of the length of the string to be verified, differentiating between websites for which some probes tested positive and ones which no probes did. We applied least squares regression to calculate the shown trend lines. The Spearman's rank [8] correlation values for no probe, probe, and overall are 0.91, 0.91, and 0.72 respectively, demonstrating positive correlation. Since both our probes and signatures use regex matching, we expect both

[8] The Spearman's correlation coefficient measures the strength and direction of association between two ranked variables: `https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php`

trend lines to be linear, as seen in the graph. We expect the slope of the line to be higher when a probe passes, as it performs additional string verification. Around 37.4% of all web sites use frameworks covered by our probes [32], thus, we expect the impact of our network filter to be closer to the non-probe values, as corroborated by our overall trend line.

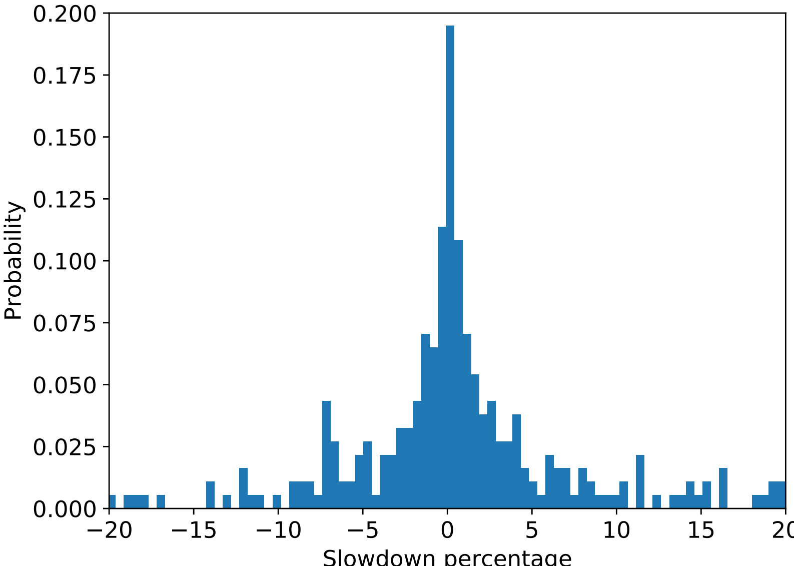


Fig. 12: Density histogram of network filter slowdown for top sites

Figure 12 shows a closer look at the distribution for the cold network filter slowdown on the top sites (same values as in Figure 10). We filtered out any values above |100|%, as these were mostly due to timeouts and other network delays or errors. For this component, 87.6% of the slowdown values are less than 10%.

When factoring in the rest of the network component, additional noise is introduced in the measurements. Figure 13 shows the network filter time as a function of the page's decoded byte size. Applying least squares regression shows an upwards trend. However, the Spearman's rank correlation for this set of data is −0.054; we believe this to be due to the number of factors affecting this measurement. The data demonstrates that the size of the page being processed has a smaller impact than expected (see Figure 11) on the network filter time when the whole page load process is accounted for.

**Scalability with signatures.** We tested our system with a large number of signatures. We added 15,500 signatures to our database and recorded the

