Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting

William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, Limin Jia
{billy, anupamd, mahmoods, lbauer, liminjia}@cmu.edu

***Abstract*—Cross-site scripting (XSS) vulnerabilities are the most frequently reported web application vulnerability. As complex JavaScript applications become more widespread, DOM (Document Object Model) XSS vulnerabilities—a type of XSS vulnerability where the vulnerability is located in client-side JavaScript, rather than server-side code—are becoming more common. As the first contribution of this work, we empirically assess the impact of DOM XSS on the web using a browser with taint tracking embedded in the JavaScript engine. Building on the methodology used in a previous study that crawled popular websites, we collect a current dataset of potential DOM XSS vulnerabilities. We improve on the methodology for confirming XSS vulnerabilities, and using this improved methodology, we find 83% more vulnerabilities than previous methodology applied to the same dataset. As a second contribution, we identify the causes of and discuss how to prevent DOM XSS vulnerabilities. One example of our findings is that custom HTML templating designs—a design pattern that could prevent DOM XSS vulnerabilities analogous to parameterized SQL—can be buggy in practice, allowing DOM XSS attacks. As our third contribution, we evaluate the error rates of three static-analysis tools to detect DOM XSS vulnerabilities found with dynamic analysis techniques using in-the-wild examples. We find static-analysis tools to miss 90% of bugs found by our dynamic analysis, though some tools can have very few false positives and at the same time find vulnerabilities not found using the dynamic analysis.**

I. Introduction

Cross-site scripting (XSS) is the most frequently reported class of web-application vulnerabilities, constituting 25% of web vulnerabilities reported in 2014 [9]. By compromising client-side browser security using XSS, attackers can gain control over login cookies, passwords, and authentication tokens, and perform application-level actions as users, for example, send emails or make financial transactions [25]. Preventing XSS typically requires website owners to not only sanitize all untrusted inputs to their web application, but also to sanitize all input that could be received by the client's JavaScript interpreter—a task that can be error-prone due to the complexity of web applications and the widespread use of sensitive functions in JavaScript. Document Object Model cross-site scripting (DOM XSS)—a particular type of XSS vulnerability that occurs entirely in client-side JavaScript—is more and more of a threat as JavaScript on the web becomes increasingly complicated. Traditional methods for defending against XSS vulnerabilities in server-side code—for example, server-side taint tracking or web application firewalls—typically do not apply because the vulnerability lies entirely in client code and servers may not even have logs to detect when an attack occurs.

In this paper, we aim to answer the following questions about DOM XSS. Are DOM XSS vulnerabilities becoming more or less common? How do state-of-the-art methods for detecting DOM XSS vulnerabilities compare? Are web developers learning to avoid such vulnerabilities through good coding practices, for example, using encoding schemes or design patterns such as HTML templating? What are the causes of DOM XSS? Do shared libraries or web-content-generation frameworks propagate DOM XSS vulnerabilities across a large number of sites?

To answer these questions, we use a dynamic approach to detect DOM XSS vulnerabilities on the Internet. Prior work showed how to detect DOM XSS vulnerabilities using taint tracking to track flows of attacker-controllable information sources to sensitive sink functions (e.g., `eval` and `document.write`) [8], [22]. The existence of such flows only indicates that data from a source can reach a sink, but does not account for whether the data has been sanitized by the programmer. Thus, once a flow with a potential DOM XSS vulnerability is observed, the flow must be confirmed to be exploitable. In this paper, we show how to more accurately detect whether a flow that is potentially vulnerable is capable of being exploited. Although an attacker can use several types of sources (e.g., cross-origin messages and cookies), we focus, similarly to prior work [22], on confirming flows from URL-based sources. These are of particular interest because, compared to other flows, they are easy for attackers to exploit.

We used this methodology to detect DOM XSS vulnerabilities on Internet. We crawled the homepages and five random subpages of websites on the Alexa Top 10,000 most popular websites list [11]. Compared to previous work [22], we observed both more flows per web page and determined a higher proportion of those flows to be vulnerable, even when using the same methodology as previous work to determine which flows are vulnerable. Using our improved method for determining which flows are vulnerable, we found 83% more vulnerabilities than by using prior methodology [22]. We believe this indicates that DOM XSS vulnerabilities are becoming more common in the four years since the previous study was undertaken.

In addition, we qualitatively examined the code paths that led to the vulnerabilities. We observed, for example, that most of the vulnerabilities did not share code, implying that the vulnerabilities we found are due to custom code, rather than

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23309
www.ndss-symposium.org

the inclusion of buggy shared libraries. We also observed errors in the implementation of HTML templating that allowed XSS vulnerabilities. Templating can be an effective way to prevent DOM XSS vulnerabilities, and is similar to using parameterized SQL queries. We found cases where bespoke templated HTML designs failed to properly encode template values, which attackers could then inject code into.

Finally, using our collected dataset of DOM XSS vulnerabilities, we evaluated static-analysis tools that are designed to detect DOM XSS. In the past, researchers have compared the effectiveness of vulnerability scanners on synthetic datasets [16], [36], whereas we used real-world vulnerabilities. We found that static-analysis tools performed poorly at detecting the vulnerabilities found by the dynamic analysis. However, some static tools were shown to have low false-negative rates and at the same time identify DOM XSS issues not found by the dynamic analysis, suggesting that dynamic analyses and static analyses are finding qualitatively different types of vulnerabilities. Our findings on static-analysis tools suggest that testing using both dynamic and static approaches may be necessary to secure web applications from DOM XSS.

In summary, our contributions are as follows.

- We improve the methodology to confirm DOM XSS vulnerabilities, and find that 83% more detected flows are vulnerable than suggested by prior work [22].
- We empirically analyze the prevalence of and causes behind DOM XSS vulnerabilities. This yields a number of insights for example, that HTML templating is error prone to implement and that DOM XSS vulnerabilities are becoming more prevalent. We also provide guidance for preventing DOM XSS vulnerabilities.
- We compare static-analysis tools that detect DOM XSS vulnerabilities, finding them to detect different vulnerabilities than our dynamic analysis.
- We develop a modified version of Chromium for tracking the taint information of strings, which we are releasing as open source.[1]

Next, in Section II, we provide background on DOM XSS vulnerabilities and compare our work to prior work for detecting DOM XSS vulnerabilities. Then, in Section III, we detail our methodology for crawling the Internet for DOM XSS vulnerabilities, and our improved technique for confirming potentially vulnerable flows. In Section IV we describe the results of our experiments for detecting DOM XSS vulnerabilities and evaluating static-analysis tools for detecting DOM XSS. We describe the limitations of our work in Section V. We discuss the implications of our findings in Section VI, and conclude in Section VII.

II. Background and related work

Here, we cover background and prior work relevant to DOM XSS vulnerabilities. First, we give examples of and general background on DOM XSS vulnerabilities in Section II-A. Then, in Section II-B, we give examples of general XSS defense mechanisms and why those mechanisms fail to adequately apply to DOM XSS. In Section II-C, we cover work that discusses the impact of previously discovered vulnerabilities. Next, we describe prior work on detecting DOM XSS vulnerabilities using taint tracking in Section II-D. We describe prior work on comparing web-application scanners in Section II-E. Finally, we describe how static-analysis tools help prevent DOM XSS in Section II-F.

[1] https://github.com/wrmelicher/ChromiumTaintTracking

```
document.write(
  '<a href="' + document.location +
  '">Link</a>');
```

Fig. 1: Example of a DOM XSS vulnerability. An attacker could inject arbitrary markup using `document.location` as an attack vector by crafting a link that injects an attacker-controlled script into the page. An attacker may execute code by crafting a link like: `http://[website]/[page]#"><script>CODE</script><!--`.

A. DOM XSS vulnerabilities

Cross-site-scripting (XSS) vulnerabilities are a type of injection vulnerability in which an attacker can inject arbitrary code into a running web application to, for example, take control of the data and credentials used in the application. For example, attackers may get access to the websites's cookies (which potentially contain login tokens), or may execute user actions with respect to the compromised website [25]. In XSS, the injected code is JavaScript that runs in a web application with the permissions of the compromised website. Unlike traditional XSS attacks in which an attacker's injection might be the result of a server-side failure to sanitize input, DOM XSS is a relatively new type of XSS vulnerability that occurs purely as a result of JavaScript executing on the client. Figure 1 shows a example. In the example, an attacker could craft a link that breaks out of the `href`'s single quoted attribute and inject an arbitrary script; for example, `http://[website]/[page]#"><script>CODE</script><!--`. This link, when clicked, would execute the attacker-controlled code (**CODE**). An attacker may convince their victims to click on the link using social engineering, or may embed the link in an iframe on a website that the attacker controls. Like traditional XSS bugs, the details depend heavily on the website and the victim's browser. Chromium, for example, does not encode any characters after the '#' symbol, whereas Firefox encodes such characters using URL encoding. Hence, it is not uncommon for a specific exploit to work only in specific browsers [29].

For a DOM XSS vulnerability to be present, there must be a flow of information from a potentially attacker-controlled source to a sensitive sink function. Examples of potentially attacker-controlled sources include: the URL of the document, accessed via the `document.location` JavaScript object; data passed in cross origin messages using the `postMessage` API; cookies, accessed via the `document.cookie` object; and the HTTP referrer accessed by the `document.referrer` JavaScript object and other methods. Sinks can include any mechanism to execute arbitrary code, for example: the `eval` function, `document.write`, JavaScript event handlers (e.g., the

"onclick" attribute), and URLs that have a JavaScript scheme (e.g., <a href='javascript:...'>).

