Auto-Patching DOM-based XSS At Scale

Inian Parameshwaran Enrico Budianto Shweta Shinde
Hung Dang Atul Sadhu Prateek Saxena
National University of Singapore, Singapore
{inian, enricob, shweta24, hungdang, atulsadh, prateeks}@comp.nus.edu.sg

ABSTRACT

DOM-based cross-site scripting (XSS) is a client-side code injection vulnerability that results from unsafe dynamic code generation in JavaScript applications, and has few known practical defenses. We study dynamic code evaluation practices on nearly a quarter million URLs crawled starting from the the Alexa Top 1000 websites. Of $777,082$ cases of dynamic HTML/JS code generation we observe, 13.3% use unsafe string interpolation for dynamic code generation — a well-known dangerous coding practice. To remedy this, we propose a technique to generate secure patches that replace unsafe string interpolation with safer code that utilizes programmatic DOM construction techniques. Our system transparently auto-patches the vulnerable site while incurring only $5.2 - 8.07\%$ overhead. The patching mechanism requires no access to server-side code or modification to browsers, and thus is practical as a turnkey defense.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Web Security, Taint Analysis, DOM-based XSS, Auto-patching

1. INTRODUCTION

JavaScript has become a de-facto scripting language that powers popular web applications, browser extensions (or add-ons), HTML5 mobile applications (e.g., WebView, Windows 8 Metro apps), and server-side applications (e.g., NodeJS apps). Recent experimental systems have demonstrated that JavaScript can be used to develop traditional games, browsers and even an operating system [2, 6, 16, 17]. However, presently, applications built with JavaScript are fraught with security holes — for example, code injection in JavaScript applications (such as DOM-based XSS) is known to be highly pervasive and an elusive category of vulnerabilities for many commercial scanners to find [54]. A majority of popular web sites including Google+, Twitter and Yahoo have recently been vulnerable to these attacks [3, 4, 8].

DOM-based XSS has few known practical defenses. Conceptually, several defense-in-depth techniques are known to thwart DOM-based XSS; however, their practical deployment requires intrusive changes to the application and do not retain compatibility with older browsers. For example, Content-Security Policy (CSP) [55] and capability-controlled DOM [39] are two techniques that can ensure that JavaScript code only from whitelisted sources can be evaluated in a website. However, these techniques require significant rewriting of applications to be made practical, and as result, less than 1% of the sites have deployed these protections [33, 34, 60]. Further, these techniques work only on browser backends that support CSP or ECMAScript 6 features. Defenses which rely on browser-specific implementations or new language features fail to protect applications on existing browser and HTML5 backends. At the same time, the number of browser backends are proliferating — there are over 16 mobile browser vendors on the official Android market, each with over a million users [15].

The root cause of DOM-based XSS vulnerabilities is *unsafe* software engineering or coding practices — specifically, the use of unsafe string interpolation in dynamic code evaluation constructs. A common defense developers employ is to validate or sanitize untrusted data before using it in dynamic code evaluation constructs. However, this mechanism is well-known to be prone to errors in designing sanitization routines [24, 40] and mediating on all uses of dynamic code evaluation constructs consistently [23, 29, 52, 53].

In this paper, we propose a new approach to protecting JavaScript applications based on auto-patching. Our approach aims to retrofit protection to existing JavaScript applications, requiring only standard browser features available in existing browser implementations. Most previous prevention techniques have targeted browser-based script-blocking [20, 55, 56] or application of sanitization routines in client-side code [24, 52, 53], rather than rectifying the code to perform safe code evaluation. Our proposed approach is to automatically synthesize secure code patches which replaces unsafe application logic responsible for dynamic code evaluation. The generated patches generate dynamic code without invoking the HTML parser with untrusted strings in the browser. Instead, our patches directly construct the rendered page by using alternative DOM construction API — a well-known robust programming practice which eliminates the need for sanitization [45].