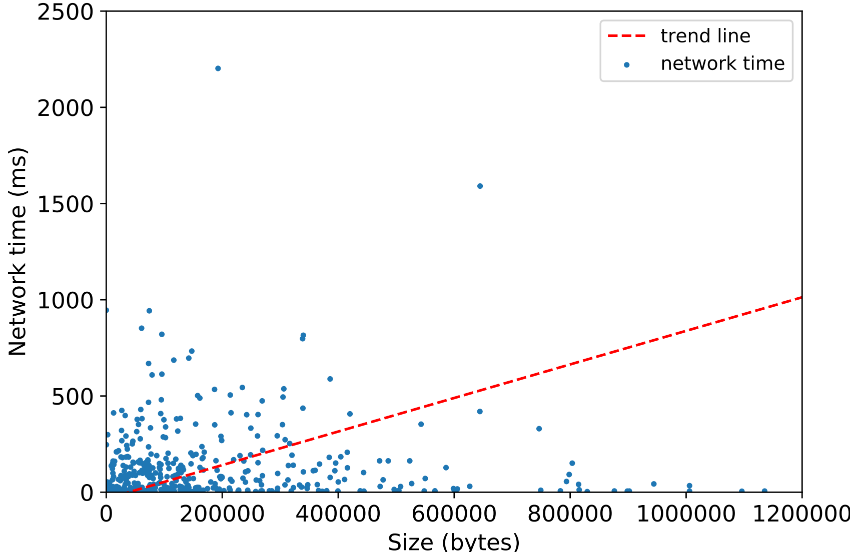


Fig. 13: Scatter plot of network filter time as a function of decoded byte size for top sites

time spent by the network filter to process these sites[9]. These were crafted so that the extension would check each one against the loaded sites, without triggering the injection search and sanitization. Thus, we effectively forced our extension to test each site against 15,500 signatures. The mean time spent by the filtering process was 1,930ms, with less than 2,000ms for 88% of the sites. In practice, we expect a smaller filter time, as many frameworks would have many signatures. For example, there are currently 200+ CVEs listed for WordPress core and its plugins.

## 11 Wordpress websites load times

We ran similar experiments as in Section 10, but with the WordPress sites described in Section 4. Thus, all of these have either one or multiple injection points in their HTML, and the network filter will spend an additional amount of time sanitizing these as defined by the signatures. Note that the dataset is smaller here, and some of the trends might be harder to infer.

As before, Figure 14 shows the results for slowdown with the extension running. Recall that the only difference between a page which passes the WordPress probe and one that matches a signature is that the latter has to replace

[9] There are 15,303 CVEs related to XSS in CVE Details [42].

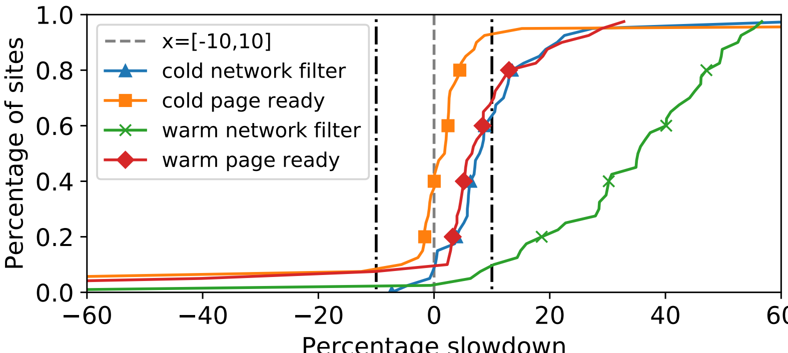


Fig. 14: Cumulative distribution of relative percentage slowdown with extension installed for WordPress sites

a portion of the original string by its sanitized version. In this case we see a slowdown of less than 10% for 60% of sites, and less than 40% for 96.25% of them. The warm network filter curve suffers from a particularly high slowdown. We believe this to be the case because the locally hosted pages decrease the network component time, causing any overhead to be seen as relatively high. However, as 48% of the original values were below 60ms, conclude a small impact on user experience as well.

Figure 15 shows the probability density of the cold network filter slowdown. In this case, we see that the distribution is skewed more towards a higher slowdown. As it is harder to discern the trend for this dataset than its top site counterpart, we have also plotted the normal distribution of the data between 3 standard deviations. 65% of values are below 10%.

Finally, as in Figure 11, we report the string verification time as a function of its length, for the WordPress sites, shown in Figure 16. The Spearman's rank correlation for this dataset is 0.630.

## 12 Limitations and Future Work

**Generalizability and scope of study.** As discussed in Section 4, while many websites share similar structures to the ones we covered, our study only considered 4 other sites not running WordPress, and our signatures only cover CMS content. Not all websites might be identified as easily. Furthermore, we only studied 81 CVEs. While this gave us a diverse set of scenarios to test and develop our signature language and extension implementation with, which account for about 7.5% (84/1124) of WordPress XSS CVEs at the time of the study, this could still be improved in a future study.