B. Generic XSS defenses

Many methods have been used to mitigate or defend against XSS vulnerabilities in general but do not apply to DOM XSS. Server-side taint-tracking and static-analysis techniques fundamentally cannot be applied for detecting client-side vulnerabilities [14], [37]. Content Security Policies (CSPs) also aim to solve the problem. However, the adoption of CSPs has been limited and developers frequently misconfigure the policies, allowing unsafe code to execute [15], [38]. Web-application firewalls attempt to solve the problem by blocking all requests that match certain patterns (often lists of regular expressions) that indicate an XSS attack is occurring. However, web-application firewalls are known to allow many attacks due to their reliance on simple pattern matching [13], [18]. Furthermore, since DOM XSS exploits might not be sent to the server, the injection may never be visible to a firewall on the network. Dynamic taint tracking, a technique to observe the flows of information throughout a program, has been proposed as a run-time defense against DOM XSS attacks [34]; however, it requires large infrastructure changes to web browsers. Additionally, taint tracking at run time can decrease performance. The effect of such dynamic instrumentation on performance can potentially be small; however, it often has high variability, where a handful of websites have a serious performance decrease [23]. In contrast, our work uses taint tracking to detect vulnerabilities, rather than defending against attacks at run time.

C. Studying the impact of known vulnerabilities

Prior research has studied the impact of known vulnerabilities on websites. Researchers found that in 2017, 37% of websites included at least one version of a library with a known vulnerability [21]. Researchers in that work measured the prevalence of websites including old, outdated versions of 72 popular libraries. Other work has found that many websites include third-party JavaScript that does not take all necessary security precautions [24]. Our focus differs from these works in that, rather than studying the prevalence of already known vulnerabilities, we detect vulnerabilities without prior knowledge of them. In addition, our approach goes beyond vulnerabilities in outdated versions of code in popular libraries and allows us to discover potentially unknown vulnerabilities.

D. Finding DOM XSS vulnerabilities using taint tracking

The state of the art for detecting DOM XSS vulnerabilities is using dynamic taint tracking. This technique marks potentially attacker-controlled sources as "tainted" and propagates information about tainted values throughout the program. For example, the taint-tracking engine might mark the concatenation of one tainted piece of information and one untainted as also tainted. When a tainted string is used in a sensitive sink, the taint-tracking engine may flag this as a potential DOM XSS vulnerability.

A variety of tools and research have used dynamic taint tracking to detect DOM XSS vulnerabilities. The DOMinator tool is a Firefox-based technology that tracks the taint status of strings in JavaScript [8]. It is the oldest tool to apply taint tracking to JavaScript. In 2013, Lekies et al. showed that DOM XSS is prevalent, and introduced a method for detecting DOM XSS using more precise, byte-level taint tracking of JavaScript code, also accounting for the built-in encoding functions used in JavaScript [22]. To generate automated exploits, the researchers used a context-specific exploit generation methodology designed to create a workable exploit by analyzing the context in which the tainted string occurs in the sink. Their technique was tested by performing a crawl of the Alexa Top 5,000 websites searching for DOM XSS vulnerabilities. Their work showed that automatically generated exploits can be created and that DOM XSS vulnerabilities affect 9.6% of domains on the Alexa Top 5,000 websites. Our work uses a similar methodology as Lekies et al.'s work for identifying tainted flows, but we use a new and novel method for confirming whether flows are indicative of DOM XSS vulnerabilities. We describe in detail the similarities and differences between our methodologies and results in Sections III and VI.

Other work, building upon a system for detecting DOM XSS vulnerabilities using browser-agnostic taint tracking [29], provided a method to track taint information and inject an extension that sanitizes injected strings at run time just before those strings are inserted into the sensitive sink functions [28]. The browser-agnostic framework allows detecting vulnerabilities that are specific to certain browsers; however, such vulnerabilities account for a small fraction of all vulnerabilities [29]. Their work focused on their proposed defense mechanism and the capability of using a browser agnostic taint tracking, in contrast to our work which provides a measurement of the prevalence of DOM XSS vulnerabilities and the ability of static-analysis tools to detect DOM XSS vulnerabilities.

Researchers have also shown how to use taint tracking to defend against DOM XSS vulnerabilities at run time [34]. That work began from a list of known DOM XSS vulnerabilities, and showed that in 73% of cases, current client-side filtering technology—the XSS Auditor in webkit-based browsers—fails to filter an attack. The work proposes the use of browser-based taint tracking to more precisely prevent XSS vulnerabilities. However, this requires modification of the browser engine and, as mentioned earlier, can cause performance degradation. The researchers conclude that many domains make use of partly-tainted HTML markup injection, and that blocking all such cases would not be feasible, instead recommending a specific heuristic policy to separate safe cases from dangerous cases.

Additionally, prior work quantitatively examined DOM XSS vulnerabilities [35], finding that while many vulnerabilities are of low complexity, some are the result of highly complex JavaScript interactions. The researchers found that many DOM XSS bugs are the result of vulnerable third-party scripts, missing knowledge about browser-provided APIs, unaware developers, or incompatible first- and third- party code. Our findings about vulnerability complexity and the role of third-party code are similar; we compare them in detail in Sections IV-A and IV-C. Differently from previous work, we also explore the role of advertising domains, the effectiveness of static-analysis tools, the distribution of vulnerabilities across and within domains, and design-level prevention mechanisms such as HTML templating.

E. Web-application scanners

Web-application scanners are commonly used tools to actively test for a variety of security issues in web applications. Scanners use different methods to detect security issues, of which DOM XSS is one. Prior work has surveyed web-application scanners, finding them to overlook many classes of vulnerabilities, and to be limited by their ability to crawl websites; however, that work did not focus on DOM XSS, and also did not examine web vulnerabilities in the wild, instead creating a test environment [16]. Other work reports on the false-positive and false-negative rates of web scanners [36]. That work also tests against a set of manufactured vulnerabilities and not on in-the-wild vulnerabilities and does not focus specifically on DOM XSS vulnerabilities.

F. Static-analysis tools

Static-analysis tools that support JavaScript, such as ScanJS [12] and JSLint [10], have gained popularity. Those tools statically, without executing code, attempt to detect common programming errors in JavaScript, for example, pointing out the use of dangerous functions or undefined variables. In general, static-analysis tools suffer from more false positives than dynamic approaches [27], [31]. However, static-analysis tools are becoming a generally accepted part of the way that software is developed—passing a static analysis without errors is a requirement often listed in style guides (e.g., [1], [5]). In part, this is due to their ability to be run practically and repeatably with little setup, for example, in nightly builds [27]. Some vulnerability detection tools, like Burp Suite [32], also have a static-analysis component. However, the degree to which static-analysis tools can detect and prevent real DOM XSS vulnerabilities is unknown. In addition, JavaScript as a programming language has traditionally been difficult to statically analyze because of its dynamic features (for example, widespread use of the `eval` function and reliance on dynamic typing). In this work, we study the ability of static-analysis tools to detect DOM XSS vulnerabilities in JavaScript.

III. Methodology

Next, we describe the methodology for our experiments to detect DOM XSS vulnerabilities on the Internet. In Section III-A, we describe how we crawled websites and which web pages we visited. In Section III-B, we discuss the specifics of the taint-tracking engine we developed. In Section III-C, we describe how we confirmed vulnerabilities. Finally, we detail the methodology for testing static-analysis tools in Section III-D.

A. Crawling for DOM XSS vulnerabilities

We first crawled the Internet using a browser instrumented to perform taint tracking. The browser collected information about what data flows occurred in the page, and output a log file detailing the flows and the encoding methods applied. Then, for a subset of flows, we tested whether the flow was exploitable by generating example inputs crafted to deliver a payload to the sensitive sink. This methodology builds on the methodology used in prior work for detecting DOM XSS vulnerabilities at scale [22]. We describe the differences from this prior work in this Section and in Section III-C.

```
document.write(
  '<a href="' + encodeURI(document.location) +
  '">Link</a>');
```

Fig. 2: Modified example of Figure 1, in which the code would have a DOM XSS vulnerability if the encoding function was not applied. In this example, the `encodeURI` function encodes the location so that the double-quote character cannot be injected.

To crawl websites, we started by visiting the Alexa Top 10,000 websites, a list of the globally most popular websites [11]. Then, we collected all the links to other web pages on the home page of each website, and randomly selected five web pages that were hosted on the same domain as the original domain to limit our crawl to a manageable size given our resource constraints. This crawl is broader, but more shallow, than Lekies et al.'s [22], and the difference between the two offers the opportunity for new insights about the incidence of DOM XSS vulnerabilities (see Section VI-A). We also obeyed the `robots.txt` directives, which direct automated programs—robots—as to which pages may be traversed [7]. We automated the process of visiting web pages and extracting the links on a page by developing and using a browser plugin. Whenever any web page did not load correctly—for example, because of a timeout—we attempted to load the same page three times. If the failed page was not the top-level page of a domain, we attempted to load a different web page in the same domain. Crawling occurred during the summer of 2017, roughly four years after Lekies et al.'s work was published [22].

B. Dynamic taint analysis

Like prior work [22], we instrumented Chromium to perform byte-precise tracking of the provenance of each byte of strings in JavaScript. We will not focus on the design of our taint-aware browser because it is not a core contribution of our work and the design is similar to prior work. We have released the source code for our modified, taint-aware version of Chromium and V8, the JavaScript engine used in Chromium (see https://github.com/wrmelicher/ChromiumTaintTracking).