We develop a complete system called DEXTERJS for automatically synthesizing patches for DOM-based XSS vulnerabilities in JavaScript applications. DEXTERJS performs dynamic analysis to detect and repair DOM-based XSS bugs in real web applications. Our automatically synthesized patches are directly deployed on the website via a hot-patching mechanism we develop, offering a quick

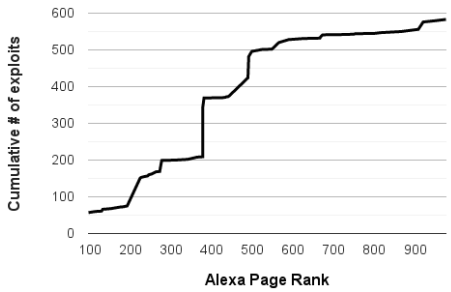


Figure 1: A cumulative distribution function (CDF) of zero-day exploits found in Alexa Top 1000 sites.

defense that requires no developer effort. Our patches are browser-agnostic, require no browser or server-side code modifications, and do not require users to install any plug-ins or add-ons.

Results. We have released our core analysis engine (DEXTERJS) as a cloud-hosted proxy for detecting DOM-XSS [7]. In our evaluation, we have used DEXTERJS on nearly a quarter million URLs crawled starting from the Alexa Top 1000 websites. First, we find 820 DOM-based XSS vulnerabilities are exploitable — a finding that is consistent with similar reports [54, 57]. DEXTERJS generates exploit-exhibiting witnesses for these exploitable cases automatically. Second, we find that of the $777,082$ dynamically generated code fragments observed during our evaluation, $103,335$ (13.3%) of the cases utilize unsafe string interpolation. This suggests that disabling all dynamic code evaluation is likely impractical and perhaps an overkill, since developers often use safe programming practices in dynamic code evaluation. Finally, our system successfully generates safe patches for all the vulnerabilities it confirms to be exploitable. The auto-patched websites show an average performance overhead of $5.2 - 8.07\%$, which is not significantly perceivable for interactive use. This enables auto-patching as a turnkey defense for existing sites.

Contributions. In summary, we make the following contributions:

- **Automated Patching:** We propose a backwards-compatible auto-patching defense for DOM-based XSS. Our approach requires no modification of browsers or server-side code, and requires no developer effort.
- **Browser-agnostic Fine-grained Tainting:** DEXTERJS comprises of a browser-agnostic JavaScript rewriting engine performing character precise fine-grained taint analysis robustly. It is tested on several popular web browsers, including Firefox, Chrome, Internet Explorer and Safari. We have released it for public testing [7].
- **New Findings:** We carry out an extensive evaluation of the Alexa Top 1000 sites. We crawl links from the top 1000 Alexa websites and analyzed JavaScript code from $228,541$ URLs, resulting in over $777,082$ distinct dynamic code evaluation instances constructed from untrusted data. First, we find that approximately 5% of these instances arise only on certain browsers owing to subtle browser variations, highlighting the advantages of browser-agnostic detection and defenses. Second, we find that developers use unsafe coding practices when generating dynamic web content in 13.3% of all the cases. Third, a total of 820 distinct zero-day DOM-based XSS vulnerabilities across 89 different domains were confirmed to be exploitable. 583 of these exploits were from the Alexa top 1000 domains (Figure 1), the remaining were from external iframes embedded in these websites. Finally, all the found vulnerabilities are auto-patched with a average performance overhead of $5.2 - 8.07\%$.

2. PROBLEM OVERVIEW

As opposed to traditional (reflected and persistent) XSS, DOM-based XSS executes purely at the client-side, rendering server-side sanitization insufficient. We give an example to motivate the specific problem we tackle in this work.