As previously mentioned, we do not claim to cover all types of attacks. Unfortunately, as we do not have a rigorous study with different attack vectors

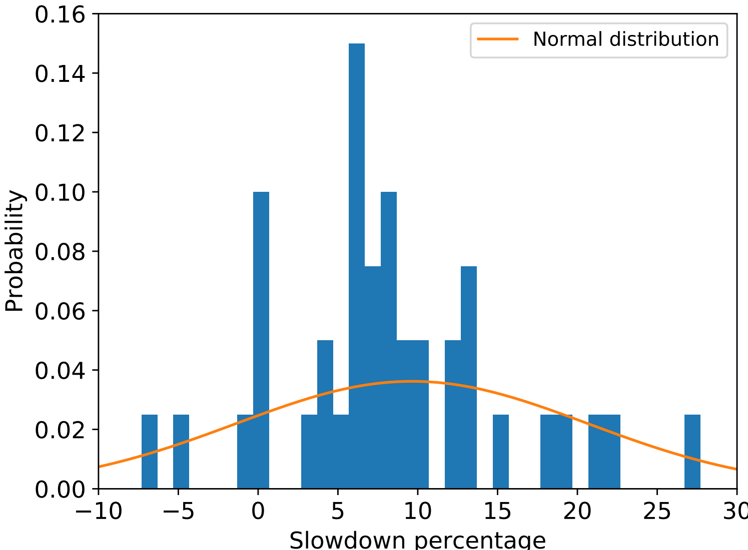


Fig. 15: Density histogram of network filter slowdown for WordPress sites

for each vulnerability, there is a possibility for a new or existing vulnerability to not be covered by our signatures.

**False positives and false negatives.** It is difficult to rid entirely of false positives (FPs). Even if a signature is loaded in a correct page, if the sanitization targets JavaScript code, for example, a FP will likely be triggered, by sanitizing potentially harmless code. An even more detailed study on FPs could investigate the rate of harmless content that has been deleted by our sanitization choices in vulnerable pages. In particular, it could also analyse different scenarios of false positives to ascertain the representativeness of our unique false positive case.

Faulty sanitization could be reduced by implementing our sanitization methods as a lexer instead as a declarative version (our current design). In this case, the signature developer would provide the allowed behaviour in a given injection point, and the detector would check the injection content against the specified behaviour, providing a proof of whether the content could have been generated by following the rules. This would make sanitization more accurate.

Furthermore, since we rely on handwritten signatures, vulnerable sites for which no signature has been written will be subject to FNs. In the future, we intend to study the rate FNs in our approach and to compare it to previous work.

**Protection against CSRF.** We believe that we can adapt our work to defend against client-side Cross-Site Request Forgery (CSRF) exploits, as well. Using a similar signature language as the one for XSS, a signature developer

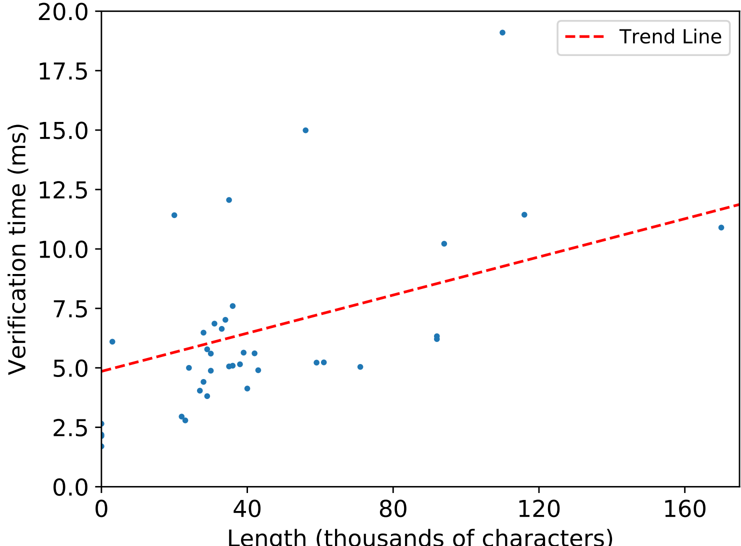


Fig. 16: Scatter plot of network filter time as a function of string length for WordPress sites

could specify pages with potential vulnerabilities to only allow network requests that cannot exploit such vulnerabilities. This would mostly be based around blocking requests for certain URLs. In initial designs of XSnare, we had looked at intercepting JavaScript execution, similarly to a Proxy. This turned out to have bypasses, but in CSRF, this would work, as these exploits occur after the page has loaded. For this application, we could give developers the power to create customized proxies for JavaScript execution to block malicious requests, as extensions can interpose on any JavaScript execution. While the language used would certainly require a similar design process, we believe the basic framework would keep many similarities.