To summarize the design of the taint-aware browser: We first allocated space in each JavaScript string primitive for a one-byte taint value that stores the provenance of each byte of the string. This allows taint information to be precisely propagated during string concatenation or slicing. In addition to the provenance of each string byte, each bookkeeping byte also records which built-in encoding methods have been applied. For example, using the `encodeURIComponent` JavaScript function will modify the taint information to reflect that the string has been encoded using the `encodeURIComponent` function. During taint propagation, matching encoding-decoding pairs will cancel each other. For example, if a string is encoded using `encodeURIComponent` and later decoded using `decodeURIComponent`, then the string will be identified as having no encoding applied. The taint information is only stored for string types and not for arbitrary JavaScript objects. This prevents tracking across different data types: for example,

parsing a string into an integer and then writing the integer to a string would remove all taint information.

Our browser checks the arguments of sensitive functions (e.g., the `eval` function or `document.write`; see Appendix VIII for an exhaustive list) for tainted bytes. If an argument contains tainted bytes, then a record is written to a log file describing the flow, including: the type of taint, the locations of the tainted bytes, the sensitive sink function, and a stack trace. Afterwards, we analyze the logs to determine which flows are *potentially* vulnerable to DOM XSS attacks and which flows are not.

Whether a flow is vulnerable depends on the context of the injection in the HTML or JavaScript, the encoding functions that have been applied, and the source and sink types. For example, if we detect that a tainted value is not encoded and begins in the context of an HTML double-quoted attribute, then that flow is potentially vulnerable. However, if the string is encoded using the `encodeURIComponent` built-in function, then a double quoted attribute is not vulnerable because the `encodeURIComponent` function encodes the double quote as "%22". Figure 2 shows code that would have a DOM XSS vulnerability if an encoding function was not applied. This list of potentially vulnerable flows is then tested to decide whether the flow is actually vulnerable to XSS attacks using a process we describe in Section III-C. One example of a potentially vulnerable flow that is not actually vulnerable to DOM XSS attacks is when the application performs custom sanitization of inputs that is not detected by the taint-tracking engine—for example, by halting execution if the input does not match a certain form that is known to be safe. The log files also contain the stack trace for the sink call of each flow. In addition to making the flow more repeatable for post-hoc manual analysis, this allows us to examine the code path that led to the vulnerability (see Section IV-A).

C. Attack confirmation

By crawling web pages using the taint-aware browser described in Section III-B, we generate a list of potentially vulnerable flows. We then simulate an exploit to test those potentially vulnerable flows to decide whether they are actually vulnerable. We experimented with two methods of automatically crafting injections to test: one used by prior work, which appends the injection to the end of the string [22]; and a novel method that attempts to more accurately pinpoint the specific bytes of the string in which to inject a payload. For the purposes of automatically crafting injections we limited ourselves to URL-based sources (e.g., the `document.location.href` object and derivatives like `document.location.search`, `document.location.hash`, etc.). Those types of potential vulnerabilities are straightforward to generate potential exploits for, and therefore can be easily verified to be actual vulnerabilities. For the same reasons, they are the flows commonly targeted by attackers [26].

The first method (termed method A), used in prior work [22], appended the exploit to the end of the URL. The new method (termed method B), which more accurately pinpoints where in the string to inject the payload, attempts to insert the exploit into the bytes of the source string that match the tainted bytes in the sink. An example is shown in Figure 3. The log files contain the information about which bytes of the string are tainted and the semantic source label for those bytes (e.g., from the URL). Therefore, we can infer which bytes of the source will make their way into the sink by examining the string that is injected into the sink and comparing it to the source string. The insight behind this method comes from the observation that many of the values injected into sinks are values of parameters provided in the URL. Our method is designed to capture URL parsing in client-side code. It is relatively commonplace for JavaScript code to manually parse query parameters on the client, for example, by parsing the URL looking for the special characters that signal parameters: ?, &, and =. In this way, the URL is often used to pass parameters to other links or to control the display of the web page. While this method of confirming that a flow is vulnerable is extremely simple, in practice we find the combination of both methods to generate 83% more exploits than just the first method (method A).

Method A: injection at end of URL
Observed URL:
`example.url.com/path?param=test&a=b`
Generated injection URL:
`example.url.com/path?param=test&a=b#`**`INJECT`**

Method B: injection into parameter
Observed URL:
`example.url.com/path?param=test&a=b`
Observed `eval`-ed string:
`var a = 'test';`
Observed taint location:
The 9th through 13th bytes of the string—starting with the first 't' in test and ending with the last 't' in test.
Generated injection URL:
`example.url.com/path?a=b#&param=`**`INJECT`**

Fig. 3: Explanation of injection methods using an artificial example. In Method A, the injection is inserted at the end of the string. In Method B, we attempt to insert the injection into the parameter value that matches the tainted string in the text of the observed argument to the sensitive sink. **INJECT** marks the point of injection.

To test candidate exploits purely in the browser, i.e., without affecting the website, we limit our candidate exploits to the part of the URL string after the hash (the '#' character), as this segment of the URL string is not sent to the website hosting the page, but only processed internally by the JavaScript running in the browser.

We also did not craft actual valid HTML and JavaScript exploits for attack confirmation, but rather crafted a unique string that included characters necessary for an exploit (e.g., the single quote character if injecting a value into a single quoted HTML attribute). In our payload, we injected the string `marker<>'"` and then examined our sink injection log files for this string.

We believe that avoiding the use of valid HTML and JavaScript in simulated exploits and targeting only the portion

of the URL string after the hash—beyond limiting risk to web servers—leads to simulated exploits that are both easier to generate and less likely to be caught by client or server-side filters. Such filters are notorious for being easily bypassable by humans [13], [18]. However, for an automated injection, we wanted our approach to scale to many websites and detect when an exploit could likely be crafted, instead of being filtered by an easily bypassable defense mechanism. To confirm that a flow was vulnerable to DOM XSS attacks, we searched the logs for the unique injection string in the output. To confirm that our methodology did not yield false positives, we randomly sampled 40 flows that our process flagged as vulnerable and manually developed a working exploit. We found that all 40 instances were vulnerable; therefore, we believe that the vast majority of cases found by our automated method were actual vulnerabilities.

After confirming vulnerabilities, we qualitatively examined a subset of these vulnerabilities for insights into the root cases of DOM XSS vulnerabilities. For each vulnerability that we manually analyzed, a researcher manually reproduced the vulnerability based on the saved stack trace in our log files. Then, we distilled the vulnerability to a small amount of code that could describe the flow of data in the vulnerability. These code snippets were then analyzed to extract the themes common to vulnerabilities. We classified vulnerabilities by complexity and also noted other interesting aspects of the code that had the vulnerability. Our results for this analysis are presented in Section IV-C.

D. Static analysis

After we collected a list of confirmed DOM XSS vulnerabilities, one of the analyses we performed was to evaluate the effectiveness of static-analysis tools to detect these vulnerabilities. We sampled our dataset in two ways to create test sets to evaluate the false-positive rate and the rate with which the tested static-analysis tools detect these vulnerabilities. First, we sampled websites that have known vulnerabilities from our dataset of confirmed DOM XSS vulnerabilities, found using methodology described in Section III-C. Then, we sampled from all websites that we visited to measure the false-positive rate. Note that for measuring the false-positive rate, we sampled from all websites, not only from websites where we did not detect a vulnerability. We sampled in this way so that our sampling would not be biased towards sites that might be less buggy. Sampling from our dataset of known vulnerabilities, rather than using manufactured vulnerabilities, has the benefit that we are using real-world bugs.

a) Description of static-analysis tools: We evaluated three tools for detecting DOM XSS vulnerabilities: ScanJS [12], esflow [4], and the static-analysis tools in Burp Suite Pro [32]. We also attempted to test jsprime [30], but were unable to get it to work without crashing. We focused on open-source or inexpensive proprietary tools that statically detect DOM XSS vulnerabilities. There are variety of other, more expensive proprietary vulnerability scanning tools, including: IBM Security AppScan, Acunetix, Trustwave App Scanner, Retina web application scanner, Qualys web inspect, HP Fortify static code analyzer, and Coverity's JavaScript scanner. However, for our application of scanning a large number of domains, these were prohibitively expensive. Prior work has compared a wide variety of these proprietary tools for general purpose vulnerability detection (i.e., not restricted to DOM XSS vulnerabilities), and found them to have comparable error rates to each other [36].

The static-analysis tools that we chose appear to have different tradeoffs. ScanJS is a tool meant to help people avoid coding practices that lead to, among other things, DOM XSS vulnerabilities. As such, it flags code that could be unsafe without aiming to identify whether the code leads to an exploitable bug. For example, it may point out all locations where the `document.write` function was used with a non-static string as an argument. While this is a good practice to avoid, it is not always indicative of a vulnerability. In fact, the majority of cases are benign. Burp Suite attaches a confidence rating to each potential vulnerability that it flags, giving guidance about which findings are most reliable. Burp Suite also receives code from the website by acting as a proxy between the browser and the website, meaning that it has access to code that is dynamically loaded (e.g., by a `<script>` tag added during execution) unlike the other tools; however, it still is not able to analyze code that is dynamically *generated* (e.g., by using the `eval` function). Esflow is unique in that it often attaches source and sink information to its issue reports for easier debugging.