2.1 Motivating Example

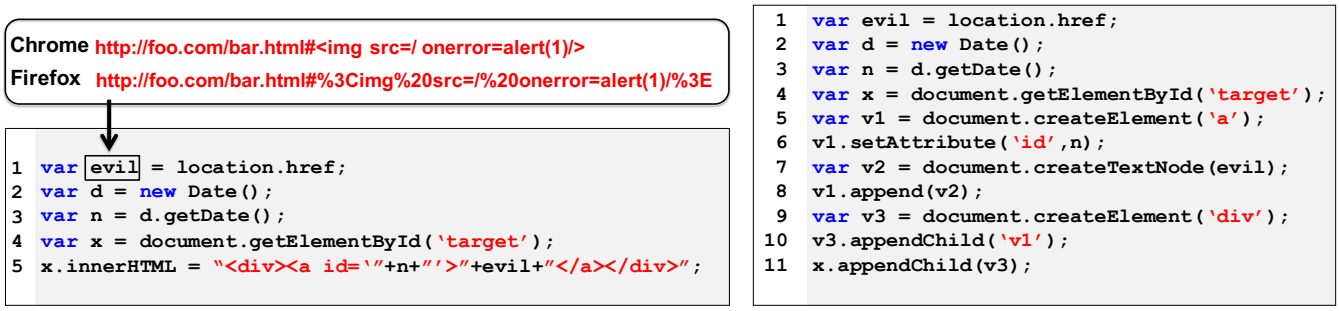
DOM-based XSS is a code injection vulnerability in which a web attacker is able to inject malicious JavaScript in a client's web session. In the attack, we assume that the attacker can entice an unsuspecting user into visiting a URL of his choice. The attacker embeds the attack payload either directly into the maliciously crafted URL pointing to the victim website, or may load the victim's web page in an `iframe` and subsequently pass data to it (say via the `postMessage` API) [48]. Using these mechanisms, the attacker can control some data which the victim site consumes when executed on the client's browser.

A simple example of a vulnerable JavaScript application is shown in Figure 2(a). In the attack, the user clicks an attacker-provided URL, which he can craft targeting the victim's web browser (e.g., `http://foo.com/bar.html#<img src=/ onerror=alert(1)/>`). The vulnerable JavaScript code runs in the domain `foo.com`. The code snippet programmatically reads the location of the webpage (using `location.href`). It constructs a string with this information and uses the browser's dynamic HTML generation constructs like `innerHTML` to render the string as HTML code in the client's browser.

The attack succeeds because the JavaScript code renders attacker-controlled strings — an `img` tag embedded in the URL in this case — as HTML content. The attacker's payload automatically triggers a JavaScript event, allowing the attacker to run arbitrary code on behalf of the origin `foo.com`. As a preventive measure against XSS attacks, several browsers deploy client-side XSS filters [20]. These filters are quite effective against its server-side counterpart (reflected XSS) [26]. However, DOM-based XSS attack vectors can easily bypass these browser-based XSS filters [56]. Safe sanitization of untrusted string embedded in HTML content is a known difficult problem, prone to errors [24, 40, 43, 53].

Safer coding practices. A cleaner and more effective approach is to avoid using code evaluation constructs such as `innerHTML` on unsafe data, as a safer coding practice. For example, the logic in Figure 2(a) can be alternatively programmed with safer code shown in Figure 2(b), which preserves the intended functionality of the original code. The vulnerable code "interpolates" untrusted string values into a fixed string, and then evaluates the interpolated string as HTML. Instead, the safe code renders the same visual DOM using safer DOM construction API without invoking the HTML parser. As discussed in previous work (c.f. Blueprint [45]), this approach preserves the intended structure of the dynamically generated HTML and cleanly separates the code from attacker-controlled data. Other code evaluation constructs that evaluate strings (e.g., `eval`, event attributes) can similarly be replaced with programmatic constructs that avoid string interpolation (e.g., using `JSON.parse` instead) [41, 51].