**Deployment limitations.** Our filter's design depends on Firefox's implementation of the WebRequest API. Firefox's filterResponseData method allows the extension to modify an incoming HTTP request[10]. This feature has been requested in other browsers like Chrome, but it has not been implemented. This design limits our deployability to Firefox users. Recently, other network intercepting functionality has been requested and implemented in several other browsers, mainly in the form of Service Workers [11]. According to the API, these "essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended,

[10] `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData`

[11] `https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API`

among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server.” However, it seems like these workers are usually installed after the page is loaded, and working on subsequent events. Our filter needs to run before the page even loads, otherwise XSS could be triggered. An extension might be able to install service workers when we need them instead of using the Firefox WebRequest API. In the future, we intend to investigate this feature further in order to be able to expand our deployment to more browsers.

**Design considerations**. Currently, each browser user has to install our extension. However, the same functionality could be offloaded to a single processing unit similar to a proxy, which can handle the filtering for all machines in a network. This deployment model might be more appropriate in certain environments, such as in an enterprise setting.

**Signature updates**. Our signatures come pre-installed with the extension, and are currently stored in a JavaScript file. Our signature creation process is open-source in nature, and we believe users should have the right to import signatures at will. Currently, an user would need to modify the signature file to add new signatures. In the future, we intend to implement features that allow users to import signature files. Popular blocking extensions like uBlockOrigin and NoScript, as well as the previously mentioned application Nmap work in similar ways [25,8,30]. Furthermore, we believe that by hosting our source code in an open-source environment, signature developers can propose new signatures to upload to the database and these can be audited and merged. The extension could load the file from the repository, so it wouldn't require manual updates.

**Extension customization.** Our current implementation could benefit from several customization options. As specified above, importing signature files would be useful. Furthermore, an user might be willing to add to an allowlist certain pages or even customize the warning that is displayed when the HTML is modified. Moreover, adding custom probes would also be useful, as distinct users could visit many different sites. In the future, we intend to add these and other customization options to improve the user experience.

**Signature Validation.** While most of the sanitization heavy duty is left to DOMPurify and the signature writer, we have not evaluated our signatures against such a wide variety of strings, only what was required to achieve the CVEs exploits. This does not mean we used the exact same string over and over. However, in the future, we intend to automate this process with different known XSS attack strings.

**Signature Writing.** Our signature evaluation consisted of a single researcher whose experience is most likely not representative of the set of people we expect to write these signatures. In particular, a writer only needs to be able to translate a CVE into a signature according to our language, and need not be a security researcher. However, we believe that there still might be a difference in difficulty and time between our researcher and an average signature writer. In the future, we would like to extend our evaluation to include user

study of analysts, and track the difficulty to learn the language and transform a CVE into a signature, the time taken to do so, and the knowledge gaps that might exist during this translation.

## 13 Related Work

We classify existing work into several categories: client-side, server-side, browser built-in, and hybrid approaches.

**Server-side techniques.** In addition to existing parameter sanitization techniques, taint-tracking has been proposed as a means to consolidate sanitization of vulnerable parameters, and identify vulnerabilities automatically. [2,16,17,18,19,43]. These techniques are complementary to ours, and provide an additional line of defence against XSS.

There has also been work on other server-side analysis approaches to find bugs security vulnerabilities in web applications. [44,45,46]. However, these do not target XSS specifically.

**Client-side techniques.** DOMPurify [7] presents a robust XSS client-side filter. The authors argue that the DOM is the ideal place for sanitization to occur. While we agree with this view, this work relies on application developers to adopt the filter and modify their code to use it. We have partly automated this step by including it as our default sanitization function.