IV. Results

We used the taint-tracking and crawling methodology described in Section III to collect a dataset of tainted flows. We visited 44,722 web pages, which had in total 319,481 frames. One would expect that trying to visit five subpages on each domain, we would have visited 60,000 web pages: 10,000 top level pages and 50,000 subpages. However, we skipped loading 1,761 web pages due to `robots.txt` directives; and we were unable to load 4,094 web pages after three attempts due to timeouts, 462 because Chromium would not load the page (most often due to SSL warnings), and 26 because Chromium crashed when rendering them. Some of the pages unable to be loaded were top-level pages; in that case we also did not visit other pages on that domain.

We describe how we detected DOM XSS vulnerabilities using our dynamic analyses in Section IV-A. Then, in Section IV-B, we use the results from our dynamic analysis to evaluate different static-analysis tools for detecting DOM XSS vulnerabilities. Finally, in Section IV-C, we describe the qualitative trends that we observed from manually analyzing a sample of our dataset.

A. DOM XSS vulnerabilities detected using dynamic analysis

After crawling our set of web pages, we post-processed the generated taint-tracking logs to generate a list of observed data flows. Each flow has a source, through which an attacker could inject code, and a sink, a sensitive function that consumes data derived from the source of the flow. We tracked flows that have sources that could be potentially manipulated by an attacker, and sinks that could potentially execute JavaScript, including functions that directly execute JavaScript (e.g., `eval`), functions that inject HTML (e.g., assigning to innerHTML or calling `document.write`), and JavaScript event handlers. A summary of the sources and sinks that we tracked can be found in Table I.

		Sinks													
		Anchor src	Cookie	Css	Css style attribute	Embed src	HTML	Iframe src	IMG src	JavaScript	Event handler	setTimeout	Location	Script src	**Total**
Sources	Cookie	11,269	256,784	297	297	0	61,164	2,098	115,363	20,469	114	28	582	50,176	518,641
	Message	16,704	18,373	311	311	0	20,974	3,475	70,517	1,182,456	98	73	535	24,393	1,338,220
	Multiple	4	0	0	0	0	9	3	35	0	0	0	0	15	66
	Referrer	62,476	3,670	31	31	0	55,796	3,657	42,193	645	11	11	537	16,659	185,717
	Storage	11,023	4,590	112	112	0	3,712	396	7,146	3,541	9	1	23	9,494	40,159
	URL	226,214	31,150	418	418	15	237,714	137,364	193,200	2,446	914	140	2,711	238,354	1,071,058
	URL hash	1,601	171	2	2	0	1,938	148	2,322	173	0	101	33	2,400	8,891
	URL host	3,383	116,967	19	19	0	17,147	10,035	25,394	389	6	3	308	5,716	179,386
	URL hostname	21,494	612,759	127	127	0	44,903	24,761	104,664	1,001	269	74	400	16,218	826,797
	URL origin	21,225	46	1	1	0	1,801	47,887	3,273	336	0	2	64	1,762	76,398
	URL pathname	20,235	9,807	15	15	0	3,913	1,301	102,945	1,457	628	12	193	13,326	153,847
	URL search	4,549	2,922	0	0	0	5,665	474	13,425	63	0	0	48	2,759	29,905
	URL port	0	0	0	0	0	0	0	2	0	0	0	0	0	2
	URL protocol	82,953	661	92	92	1	94,538	20,746	152,501	123	11	33	356	72,075	424,182
	window.name	2,109	4,504	8	8	0	24,845	160	3,826	12,621	0	3	67	2,374	50,525
	Total	485,239	1,062,404	1,433	1,433	16	574,119	252,505	836,806	1,225,720	2,060	481	5,857	455,721	

TABLE I: Source-to-sink flow counts for different source-sink pairs. Rows in the table are sources and columns are sinks. We focus on the shaded columns and rows in this work. "Cookie" as a sink means assignment to the `document.cookie` object; as a source means data originating from `document.cookie`. "Location" refers to assignment to `document.location`.

Step #		#		as % of total flows	
		this work	25m flows [22]	this work	25m flows [22]
	Seed domains	10,000	5,000		
	Web pages	44,722	504,275		
	Frames	319,481	4,358,031		
1	Total flows	4,140,873	24,474,306		
2	URL*, referrer, window.name sources to JS, HTML sinks	363,034	1,825,598	8.77%	7.46%
	URL* sources to JS, HTML sinks	285,147	‡	6.89%	
3	Flows from step 2 excluding those blocked by encoding methods	97,924	313,794	2.36%	1.28%
4	Flows from step 3 excluding those blocked by natural encoding in Chromium	93,481	‡	2.26%	
5	Flows from step 4 including only URL-based sources	54,954	‡	1.33%	
6	Unique† flows from step 5	5,217	‡	0.13%	
7a	Unique† vulnerabilities from step 6 after exploit step using method A§	1,754	6,167‡‡	0.04%	0.03%
7b	Unique† vulnerabilities from step 6 after exploit step using method A and B††	3,219		0.08%	
	Vulnerable iframe URLs	4,668	‡		
	Vulnerable domains	364	480		
	Unique vulnerabilities as percent of pages visited using method A	4%	1.2%		
	Unique vulnerabilities as percent of pages visited using method A + B	7.3%			

TABLE II: Break down of flows comparing replication of prior work with the same methodology [22]. *) Excludes the JavaScript `location.protocol` property as it is not readily exploitable. †) Applying the uniqueness filter of hosting domain, code location, breakout sequence. ‡) not reported in that work. §) Method A appends the injection to the end of the source string. ††) Method B inserts the injection into the bytes of the source string which match the tainted bytes in the sink after encodings and decodings have been applied. ‡‡) Includes flows from `window.name` sources because that work includes those exploits.

Overall, visiting 44,722 pages resulted in 4,140,873 detected flows. We focus on flows with URL sources and HTML or JavaScript sinks, as these are the most straightforward to exploit. Consistently with that, research has generally focused on examining this subset of flows or found it to account for the majority of exploitable flows (e.g., [22], [29]).

Other flows have preconditions that make automatically exploiting them more difficult at scale. For example, to exploit a cookie flow, an attacker must find a way to manipulate the victim's cookies; for message flows, an attacker must find a potential flow whose code does not check the message origin and also send the message at the proper time, when the receiver is expecting it. With the exception of message flows, URL-based sources account for the largest number of flows to sinks that can execute arbitrary JavaScript (HTML and JavaScript sinks). Hence, these are the flows we analyze, and we show that they lead to many instances of DOM XSS vulnerabilities. Of the 4,140,873 flows we detected, 285,147 (7%) had a URL-based source.

a) Confirmed vulnerable flows: We determine whether a tainted flow is vulnerable as follows. We first discard flows in which the tainted value is encoded using a built-in encoding method, for example, the `encodeURIComponent` function; we are certain that such flows would ordinarily not

Method	# of unique vulnerabilities
Only injection at end	715
Only injection in key-value pair	1,465
Both methods	1,039
Total	3,219

TABLE III: Summary of the injection methods used to confirm different vulnerabilities. "Only at end" refers to the injection method that inserts the injection at the end of the source (Method A). "Only key-value pair" refers to the injection that inserts the injection in the value of a tainted query key-value pair (Method B). "Both methods" refers to cases where either method would have identified the flow.

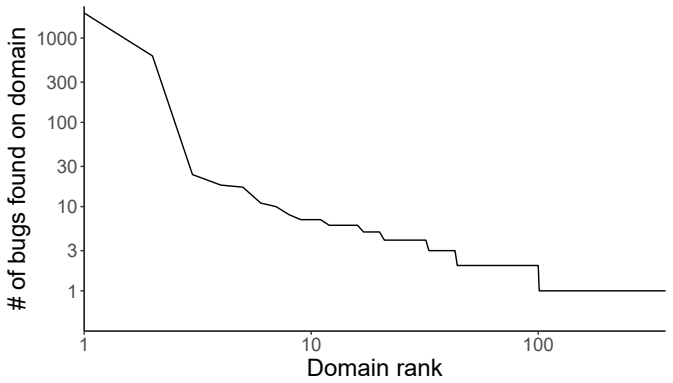

Fig. 4: Distribution of unique vulnerabilities across domains. The y-axis shows the number of unique vulnerabilities found on a particular domain in log scale. The x-axis shows domains sorted by frequency; for example, ten on the axis shows the domain with the 10th most vulnerabilities. For example, the domain with the most vulnerabilities had nearly 1,987 unique vulnerabilities.

be exploitable. This eliminates 66% of the flows we focus on (URL sources to JavaScript or HTML sinks). We next remove from consideration flows that could not be exploited in Chromium due to Chromium's natural encoding of some URL variables (for example, Chromium automatically encodes the content of `document.location.search` to prevent the occurrence of any character that would not be allowed in a URL). After removing those types of flows, we determine which of the remaining 1.33% (54,954 flows) of flows are actually vulnerable by attempting injections. A summary of the number of flows removed at each stage compared to previous work with similar methodology [22] can be seen in Table II.

In our taint-tracking system, we specially mark flows that have multiple, incompatible encodings with a flag that represents the use of multiple encodings, but not which specific encodings or in what order. Such flows accounted for 2% of flows overall. While we did not attempt to determine whether these flows were actually vulnerable, there were 716 unique flows of this type that may have been potentially vulnerable (i.e., that could have been included in row 6 of Table II). If they had been included, they would account for 12% of potentially vulnerable flows.