In our large-scale evaluation, we find that developers often make use of programmatic DOM manipulation (e.g., attaching attributes or extending DOM hierarchies) rather than unsafe HTML evaluation on attacker controlled data (e.g., using `innerHTML`). Direct code evaluation of strings interpolated from attacker controlled data, such as in our vulnerable example, accounts for less than 14% of the cases. Our defense goal is to identify and convert such vulnerable constructs to safe-code by patching the application.



(a) (b)

Figure 2: (a) Example of string interpolation followed by code evaluation (unsafe coding practice). (b) Example of code which uses programmatic DOM constructs and avoids DOM-based XSS attack (safe coding practice).

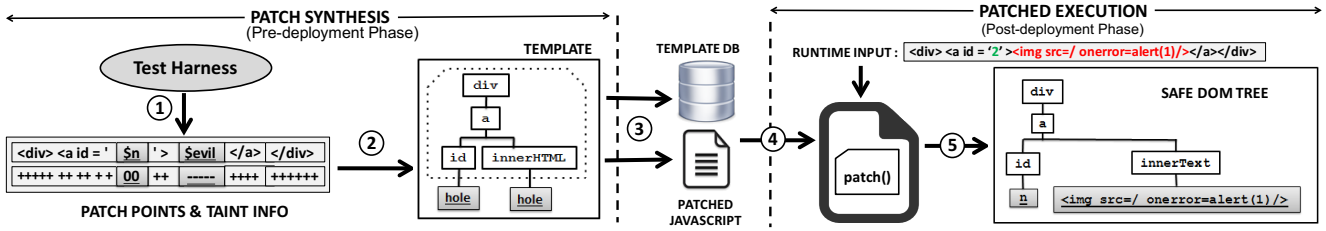


Figure 3: Approach overview of DEXTERJS: (1) Patch point identification (2) Template Inference (3) Hot-patching (4) Template selection (5) Runtime template instantiation for safe DOM rendering.

2.2 Problem Statement

We refer to all the points in the program where attacker controlled data is used in a code evaluation construct as *dynamic code evaluation points (DCE points)*. Only a fraction of these are actually vulnerable to DOM-based XSS and are candidates for auto-patching [1]. This subset is called as *patch-points*. By default, DEXTERJS generates patches for patch-points. Developers can configure DEXTERJS to generate patches for other unconfirmed DCE points as well.

Goal: Auto-Patching Vulnerable Applications. DEXTERJS takes as input a website which may be vulnerable to DOM-based XSS, a benign test harness, and a list of browser backends on which the website renders correctly[2]. As output, DEXTERJS generates a list of confirmed DOM-based XSS exploits and the "auto-patched" JavaScript application that invokes the patch (See Figure 3).

The code patch applied by DEXTERJS to safeguard vulnerable web applications should satisfy the following properties.

- *Correctness.* The patch should render a valid DOM tree, preserving the structure of the DOM created by the original operation at the patch-points, for inputs observed during the execution of the test harness. This property specifies a necessary condition to ensure that the patch does not break the original functionality of the application.
- *Safety.* The patch should never execute script code from data under the direct control of the attacker, for all inputs reaching the patch-points.
- *Browser-agnostic.* The patch added by DEXTERJS should not use any non-standard (or browser-specific) language features that are unsupported by the given set of browsers.
- *Low Overhead.* The patched applications should have a minimal performance overhead.

Overview & Deployment Setting. DEXTERJS does not require any user installation on the client's end or access to the server-side code. Website developers and testing teams can use DEXTERJS service via a proxy to analyze and auto-patch their applications. Therefore, the instrumented code can be easily sent out from proxy server to their web browser or analyzed by a cloud-based browser backend. DEXTERJS aims to achieve scalability and compatibility with existing web applications and platforms.