Jim et al. [3] present a method to defend against injection attacks through Browser-Enforced Embedded Policies. This approach is similar to ours, as the policies specify prohibited script execution points. Similarly, Hallaraker and Vigna [23] use a policy language to detect malicious code on the client-side. Like XSnare, they make use of signatures to protect against known types of exploits. However, unlike our approach, their signatures are not application-specific, and there is no model for signature maintenance.

Snyder et al. [10] report a study in which they disable several JavaScript APIs and test the number of websites that are do not work without the full functionality of the APIs. This approach increases security due to vulnerabilities present in several JavaScript APIs, however, we believe disabling API functionality should only be used as a last resort.

Additionally, client-side taint tracking, through the use of static and dynamic analysis, has also been applied as a means to detect XSS, either at the browser level or at the extension level [47,48].

**Browser built-in defences.** Browsers are equipped with several built-in defences. We previously described XSS Auditor in Section 1. Another important one is the Content Security Policy (CSP) [49]. It has been widely adopted and in many cases provides developers with a reliable way to protect against XSS and CSRF attacks. However, CSP requires the developer to identify which scripts might be malicious. Previous work has also highlighted the need for further built-in defences [50].

**Client and server hybrids.** XSS-Dec [6] uses a proxy which keeps track of an encrypted version of the server's source files, and applies this information

to derive exploits in a page visited by the user. This approach is similar to ours, since we assume previous knowledge of the clean HTML document. Furthermore, they use anomaly-based and signature-based detection to prevent attacks. Our system offloads all this functionality to the client-side, without the need for any server-side information.

## 14 Conclusion

Users cannot depend on administrators to patch vulnerable server-side software or for developers to adopt best practices to mitigate XSS vulnerabilities. Instead, users should protect themselves with a client-side solution. In this paper we described the design, implementation, and evaluation of XSnare, one such client-side approach. XSnare prevents XSS exploits by using a database of exploit signatures and by using a novel mechanism to detect XSS exploits in a browser extension.

We evaluated XSnare through a study of 81 CVEs in which we showed that it defends against 93.8% of the exploits. We compared XSnare's functionality and protection with two well known content filtering extensions: NoScript and uBlockOrigin, and showed that it has superior capabilities in terms of precise detection of vulnerabilities and website usability. We also showed that XSnare's false positive rate is extremely low (0.0205%).

## Data Availability

The datasets for this paper, including the XSnare artifact, are available from the corresponding author on reasonable request.

## A Signature Language Description

We now provide further description of our signature language. We implement a signature as a key-value JavaScript object. Each of the fields details an aspect of the signature's behaviour. Table 4 describes this.

If the value of *type* in Table 4 is 'listener', the signature will have an additional field called *listenerData*, which acts as a nested signature with fields that identify a particular dynamic request. A developer can optionally specify the request's injection points using this nested signature. Table 5 describes this sub-signature format.

## References

1. I. Muscat. (2017, jun) Acunetix vulnerability testing report 2017. https://www.acunetix.com/blog/articles/acunetix-vulnerability-testing-report-2017/.
2. G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 171–180. [Online]. Available: https://doi.org/10.1145/1368088.1368112