We used two methods to confirm vulnerabilities—each described in Section III-C—based on where to insert the injected payload: inserting the payload at the end of the URL or inserting the payload into the key-value pair from which we observed a flow to a sink. We found that 45% of the confirmed vulnerabilities we detected were due to flows from key-value pairs, 22% of the vulnerabilities were only the result of inserting the payload at the end of the source, and 32% of the vulnerabilities were observed to work with both methods.[2] Table III shows the breakdown of how many of the 3,219 unique vulnerabilities came from which injection method. Both methods of injection work in cases where the entire URL is concatenated with markup (i.e., `document.location`, rather than a specific substring, is included in markup). Our key-value pair injection method identifies vulnerabilities that involve parsing URL parameters, while inserting the injection at the end identifies vulnerabilities in which part of the path or URL besides the URL parameters is used as part of the parameter to the sink.

To count unique vulnerabilities, we removed duplicates using the same method used as Lekies et al. [22]: unique bugs are identified by their domain, their location in the script, and the context (e.g., inside a double-quoted attribute or the name of an element attribute) of the tainted section of the string argument to the sensitive sink.

We also computed the number of unique vulnerabilities across different domains, as shown in Figure 4. We found that the majority of vulnerabilities come from a handful of domains, and that many domains had only a few unique vulnerabilities or one vulnerability: the ten domains with the most vulnerabilities had in total 2703 unique vulnerabilities; the remaining 354 domains accounted for the remaining 516 vulnerabilities.

Interestingly, when performing the vulnerability confirmation crawl we observed vulnerabilities in six iframe URLs that were not previously seen in our first crawl. These iframe URLs were part of the confirmation crawl because either they or the top-level pages that included them had previously been marked as potentially containing a vulnerability. The difference in time between collecting data and confirming vulnerabilities was nine days.

b) Vulnerability attribution by domain and domain category: We next attempted to shed light on the cause of the vulnerabilities that we observed by examining where they occurred. Were they due to third-party scripts, old versions of popular libraries, custom code for each website, or other causes? For this measurement, we used the URL of the frame where the vulnerability was found, since this is the context in which an attacker would be able to execute JavaScript, rather than using the URL of the top-level frame. We used the location of the sink as a starting point for determining to which entity to attribute the vulnerability. In particular, we examined the distribution of vulnerabilities in three ways: (1) the domain on the iframe in which the script executed; (2) the domain on which the script was hosted (web pages often import scripts from other domains); and (3) the domain of the top-level page that the user was visiting. Rather than reporting results about individual domains, we report them by the topic category of the

[2]Numbers do not add up to 100% due to rounding.

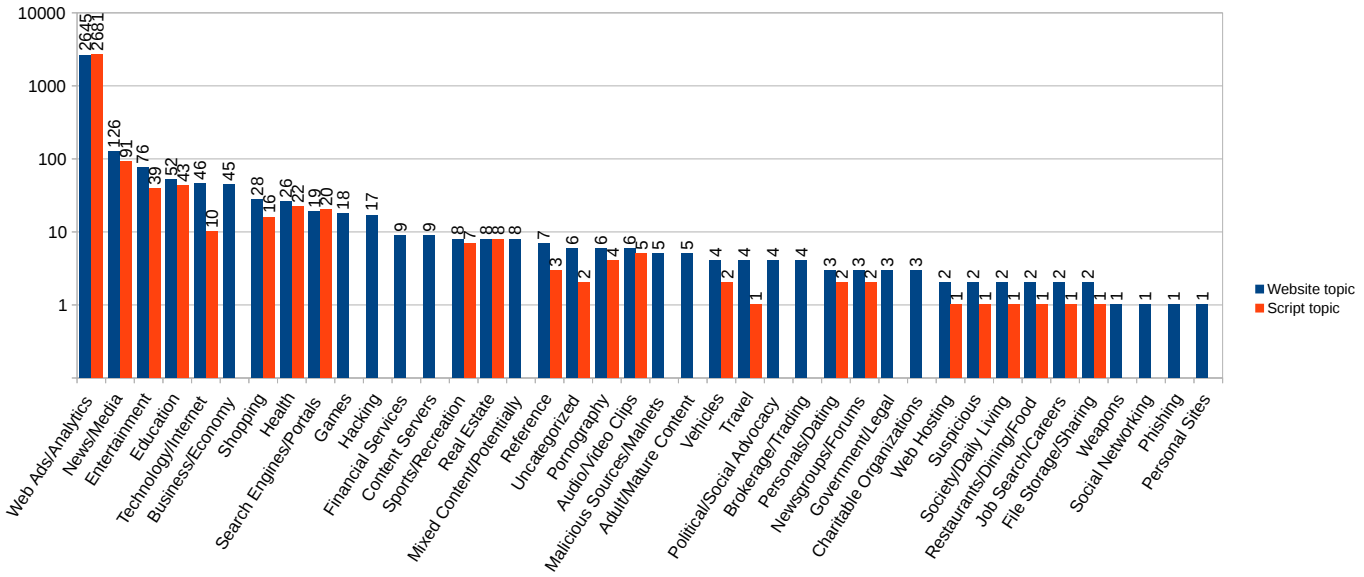

Fig. 5: The count of URL domains and script domains in different website categories. The bar height shows the number of script domains or frame domains with the corresponding category in our dataset of unique vulnerabilities. The y-axis is in logarithic scale. Script domains are the domains that the vulnerable scripts were hosted on. Frame domains are the domain of the frame where the vulnerability was located. Note: the numbers for script domains do not add to 3,219 because some sinks did not have a script URL. This may happen when the sink location is in dynamically generated code.

domains. We use the Blue Coat K9 classification of domains into topics [2] for this purpose.

We found that the vast majority, 2,645 of 3,219 of our unique vulnerabilities (82%), were found to execute inside iframes with domains that were known to serve web advertisements or perform analytics. Other domain types that accounted for many vulnerabilities included shopping and news.

We also analyzed what type of domains hosted the scripts in which we found vulnerabilities. Similarly to the above result, we found that 2,681 of 3,219 vulnerabilities (83%) were in scripts hosted on advertising and analytics domains. Figure 5 shows the analysis of the types of script domains and website domains with confirmed vulnerabilities. For this measurement, we used the domain of the script where the sink function call was found. While advertising domains were the most popular source of vulnerable scripts, our data-collection infrastructure did not capture enough information to similarly categorize scripts that did not have potentially vulnerable flows. Hence, while we can report that, in web pages that had at least one flow, 38% of the time the flow originated in a script that was categorized as an advertising script, we cannot determine whether the fraction of advertising-domain scripts that was vulnerable was greater than the fraction of scripts from other domain categories.

We matched the unique vulnerabilities that we found with the top-level web pages that contained those vulnerabilities in our dataset. Many of the vulnerabilities that we found were on subframes of other web pages, and we wanted to understand how much exposure users would have if they visited the top-level pages in our dataset. Table IV shows the categories of these top-level web pages. Note that there are significantly more data points than unique vulnerabilities because some

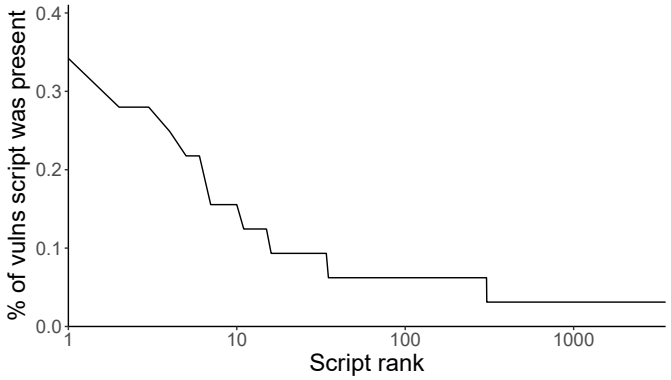

Fig. 6: The percent of stack traces from the dataset of unique vulnerabilities that scripts were found in. The x-axis shows the rank in log scale of the scripts in a sorted list; for example, 10 shows the script that was 10th most frequently present in stack traces. The y-axis is the percent of script URLs that were less than that rank. For example, the first script was present in the stack traces of 0.34% (11 of 3,219) unique vulnerabilities.

vulnerabilities were present in web pages that were subframes of multiple web pages. In contrast to Figure 5, where the most popular category was web ads and analytics, here the popular topics are news/media (27.7%) and entertainment (12.9%).