3. DEXTERJS APPROACH

DEXTERJS system comprises of two phases shown in Figure 3: a pre-deployment analysis phase and patched execution in the deployment phase. In pre-deployment analysis, DEXTERJS executes the application under the given test harness. During this process, it observes if the application uses any string interpolation for code generation. For each such instance, DEXTERJS infers a set of benign DOM (or parse) trees that the application intends to generate by evaluating the interpolated string. In essence, these benign parse trees define a fixed *template* which can be "instantiated" at runtime with attacker-controlled values safely[3]. In our pre-deployment testing, we create a database of such whitelisted or allowed templates and instrument the application to instantiate the templates securely at runtime. In deployment, the instrumented patches inserted by DEXTERJS select the appropriate template from the whitelisted database, and instantiates it using safe programmatic DOM construction API available in all commodity web browsers.

[1]Rest of them may not be exploitable by the virtue of proper sanitization defense in place.

[2]The test harness can either be a set of functional test cases (Selenium Scripts), or a URL test suite. Alternatively, DEXTERJS can automatically crawl to generate the test harness.

[3]Our templated execution is analogous to how prepared statements in SQL help avoid SQL injection.

3.1 Patch Synthesis

During the pre-deployment phase, DEXTERJS first instruments and analyzes the given JavaScript application (See Figure 3). This includes extracting the position of all DCE points, verifying if they are exploitable and marking the exploitable DCE points as patch-points. DEXTERJS utilizes character-precise dynamic taint-tracking to infer DCE points. Further, DEXTERJS infers the template structure using the taint information identifying the attacker-controlled bytes in the string used for generating client-side code.

Table 1: Default sources and sinks used by DEXTERJS.

	Sources	Sinks
HTML	-	innerHTML, outerHTML, Script.src, Image.src, Script.text, Script.textContent, Script.innerText, write, writeln
JavaScript	-	eval, setTimeout, setInterval
Location	URL, baseURI, documentURI, window.location, document.location, location.hash, location.search, location.href	window.location, document.location, location.href, location.assign, location.replace
Storage	cookie, localStorage, sessionStorage	cookie, localStorage, sessionStorage
Postmessage	Postmessage data	window.postmessage

3.1.1 Identifying patch-points & Vulnerabilities

Dynamic Taint Analysis. We use dynamic taint analysis to detect all flows from unsafe input sources (i.e., can be controlled by attacker) to code execution sinks [48]. For this, our DEXTERJS server instruments client-side application code and runs the instrumented code in multiple browsers. During this benign-run of the application, the character-precise taint-tracking records all the flows that use the tainted portions of the string in code execution constructs. The code execution sinks for these flows are then marked as DCE points. For example, the instrumented code must report the taint flow from the taint source `location.href` (line 1) to the taint sink `x.innerHTML` in Figure 2(a). DEXTERJS uses the sources and sinks defined in Table 1 by default.

We build a source-to-source rewriting infrastructure for embedding fine-grained dynamic taint-tracking logic in the original application. The challenges in implementing such a system that scales and is robust for real-world applications are discussed separately in Section 4. DEXTERJS's taint analysis engine instruments application code to perform both positive and negative tainting [37, 38]. By definition, positive tainting identifies and tracks trusted data originating from constants. On the other hand, negative tainting focuses on untrusted data derived from attacker-controlled inputs. Tracking both positive and negative taint helps us determine the set of values which are neither positively nor negatively tainted. For example, values which are derived from sources such as `Date`, `Math.random`, etc. are non-deterministic but benign. The attacker controls only parts of the string which are negatively tainted. This information is later used for auto-generating patches.

Verifying Exploitability. DEXTERJS verifies whether the DCE points are actually exploitable and if so, marks them as patch-points. It generates browser-specific exploits based on the exact context wherein the tainted data is being interpreted [48, 54]. Our attack rules are based on publicly known strategies such as the XSS filter evasion cheat sheet [21, 54]. Finally, DEXTERJS executes all the generated candidate attack vectors in commodity browsers and verifies their exploitability. All the confirmed exploitable flows are then grouped as patch-points and are given to the next phase of the DEXTERJS system for patching (See Figure 3 Step 1).