| **Signature fields and descriptions** |
|---|
| **url**. Optional (string)<br>If the exploit occurs in a specific URL or subdomain, this is defined as a string, e.g., `/wp-admin/options-general.php?page=relevanssi%2Frelevanssi.php`, otherwise null. |
| **software**. Optional (string)<br>The software framework the page is running, e.g., 'WordPress'. Some pages might not have a framework, i.e., handcrafted pages. Note that software probes, are required for specific software. Probes are built-in application detectors in the extension and are discussed in Subsection 3.1. If a probe is not available, in our current implementation, the software will not be correctly identified, as there is currently no built-in way to add probes to the extension. |
| **softwareDetails**. Optional (string)<br>If the *software* field has been given a value, this field can provide further information about when to load a signature for this software for added precision; since a vulnerability could occur in a website running the base software, or a plugin for this software (thus, this field remains optional even if *software* has been given a value). For WordPress and similar CMS's, these are plugin names as depicted in the HTML of a page running the plugin. For example, most WordPress plugins can be identified by a script element with the substring *wp-content/plugins/PluginName*. In this case, the value of *softwareDetails* would be *PluginName*. Note that this value should be as precise as possible, because it will determine when the signature is loaded. Furthermore, this depends on the value of *software*. In particular, probes for both the software and the software details need to be implemented. |
| **version**. Optional (string)<br>The version number of the software/plugin/page, used for versioning as described in Subsection 3.1. Note that versioning in general is hard and can only be done with precise information or elevated permissions. |
| **type**. Required (string)<br>Parameter for describing the signature type, 'string' describes a basic signature, 'listener' describes a signature with a listener for additional network requests. See Table 5 for more details on listener specifications. |
| **listenerData**. Only required if *type* is 'listener' (string)<br>A field for specifying the identification information for a dynamic request. This is written as a nested signature. See Table 5 for more details on listener specifications. |
| **sanitizer**. Optional (string)<br>The sanitizer to be used, "DOMPurify", "escape", or "regex". The default is DOMPurify. |
| **sanitizerConfig**. Optional for DOMPurify and escape. Required for regex. (string for regex, tuple of strings for escaping, or JavaScript DOMPurify configuration)<br>Additional configuration parameters to go along with the chosen sanitizer. For example, DOMPurify allows configuration through its API (i.e., DOMPurify.sanitize(dirty, sanitizerConfig). For 'escape', an additional escaping pattern can be provided, in the form of array, where the elements are the arguments of JavaScript's 'string.replace' method. For 'regex', this should be the pattern to match with the injection point content. |
| **typeDet**. Required (string)<br>A string describing the recurrence of injection points throughout the document, e.g., 'single-unique', as described in Subsection 2.7. This is specified in the format 'occurrence-uniqueness': 'occurrence' has values single/multiple, which describes the existence of one or multiple independent injection points; the 'uniqueness' has values unique/several, specifying whether an injection point occurs once or several times throughout the document. |
| **endPoints**. Required (array of tuples)<br>An array of startpoint and endpoint tuples, specified as strings for regex matching. |
| **endPointsPositions**. Optional (array of tuples)<br>An array of integer tuples. These are optional but useful when the one of the endPoints HTML are used throughout the whole page and appear a fixed number of times. For example: if an injection ending point happens on an element <h3 class='my-header'>, this element might have 10 appearances throughout the page. However, only the 4th is an injection ending point. The signature would specify the second element of the tuple to be 7, as it would be the 7th such item in a regex match array (using 1-based indexing), counting from the bottom up. For ending points, we have to count from the bottom up because the attacker can inject arbitrarily many of these elements before it, and vice versa for starting points. |

Table 4: XSnare Language Description. Each row describes a field in the signature object. Fields can be required or optional, sometimes based on values of other fields. Additionally, some optional fields have default non-null values.

| *listenerData* **sub-signature fields and descriptions** |
| --- |
| **listenerType**. Required (string)<br>The type of network listener as defined by the WebRequest API [12] (e.g., 'script', 'XHR', etc.) |
| **listenerMethod**. Required (string)<br>The request's HTTP method, for example "GET" or "POST". |
| **requestUrl**. Required (string)<br>The URL of the dynamic request target. Note that this is different from the page's URL. For example, an XHR could be fetched from the '/my-ajax.php' subdomain |

Table 5: Additional fields for *listenerData* when a *type* of 'listener' has been specified in a signature. Injection endpoint information (*typeDet*, *endPoints*, *endPointsPositions*) and sanitization information (*sanitizer*, *sanitizerConfig*) can be specified either in this subsignature or in the parent signature. A listener could even have another listener nested into it, by setting the *type* and *listenerData* fields. We have omitted the repetition of those fields in this table.