In total, for 282 (8.8%) of the vulnerabilities we found the domain the script was hosted on was different from the domain of the iframe in which the script executed. This suggests that while a non-trivial fraction (8.8%) of vulnerabilities may be caused by developers relying on third-party scripts, the

%	Count	Category of top level website
27.7%	2856	News/Media
12.9%	1337	Entertainment
9.9%	1026	Technology/Internet
5.1%	523	Games
4.4%	453	Education
4.1%	424	Sports/Recreation
3.6%	376	Reference
3.5%	362	Shopping
2.8%	289	Hacking
2.7%	280	Business/Economy
2.4%	246	Society/Daily Living
2.0%	202	Mixed Content/Potentially Adult
1.7%	178	Newsgroups/Forums
1.6%	164	Health
1.4%	143	Search Engines/Portals
1.2%	119	Brokerage/Trading
1.1%	114	Political/Social Advocacy
1.1%	113	Financial Services
1.0%	107	Travel
0.9%	98	Vehicles
0.8%	80	Restaurants/Dining/Food
0.7%	76	Uncategorized
0.7%	71	Real Estate
0.6%	62	Job Search/Careers
0.5%	55	File Storage/Sharing
0.5%	51	Audio/Video Clips
0.5%	50	Government/Legal
0.4%	46	Software Downloads
0.4%	43	Religion
0.4%	43	Pornography
0.4%	39	Adult/Mature Content
0.4%	37	Alternative Spirituality/Belief
0.3%	31	Email
0.3%	28	Social Networking
0.3%	28	Personal Sites
0.2%	24	Malicious Sources/Malnets
0.2%	21	Office/Business Applications
0.2%	21	Auctions
0.2%	18	Phishing
0.1%	14	Humor/Jokes
0.1%	13	Suspicious
0.1%	13	Charitable Organizations
0.1%	8	Scam/Questionable/Illegal
0.1%	7	Web Hosting
0.1%	6	Intimate Apparel/Swimsuit
<0.1%	5	Weapons
<0.1%	5	Placeholders
<0.1%	5	Gambling
<0.1%	4	Web Ads/Analytics
<0.1%	4	Personals/Dating
<0.1%	3	Nudity
<0.1%	2	Military
<0.1%	2	Chat (IM)/SMS
	10325	**Total**

TABLE IV: Categories of top level domains that contain an iframe with a DOM XSS vulnerability. The count column shows the number of top level pages in a category that contained a frame with a vulnerability. The percent shows the percent of top level pages with that category.

vast majority of vulnerabilities are in the developers' own scripts (or at least scripts hosted locally on their domains). The fraction is smaller than reported in prior work, which found that 22% of code attributable purely to an error by a third party [35]. This difference could be the result of our different methodology for confirming DOM XSS vulnerabilities.

c) Vulnerability attribution by script: We additionally examined the scripts in the entire stack trace for each vulnerability. The goal of this analysis was to determine whether some scripts occurred in stack traces of vulnerabilities particularly often. Such scripts could be good candidates for adding encoding functions, or for other remediation, as that would prevent many vulnerabilities. At the same time, if some scripts occur in many stack traces of vulnerabilities, this could indicate that developers misunderstand how to correctly use that script.

For our 3,219 confirmed vulnerabilities, we identified the scripts in the stack trace of each vulnerability, and then counted how many stack traces each script was present in. For this analysis we removed jQuery from our results because many websites use various HTML rendering functions (e.g., the `html` or `append` jQuery methods) that are working as intended, but misused by the caller. Hence, we removed any script name that contained the string "jquery" without respect to character case.

We found that the majority of vulnerable scripts was present in only one unique vulnerability stack trace—implying that the causes of vulnerabilities are unique. Figure 6 shows the percentage of stack traces that each script is seen in.

d) Vulnerabilities in commonly blocked content: Internet folklore has often claimed that ad blocking software protects your computer from XSS vulnerabilities common in ad networks [19]. Because we found that many of both the target web pages and the vulnerable scripts were located on domains that hosted advertising, we tested how much protection a normal user who uses an ad blocker would have from such vulnerabilities. For this analysis, we used the adblockparser Python library [6] to simulate what scripts would not be executed if the user was running ad blocking software that obeyed the rules defined in the Adblock EasyList, a popular rule list of advertising content to block [3]. We counted a vulnerability as being blocked if the Adblock EasyList would block either the script or the entire target URL when the target URL is loaded inside an iframe and not as the main frame. We found that, of the 3,219 unique vulnerabilities, 2,039 (63%) would have been blocked by this simulated configuration of Adblock.

B. *Effectiveness of static-analysis tools*

We next examine whether the vulnerabilities we found could be detected at development time using off-the-shelf static-analysis tools. We find that most could not, although static-analysis tools sometimes found additional bugs.

More specifically, with our dataset of confirmed vulnerabilities IV-A, we tested static-analysis tools to evaluate their ability to find the same vulnerabilities that the dynamic analysis found. To target JavaScript, dynamic analysis traditionally is seen as having fewer false positives [27]; however, static analysis is often more helpful for programmers during development because of the lack of customized analysis for adding new code—developers can set up a static-analysis toolchain once to automatically check new code. In addition, static analysis can be more complete—able to detect vulnerabilities in code that was not executed on a particular run of the program. The majority of the vulnerabilities that were caught in our experiment only by static-analysis tools were in this category.

a) Overlapping vulnerabilities: We compared the rates at which different tools, described in Section III-D, found the vulnerabilities that we had previously compiled using the dynamic analysis. We found that the tools we tested usually failed to detect the DOM XSS vulnerabilities from our dataset.

	% of detected vulnerabilities	# of reported issues
Esflow	0%	4
ScanJS	8%	2700
Burp Suite	10%	39

TABLE V: The percent of vulnerabilities detected by the dynamic analysis that were detected by different static-analysis tools out of a total of 50 web pages with known vulnerabilities.

Tool	False positive %	# of reported issues
Esflow	95%	19
ScanJS	100%	3764
Burp Suite	0%	36

TABLE VI: Empirical false positive rate computed from a random sample of 20 reported errors over 50 randomly sampled web pages.

The full results comparing different tools are provided in Table V. Notably, while Burp Suite, the most promising tool, had a low rate of finding the same errors as the dynamic analysis, it pointed out many potential issues not found by the dynamic analysis. We next describe these findings in more detail.

We randomly sampled 50 of the 3,219 unique bugs we found using our dynamic analysis. Then, for the two tools that scanned JavaScript code, we downloaded the web page where the vulnerability was located, and all the accompanying JavaScript, and used the downloaded JavaScript as input to the tools. We included any JavaScript embedded in HTML in addition to externally loaded JavaScript. Because the tool was examining the web page statically, we were not able to include JavaScript that is dynamically loaded during the execution of the page. For Burp Suite, we loaded the web page in the Chromium web browser and connected Burp Suite to the browser via a proxy. In this way, all the JavaScript requested by the browser was analyzed by Burp Suite's static-analysis tool. Therefore, it was able to access external scripts that were dynamically loaded during execution.

We counted a tool as successfully identifing one of these vulnerabilities if it output any error message related to DOM XSS referring to the code where the sink for the vulnerability was located. For Burp Suite and esflow, we manually reviewed all messages and counted how many referred to the exact line number and character offset of the sink that was vulnerable. For ScanJS, we counted messages that referred to the same script location (line number and script name) as the vulnerable sink, provided that they warned of a potential XSS vulnerability. We explicitly terminated the analysis of any program that took longer than one hour. This affected four of the 50 web pages tested with esflow. In practice, we believe that this is realistic in that static-analysis tools must give results within a reasonable time to be useful to developers. We do not count as matches cases in which the tool detects a vulnerability that was not a part of our dataset.

b) False positive rates: We compared the false positive rates of the tools by sampling tool output and manually deciding whether the particular snippets of code that the tool flagged could be exploitable. We found Burp Suite to have no false positives, while the other tools had many false positives. Table VI shows the results for all tools.

We randomly sampled 50 out of all the URLs that we visited in our dataset. We randomly sampled from all URLs, and not just URLs for which we did not confirm vulnerabilities, so that our results for false positive rates would not be biased towards websites that are more secure, and therefore more likely to have false positives than true positives. We then analyzed the scripts on those pages with each of the tested static-analysis tools. Each tool would report findings on the pages related to DOM XSs. We randomly sampled 20 of those findings, except for esflow, which only reported 19 findings. We then manually examined each finding, and the piece of code that it referred to, to determine whether that piece of code could be exploitable. In doing so we aimed to simulate how a developer assessing the same code and reviewing the output of the tool would categorize the bug.

Our guiding criteria for manually counting true and false positives in the tools' output was to look for flows from exploitable sources (e.g., URLs, cross-origin messages) to exploitable sinks (e.g., `document.write`, `eval`) without encoding that would render the flow benign. Thus, we counted flows as true positives even if the identified block of code was not executed during the page load during which our dynamic analysis detected a vulnerability. This may happen because the function is dead code, or because that particular page load did not happen to execute the vulnerable function. We believe this method of measuring false positives is a conservative estimate, because in reality some of the identified vulnerabilities may be in dead code. We also did not include findings that were not related to DOM XSS vulnerabilities—for example, warnings about bad coding practices—as either false positives or false negatives. We aimed to measure the number of actionable vulnerabilities that could be detected by our tested static-analysis tools.

Deciding whether a "bug" reported by a static-analysis tool is exploitable was a judgment call. We believe that any bias in the judgment is likely to be biased towards marking something as non-exploitable when it could be exploited, because it is easier to show how a piece of code might be dangerous, but much harder to confirm that code is safe in all cases. In practice, we believe this closely matches how an engineer reading the output of such a tool would label the output.