3.1.2 Template Inference

DEXTERJS treats each patch-point separately and gives it a unique identifier ID. During a benign run in the pre-deployment phase, DEXTERJS's synthesis engine records every runtime string S that has been passed to a particular patch-point p. Apart from the string, the engine also receives a corresponding *taint type* T for each character that forms S. Recall that the taint type of each character can either be positive $(+)$, negative $(-)$, or non-deterministic (0).

To determine a benign parse tree of S, DEXTERJS relies on commodity browser engine to programmatically convert S into a DOM tree object. This is done via an API `parseFromString()` which is available in all mainstream browsers. Using the resulting DOM tree object of S, along with the taint information T, the synthesis engine then determines nodes of the DOM tree which are negatively tainted (under the attacker's control) and nodes which are positively or non-deterministically tainted. This DOM tree information – along with the augmented taint information for each node – is referred to as a *template*. The template is a model of the benign structure of code generated at p. Using a template, DEXTERJS can determine all the static parts of the tree (i.e., positively-tainted nodes) and then use it to fix the expected structure. On the other hand, all the nodes in the tree that are either negatively or non-deterministically tainted are considered as *holes* which are intended to be filled based on the runtime string at p. Finally, DEXTERJS collects all the templates generated at each patch-point into a *white-listed template database*.

Example. We illustrate DEXTERJS's template-inference process with the help of the running example discussed in Section 2. In Figure 2(a), `innerHTML` (the sink in this case) is used to add a new anchor tag inside a `div` element. DEXTERJS receives a tainted string S along with its associated taint range T. Next, the synthesis engine parses S into its corresponding DOM tree. Along with this DOM tree and the taint information T obtained from Step 1, it generates a *template* (See Figure 3 Step 2). The synthesis engine uses this template to infer that the `<div>` element is a static structure, whereas the values of `id` and `text` of `<a>` element are holes.

3.1.3 Hot-Patching

During the pre-deployment phase, DEXTERJS also adds hooks at the patch-points such that the program will execute the `patch` function instead of the original code evaluation construct. This is akin to the notion of *hot-patching* during software updates [28]. The `patch` function implementation is added to the beginning of the instrumented program in global scope, ensuring its visibility throughout the program. Figure 3 Step 3 illustrates the redirection of control flow through hot-patching from `innerHTML` to `patch` function.

3.2 Patched Execution

All the heavy-weight analysis for identifying the patch-points is done before deployment. Once the application is deployed on the server, only instantiating the correct patch is delegated to runtime, thus minimizing the performance overhead at runtime.

3.2.1 Runtime Template Instantiation

The `patch` function takes in two parameters: (1) the static identifier ID of the patch-point and (2) a runtime string S' which is passed to the DCE point. DEXTERJS inserts the database of white-listed templates obtained during the template inference step along with the `patch` function into the vulnerable page before deployment. The `patch` function queries this database based on the ID to find the right template to be applied to the runtime string S'.

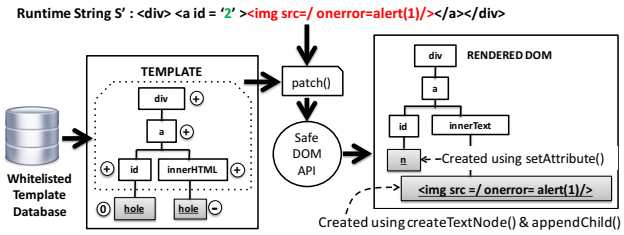


Figure 4: Template Instantiation, showing how the template holes are populated from the input and rendered using safe DOM construction API.