3. T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 601–610. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242654
4. Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense." in *NDSS*, vol. 20, 2009.
5. P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating XSS attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, ser. IWSESS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 33–39. [Online]. Available: http://dx.doi.org/10.1109/IWSESS.2009.5068456
6. S. Sundareswaran and A. C. Squicciarini, "XSS-Dec: A hybrid solution to mitigate cross-site scripting attacks," in *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*, ser. DBSec'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 223–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31540-4_17
7. M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against XSS and markup injection," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 116–134.
8. Noscript homepage. https://noscript.net/.
9. B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 971–987. [Online]. Available: http://dl.acm.org/citation.cfm?id=3241189.3241265
10. P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 179–194. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133966
11. (2016) Hacked website report 2016/q3. https://sucuri.net/reports/Sucuri-Hacked-Website-Report-2016Q3.pdf.
12. (2019) Statistics show why wordpress is a popular hacker target. https://www.wpwhitesecurity.com/statistics-70-percent-wordpress-installations-vulnerable/.

13. (2019) XSS auditor. https://www.chromium.org/developers/design-documents/xss-auditor.
14. (2019) Intent to deprecate and remove: XSSAuditor. https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/TuYw-EZhO9g/blGViehIAwAJ.
15. B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 655–670. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671267
16. W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267345
17. A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, 2005, pp. 295–308.
18. T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 124–145. [Online]. Available: http://dx.doi.org/10.1007/11663812_7
19. P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 23–43. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70542-0_2
20. (2018) Security report for in-production web applications. https://www.rapid7.com/resources/security-report-for-in-production-web-applications/.
21. M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.
22. E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, "Client-side cross-site scripting protection," *Comput. Secur.*, vol. 28, no. 7, pp. 592–604, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2009.04.008
23. O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 85–94. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2005.35
24. J. C. Pazos, J.-S. Légaré, and I. Beschastnikh, "Xsnare: Application-specific client-side cross-site scripting protection," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 154–165.
25. ublock origin. https://github.com/gorhill/uBlock#ublock-origin.
26. (2018) Wordpress plugin responsive cookie consent 1.7 / 1.6 / 1.5 - (authenticated) persistent cross-site scripting. https://www.exploit-db.com/exploits/44563.
27. (2019) Responsive cookie consent 1.8 patches. https://plugins.trac.wordpress.org/browser/responsive-cookie-consent/tags/1.8/includes/admin-page.php.
28. (2018, aug) How does adblock work? https://help.getadblock.com/support/solutions/articles/6000087914-how-does-adblock-work-.
29. (2019) Safely inserting external content into a page. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Safely_inserting_external_content_into_a_page.
30. (2019) nmap network mapper. https://nmap.org/.
31. C.-P. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY,

USA: Association for Computing Machinery, 2009, p. 81–90. [Online]. Available: https://doi.org/10.1145/1595696.1595711

32. (2019) Usage of content management systems for websites. https://w3techs.com/technologies/overview/content_management/all.
33. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203
34. (2019) Wordpress: Plugins. https://wordpress.org/plugins/.
35. (2019) Wordpress cves. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=wordpress.
36. (2019) Wpscan. https://wpscan.org/.
37. (2019) Wordpress: Vulnerability statistics. https://www.cvedetails.com/product/4096/Wordpress-Wordpress.html?vendor_id=2337.
38. (2019) Exploit database. https://www.exploit-db.com/.
39. (2019) Wordpress plugin responsive cookie consent 1.7 / 1.6 / 1.5 - (authenticated) persistent cross-site scripting. https://www.exploit-db.com/exploits/44563.
40. (2019) Navigation timing level 2. https://www.w3.org/TR/navigation-timing-2/.
41. Moz top 500 websites. https://moz.com/top500.
42. (2020) Cve details vulnerabilities by type. https://www.cvedetails.com/vulnerabilities-by-types.php.
43. A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 199–209.
44. G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 249–260. [Online]. Available: https://doi.org/10.1145/1390630.1390661
45. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, 2010.
46. X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393608
47. J. Pan and X. Mao, "Detecting dom-sourced cross-site scripting in browser extensions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 24–34.
48. F. Sun, L. Xu, and Z. Su, "Client-side detection of XSS worms by monitoring payload propagation," in *Computer Security – ESORICS 2009*, M. Backes and P. Ning, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 539–554.
49. (2019) Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.
50. E. Abgrall, Y. L. Traon, S. Gombault, and M. Monperrus, "Empirical investigation of the web browser attack surface under cross-site scripting: An urgent need for systematic security regression testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 34–41.