C. Qualitative trends in DOM XSS vulnerabilities

To gain greater insight into the causes that give rise to DOM XSS vulnerabilities, we manually, qualitatively analyzed two subsets of the vulnerabilities detected by our dynamic analysis. First, we randomly selected from the unique vulnerabilities; however, we noticed that a large portion of these vulnerabilities was semantically very similar, despite being unique bugs according to our uniqueness criteria (script location, hosting domain, and context; also used in previous work [22]). Therefore, we also selected a separate subsample of vulnerabilities, in which we first randomly selected 20 domains on which vulnerabilities had been found, and then selected a random vulnerability on each domain. This allowed us to get a sample that is conceptually more representative

of the types of bugs that occur across different domains, and hence of the types of problems that are likely to be encountered by different organizations. A summary of the trends we observed is located in Table VII.

a) Vulnerability complexity: First, we found that some bugs were extremely simple, such as concatenating the entire URL into an HTML or JavaScript execution function. Examples of why this happened were creating a form where the form submission attached a return URL for the current page, or passing the web page's URL as a query parameter for the source of an iframe. In addition to simple concatenation, we also found cases where the URL was stored in a non-local variable that could be assigned to in code that was far away from the sink (e.g., in a different file or function).

b) Failed mitigation behaviors: In addition, for eight of the forty vulnerabilities the relevant code was more complex and spanned multiple functions. Three of those eight utilized custom template processing that did not perform encoding based on the context in the template, resulting in insecure templating code. Another vulnerability was due to an attempt to perform custom, but highly incomplete, filtering—removing all instances of `<script>` tags, but still leaving open many other ways for an exploit to occur, for example, by using event handler code like `<img src=x onerror='INJECTION'/>`. For two other vulnerabilities, code involved in the flow was dynamically generated using the `eval` function, meaning that such code would typically not be visible to static-analysis tools.

We did not observe any failed attempts to use custom encoding functions. Combined with the fact that many of the bugs were shallow—a finding echoed by [35]—this suggests that perhaps engineers were not aware that URLs could contain characters that could be used to inject markup.

When manually reproducing vulnerabilities, we also observed cases where complex control-flow paths must be followed to execute the vulnerable piece of code. For example, we could not reproduce one specific vulnerability until realizing that the vulnerable code was only executed if the screen width was larger than 1,024 pixels, as it had been in our original data collection. In another case, the vulnerable section of code was executed on some page loads but not on others. We discuss code coverage further in Section VI.

V. Limitations

Despite being less straightforward to automatically exploit in the context of a live website, other types of flows besides the ones we focused on (URL to HTML and JavaScript flows) may also be vulnerable. For example, we observed (during manual analysis) a flow from a cookie source to an HTML sink that could be exploitable by a second flow from a URL source into a cookie sink on the same web page. The page could be exploited by crafting a special URL, from which content would flow through the document's cookie into the HTML sink. Other work has observed XSS vulnerabilities that derive from cookie sources and could be exploitable by web attackers [39]. Vulnerabilities that exploit the JavaScript `postMessage` API have also been reported [33]. Location sinks can be leveraged to create more potent phishing websites, in which an attacker may craft a URL that points to the target website but is redirected to a phishing website via assignment to `document.location` in JavaScript. A victim might assume that they were on a benign website because the hostname in the URL they clicked on was benign. We speculate that these other types of flows might be similar to the types of flows we study here and would be a good avenue for future work.

	By domain	By unique bug
Simple concatenation	8	1
Simple except for variable usage	4	18
Spans multiple functions	8	1
Custom templating	3	0
Custom filtering	1	0
Dynamically generated code	2	0

TABLE VII: Description of types of vulnerability qualities observed during qualitative coding of bugs. We randomly sampled our vulnerability dataset in two ways, by domain and by unique vulnerabilities; each sample contained a total of 20 vulnerabilities. "Simple concatenation" refers to bugs that were a simple concatenation of the entire source with HTML or JavaScript markup. "Multiple functions" refers to vulnerabilities that spanned multiple functions. "Simple except for variable usage" refers to bugs where a variable with the source value was concatenated with HTML or JavaScript code. Custom templating refers to code that attempted to use a custom templating library, but without encoding. Dynamically generated code refers to instances where code involved in the bug was dynamically generated, and would generally be outside of the abilities of static analysis.

We sampled only a subset of the web pages on the Internet, and on the pages we sampled, we did not exercise much dynamic functionality—for example, by clicking on webpage elements or entering text—nor were we able to visit web pages behind log-in barriers. On one hand, this allowed our analysis to scale to large numbers of vulnerabilities and websites, but on the other, the vulnerabilities we detected may not be representative of all vulnerabilities. Nonetheless, we found many vulnerabilities through our analyses. Our manual analysis of vulnerabilities may also exhibit similar biases: We performed a more in-depth analysis only of a subset of our results. This subset was by necessity small so that it would be feasible to manually analyze. We do not suggest that the examined vulnerabilities are representative of our dataset, but we analyzed them in depth to give greater insight into at least some vulnerabilities.

Due to slight differences in methodology, the comparison of our results to previous work may not be perfectly accurate. Differences in results may be due to implementation differences, although for the parts of methodology that are shared between our work and Lekies et al.'s [22], we tried to reproduce previous methodology faithfully.

VI. Discussion

We performed this study to measure the prevalence of DOM XSS vulnerabilities, evaluate and inform the design of static-analysis tools, and assess the viability of other methods for preventing DOM XSS vulnerabilities. We first discuss

how the raw results of our measurement study compare to previous work that used similar methodology to measure XSS vulnerabilities (Section VI-A), teasing out which differences are the result of methodology and which reflect a change in the prevalence of DOM XSS vulnerabilities. We then leverage our quantitative and qualitative analyses of the sources and nature of DOM XSS vulnerabilities to discuss the weaknesses of some suggested countermeasures (Section VI-B). Finally, we further interpret the results of our examination of static-analysis tools and suggest how these tools could be improved to catch more DOM XSS vulnerabilities (Section VI-C).

A. Comparing measurements on DOM XSS vulnerabilities

Our methodology for detecting DOM XSS vulnerabilities replicates and builds on Lekies et al.'s [22]. We extend Lekies et al.'s methodology by adding another method for determining whether a bug is exploitable, namely, inserting a potential exploit into query key-value pairs rather than just at the end of the URL. When inserting the injection at the end of the URL, we find that a roughly similar fraction of flows is vulnerable as reported by Lekies et al. [22]. However, using both methods of inserting the injection, we identify 83% more confirmed vulnerabilities than when just inserting the exploit at the end. This suggests that previous work, as well as our own, may substantially undercount the number of vulnerable flows.

Our methodology differed from that of Lekies et al., in that we visited twice as many top-level domains, but fewer subpages for each domain (see Section III-A). We believe this is the main cause of different findings for the number of domains that have at least one vulnerability. Previous work found 9.6% of domains to have at least one vulnerability, while we found 3.8% of domains to have one. Interesting, this shows that vulnerabilities are not systemic, i.e., a domain that has at least one vulnerability is not likely to have that vulnerability (or different ones) on a preponderance of pages. Since some parts of pages hosted on the same domain are often shared across most pages, this implies that DOM XSS vulnerabilities are usually not in this shared content.

Where our methodology and that of Lekies et al. are most directly comparable—when relying only on the simpler method of confirming vulnerabilities and examining the ratio of vulnerabilities to number tainted flows or to number of pages visited—our results are generally similar, although overall our results suggest an increase in the number of vulnerabilities over time. In our work, we find more flows per page—on average, 92.6 flows per page compared to an average of 48.5 flows per page in Lekies et al.'s work. Additionally, normalizing by the number of flows we found more vulnerabilities: We found 0.04% of flows to be vulnerable, while Lekies et al. reported 0.03%. Normalizing by the number of pages visited, we also found more vulnerabilities: 1,754 vulnerabilities on 44,722 pages (3.9%); previous work found 6,167 vulnerabilities on 504,275 web pages (1.2%). We speculate that this difference is because JavaScript programs are becoming more complex, and as a consequence DOM XSS vulnerabilities are becoming more frequent.

B. Preventing DOM XSS

In Section IV-A, we showed that the unique vulnerabilities typically did not involve many of the same scripts: the stack traces of the exploits of different vulnerabilities were generally composed of different scripts. We interpret this to mean that most DOM XSS vulnerabilities are due to custom code, and not library code that is shared by many domains.

One way to prevent DOM XSS vulnerabilities is to detect them before the software is released. We believe that a promising direction for finding DOM XSS vulnerabilities at scale is using techniques that analyze larger portions of the program space. The problem of code coverage of dynamic analysis techniques is not new or specific to DOM XSS vulnerabilities; however, it can be a bigger hurdle for large-scale analysis of web applications than for traditional programs. Running many versions of a web application may require a large amount of network bandwidth for reloading web pages, which can make it difficult to scale. Solutions that avoid reloading the page to explore more sections of the program should be explored, as well as methods to force execution down alternate program paths. In particular, work on fuzzing parameters for traditional XSS [17] and on forcing JavaScript execution through different code paths [20] holds promise.

Our analysis also provides additional evidence of the risks of developing custom versions of common design patterns. While using design patterns for templated HTML—a practice analogous to parameterized SQL queries—is generally a good approach to preventing DOM XSS vulnerabilities, it is important to correctly implement the details. For example, we observed three instances of bespoke HTML template implementations that did not apply encoding functions to the values of the templates. In general, custom templating implementations can be error prone, because differences in context can be easy to overlook. For example, to be safe from XSS, a value in a templated HTML statement that is inside a script tag must first have encoding applied for the HTML parser and then for the JavaScript parser.

C. Static-analysis tools

We next further interpret the results of running the static-analysis tools on a sample of vulnerable scripts (see Section IV-B). These results suggest that many vulnerabilities may currently escape both static and dynamic analyses. We also leverage our results to suggest ways to extend static-analysis tools to catch more bugs.

In Section IV-B, we measured the false positive rate of different static-analysis tools and the rate at which tools correctly identified vulnerabilities from our dataset of known vulnerabilities using dynamic taint tracking. For the false positive rate, we empirically sampled the tools output and manually decided whether a tool's finding was a false positive or a true positive. However, we were unable to empirically measure the false negative rate overall. This is because it was not feasible for us to know all possible vulnerabilities in non-contrived application. Instead, we measure the rate at which static-analysis tools can detect known bugs that are detected with a different methodology.