Safe DOM Rendering. The goal of the `patch` function is to render S' such that the DOM-tree generated in the browser is free from XSS payloads, and contains valid benign markup. To do this, we use the BUILD_SAFE_DOM procedure shown in Algorithm 1 line 4. The first preparatory step is to parse S' into a preliminary DOM-structure, which is not rendered directly as it is derived from the untrusted string S' (line 3). Then, we match this structure with a template from the whitelisted template database (variable *tmpl* at line 2). The matching procedure starts from the root comparing each node of the preliminary-DOM with the template recursively (line 4-27). If the template and the preliminary-DOM match at all nodes, except at the holes, then the algorithm has found runtime values to fill in the holes. This is illustrated in Figure 4.

The final step is to generate code and render the entire DOM tree securely (See Figure 3 Step 5). The fixed structure of the template is constructed directly from the whitelisted template, not from the preliminary-DOM, using standard DOM construction APIs. The values that match the holes in the preliminary-DOM derived from S' are rendered via *safe* DOM construction API that do not interpret strings as code. Specifically, these APIs are `setAttribute()` and `createTextNode()` (underlined at line 15, 32, 34) which prevent script execution [45]. The entire approach only renders whitelisted nodes completely independent of runtime values in S', and uses non code-evaluating constructs to append strings from S' to the rendered tree. Thus, this is a principled approach to separate code from data. As a result, all untrusted strings are prevented from reaching code-evaluating DOM API and achieve the equivalent of executing the safe code shown in Figure 2 (b).

If the template and the preliminary-DOM do not match at all nodes (except for the holes), this is a template which has not been encountered in our testing. In such cases, we have two options: either to proceed with the rendering (which may be vulnerable) or to avoid rendering the string (preserving compatibility). We leave this as a configurable option to the end system user; our default presently is to be safe and not render the string.

There are possible cases where just using safe DOM construction techniques are not enough. This can happen if attacker controlled data is used inside a `script` tag, event attributes, or in the `src` attribute of a `script` or `iframe` tag. However, such cases were extremely rare in our evaluation of 228,541 URLs and were often non-exploitable because of existing sanitization. Nonetheless, our algorithm uses a whitelisted policy for the resource-name for the rendered string, similar to what has been proposed in the previous work [45, 56]. This is indicated as the *context_valid* check in Algorithm 1 to protect against such cases.

3.2.2 Runtime Template Selection

There can be multiple templates for a given patch-point if multiple control flow paths reach the same patch-point (for example, if there is an `if-else` branch before the patch-point). In this case DEXTERJS should determine which one of these templates is to

Algorithm 1 Patching Technique using Safe DOM Construction

```
1:  procedure PATCH(ID, S')
2:    tmpl ← get_template(ID)
3:    BUILD_SAFE_DOM(get_rootnode(tmpl), get_rootnode(parse(S')))
4:  procedure BUILD_SAFE_DOM(N_T, N_R)
5:    if N_R is root and N_T.tagName ≠ N_R.tagName then
6:      issue_warning()
7:    else if N_T.tagName = N_R.tagName then
8:      for attribute a in list_attr(N_R) do
9:        if a ∉ list_attr(N_T) then
10:         RESTRUCTURE(N_T, html_stringify(a))
11:       else
12:         N_T.setAttribute(a.name, context_valid(a.value))
13:     if N_T.children.length = N_R.children.length then
14:       for C ∈ list_child(N_R), C' ∈ list_child(N_T) do
15:         BUILD_SAFE_DOM(C', C)
16:     else
17:       isNewTmpl ← true
18:       for C ∈ list_child(N_R) do
19:         if is_unique(C, N_R.children) then
20:           C' ← get_associated_node(C, N_T.children)
21:           if C' is ∅ then
22:             RESTRUCTURE(N_T, html_stringify(C))
23:           else
24:             isNewTmpl ← false
25:             BUILD_SAFE_DOM(C', C)
26:       if isNewTmpl is true then
27:         issue_warning()
28: procedure RESTRUCTURE(N, elm_str)
29:   hole ← get_hole(N)
30:   if hole is an attribute then
31:     sVal = context_valid(hole.value + elm_str)
32:     N.setAttribute(hole.name, sVal)
33:   else if hole is a child node then
34:     text_node = doc.createTextNode(context_valid(em_str))
35:     N.appendChild(text_node)
```