Our analysis of Burp Suite, the best-performing static-analysis tool we tested, showed low false-positive rates but also an inability to detect most of the vulnerabilities identified by the dynamic analysis. Together, these two measurements imply that the static-analysis tools were detecting largely different

vulnerabilities that our analysis. The dataset of vulnerabilities with which we tested static-analysis tools, however, was limited to vulnerabilities detected through our dynamic analysis. In our test, we visited a large number of web pages but did not attempt to exercise much of the web application's functionality, for example, by clicking on fields, entering data into forms, or sending messages to pages. It is likely that such activities would reveal more vulnerabilities. In addition, in our dataset, we found many of the vulnerabilities to be shallow, in that they involved a straightforward concatenation of data from a source into the parameter to a sensitive sink function. This is similar to prior findings [35]. Indeed, it could be that the majority of vulnerabilities are more complex and would be better detected by static-analysis tools. Given that we know that there are many bugs that escape either analysis, we speculate that there may be many more bugs that escape both analyses.

ScanJS generally appears to identify poor coding practices that lead to DOM XSS vulnerabilities, rather than detecting such vulnerabilities. Indeed, many of the suggestions that the tool gives revolve around the use of certain functions being dangerous (e.g., `document.write` or `eval`). While ScanJS unfortunately did not detect many vulnerabilities on our dataset, we believe it to be useful for, e.g., enforcing coding standards.

Based on our experiments, we can make some recommendations for improving static-analysis tools. One area where static-analysis tools could improve is the ability to track flows across function boundaries. We found a non-negligible number of such vulnerabilities (20% in the domain sampling setting in Section IV-C) and tracking such flows can be difficult statically, especially when there are many branches. Another aspect of static-analysis tools that could use improvement is the ability to track flows that go through objects. For example, a tainted string is sometimes stored as the key or value in a JavaScript object and later used in a computation. Finally, one constraint of static-analysis tools that is especially limiting in JavaScript is the inability to analyze dynamically generated code. A hybrid static-analysis tool that analyzes new code before it is executed in the browser might be better able to detect such vulnerabilities.

VII. Conclusion

We studied how to detect and prevent DOM XSS vulnerabilities in JavaScript code. In this work, we improved on the methodology to confirm DOM XSS vulnerabilities, finding 83% more vulnerabilities than by using previous methodology applied to the same dataset. We used our methodology for detecting DOM XSS vulnerabilities to empirically measure the prevalence of DOM XSS vulnerabilities on the Internet, finding them to be more common now than when previously measured in 2013. With our collected dataset of DOM XSS vulnerabilities, we also compared the ability of static-analysis tools to detect the same bugs that dynamic analysis techniques found, finding static-analysis tools to detect different types of bugs, with little overlap. A summary of our findings can be found in Figure 7. We are in the process of notifying the website owners of the vulnerabilities we discovered.

Measurement	
• Our key-value pair injection method in conjunction with prior method found 83% more vulnerabilities than found using only prior method of injection [22].	Sec. IV-A
• We identified what has changed and what remains the same in DOM XSS over a 4-year span by building on top of a prior experiment.	Sec. VI-A
XSS trends	
• We found more tainted flows overall and a higher rate of vulnerable flows than previous work, which suggests that DOM XSS is getting worse.	Sec. VI-A
• Vulnerabilities are concentrated on a small number of iframe owners and script hosting sites.	Sec. IV-A
• 83% of vulnerabilities are due to code hosted on advertising and analytics domains.	Sec. IV-A
What contributes to XSS	
• DOM XSS vulnerabilities are likely not systemic within domains.	Sec. VI-A
• Vulnerabilities are often in unique, custom code, not in shared libraries.	Sec. IV-A
• Incorrectly implemented bespoke HTML templating, a defense against XSS, introduces XSS vulnerabilities.	Sec. IV-C
XSS prevention	
• Ad blocking would block many of the vulnerabilities and is an effective client-side protection tool.	Sec. IV-A
• Incorrectly implemented templating leads to vulnerabilities and possibly false sense of security.	Sec. VI-B
• The three popular (low-cost or free) static-analysis tools we tested are not effective at finding the vulnerabilities found using our dynamic tool; however, Burp often finds vulnerabilities not found by our tool.	Sec. VI-C

Fig. 7: Summary of findings.

Acknowledgments

The authors would like to thank Cara Bloom for providing comments on a draft of this work. This work was supported in part by gifts from John & Claire Bertucci, by CyLab at Carnegie Mellon University via a CyLab Presidential Fellowship, and by the National Science Foundation via grant CNS1704542.

References

[1] "Airbnb JavaScript style guide," https://github.com/airbnb/javascript.

[2] "Blue coat k9," http://www1.k9webprotection.com/.

[3] "Easylist filter for AdBlock," https://easylist.to/.

[4] "esflow: Elegant, fast JavaScript static security analyzer for finding issues like DOM XSS," https://www.npmjs.com/package/esflow.

[5] "Google JavaScript style guide," google.github.io/styleguide/jsguide.html.

[6] "Python parser for Adblock Plus filters." [Online]. Available: https://github.com/scrapinghub/adblockparser

[7] "The web robots pages," http://www.robotstxt.org/.

[8] "DOMinator," 2011. [Online]. Available: https://github.com/wisec/DOMinator

[9] "Cenzic application vulnerability trends report 2014," 2014. [Online]. Available: https://www.info-point-security.com/sites/default/files/cenzic-vulnerability-report-2014.pdf

[10] "JSLint," http://jslint.com/, 2015.

[11] "Alexa top sites globally," http://www.alexa.com/topsites/countries/US, 2017.

[12] "ScanJS," https://github.com/mozfreddyb/eslint-config-scanjs, 2017.

[13] K. Bijjou, "Web application firewall bypassing how to defeat the blue team," OWASP open web application security project, 2015.

[14] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," in *Proc. DIMVA*, 2008, pp. 23–43.

[15] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems?: Evaluating the effectiveness of content security policy in the wild," in *Proc. CCS*, 2016, pp. 1365–1375.

[16] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Proc. DIMVA*, 2010, pp. 111–131.

[17] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "KameleonFuzz: evolutionary fuzzing for black-box XSS detection," in *Proc. CODASPY*, 2014, pp. 37–48.

[18] V. Ivanov, "Web application firewalls: Attacking detection logic mechanisms," Blackhat USA, 2016.

[19] A. Jones, "On widespread XSS in ad networks." [Online]. Available: https://blogs.msmvps.com/alunj/2016/04/09/on-widespread-xss-in-ad-networks/

[20] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-Force: Forced execution on JavaScript," in *Proc. WWW*, 2017, pp. 897–906.

[21] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *Proc. NDSS*, 2017.

[22] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *Proc. CCS*, 2013, pp. 1193–1204.

[23] B. Livshits, "Dynamic taint tracking in managed runtimes," *Technical Report MSR-TR-2012-114, Microsoft*, 2012.

[24] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proc. CCS*. ACM, 2012, pp. 736–747.

[25] OWASP, "Cross-site scripting." [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[26] ——, "DOM based XSS." [Online]. Available: https://www.owasp.org/index.php/DOM_Based_XSS

[27] ——, "Static code analysis." [Online]. Available: https://www.owasp.org/index.php/Static_Code_Analysis

[28] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proc. ESEC/FSE*, 2015, pp. 272–283.

[29] ——, "DexterJS: robust testing platform for DOM-based XSS vulnerabilities," in *Proc. ESEC/FSE*, 2015, pp. 946–949.

[30] N. Patnaik and S. Sahoo, "JavaScript static security analysis made easy with JSPrime," *Blackhat USA*, 2013.

[31] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proc. PLDI*, 2010, pp. 1–12.

[32] P. W. Security, "Burp Suite," https://portswigger.net/burp.

[33] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in HTML5 websites," in *Proc. NDSS*, 2013.

[34] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *Proc. USENIX Security*, 2014, pp. 655–670.

[35] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: exploring client-side cross-site scripting," in *Proc. CCS*, 2015, pp. 1419–1430.

[36] L. Suto, "Analyzing the accuracy and time costs of web application security scanners," 2010. [Online]. Available: https://www.beyondtrust.com/wp-content/uploads/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf

[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proc. NDSS*, 2007.

[38] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy," in *Proc. CCS*, 2016, pp. 1376–1387.

[39] X. Zheng, J. Jiang, J. Liang, and H.-X. Duan, "Cookies lack integrity: Real-world implications," in *Proc. USENIX Security*, 2015, pp. 707–721.

VIII. Appendix: List of Sink Functions

- `document.write`, and `document.writeln`
- Assignment to the `src` attribute of a script, embed, iframe, or img. Includes JavaScript assignment (`element.src = "..."`), and assignment using `setAttribute`.
- Assignment to the `href` attribute of a anchor element. Includes JavaScript assignment and `setAttribute`.
- `eval`
- Assignment to the inner text of a `script` node.
- Implicit string-to-function conversion inside `setTimeout` and `setInterval`
- Assignment to `innerHTML`, and `outerHTML`, and `insertAdjacentHTML` properties
- Assignment to `document.cookie`
- Assignment to `document.location`
- Assignment to the `style` attribute. Includes JavaScript assignment and `setAttribute`.
- Assignment to all event handler attributes. Includes JavaScript assignment and `setAttribute`.