be imposed on the runtime string (See Figure 3 Step 4). Previous works also face this challenge in the context of server-side code and there are several ways to handle this. ScriptGard [53] uses the code path and the calling context in the program to determine the appropriate choice of structure. In our work, we resolve ambiguity in contexts using the DOM position — the identity of the parent DOM node to which the newly-generated DOM sub-tree will be attached. If the DOM position is the same for two templates at a patch-point, DEXTERJS can record information about the path context along with the template. This is obtained by path profiling during pre-deployment testing. In this case, we can use the path context at runtime to resolve the ambiguity as done in ScriptGard [53]. Dynamic path context aggregation requires additional profiling, but is only necessary in an extremely small number of cases where the DOM position alone is not sufficient to resolve the ambiguity. A richer and more expressive template language could be used instead of a whitelisted template database to account for these variations. However, in our evaluation, we have not seen this selection ambiguity given the relatively small number of patch-points per web application, i.e., we always have a unique patch to apply at each patch-point based on the DOM position.

4. JS ANALYSIS ENGINE

DEXTERJS system analyzes all the JavaScript code dynamically to determine if the application is unsafe, specifically if any attacker-controlled value is used as code to generate HTML. We design our own information flow analysis for tracking attacker-controlled data.

4.1 Technical Challenges

Several previous works propose both static and dynamic analysis

of JavaScript [44, 54, 56]. However, several challenges arise when achieving the same goals in the context of auto-patching problem.

- *Browser independent JavaScript analysis.* For compatibility and effectiveness, the analysis and patching logic should not be integrated in a specific browser or a JavaScript engine. An alternative is to leverage the language's object level abstraction which makes the approach more generic [44].
- *Complete mediation on executing JavaScript.* JavaScript supports dynamic code generation via `eval`, reflection, etc. To hot-patch all the DOM-based XSS vulnerabilities in a given web application, it is essential to instrument code generated through these constructs on-the-fly.
- *Robust handling of complete JavaScript features.* JavaScript has many complex built-in constructs, native functions, and primitive operators. It is crucial to handle these language semantics in our instrumentation to preserve the original functionality of the application.

4.2 Design: Taint-Tracking Engine

Source Rewriting. One of the main challenges in carrying out browser-independent program analysis is that browsers vary a lot, making such an analysis cumbersome if it is not done through source code rewriting. Building such a fine-grained transformation system that can scale up is non-trivial. In fact, recent tools alternatively resort to either record-replay or combine static and dynamic techniques to tackle the challenges posed by features and subtleties of JavaScript [44, 57, 58]. We explain how DEXTERJS achieves robustness and a clean design in its instrumentation for inlining taint-tracking logic, some of which build upon the ideas of object wrapping previously proposed [44] but incorporate many additional new strategies to maintain complete compatibility with real JavaScript code.

DEXTERJS performs source-to-source rewriting for character-precise taint-tracking for strings. It instruments JavaScript on the fly via a proxy server as the website is executed under the given test harness. The taint propagation logic and metadata is kept within the website's execution context in the browser. Such heavy-weight instrumentation is quite intrusive and we explain two key choices DEXTERJS makes regarding where to keep taint metadata and how to instrument each string operation in the JavaScript application. The subtleties of preserving application correctness in the presence of such instrumentation is discussed in Section 4.3.

4.2.1 Attaching Taint Metadata

Dynamic taint analysis needs to store the taint information for primarily strings in the program to detect XSS flows [48]. In this section, we discuss various approaches to store taint metadata which do not scale and how our solution of object wrapping is both scalable and plays well with JavaScript semantics.

Storage in a separate namespace. Traditional dynamic taint analysis for languages such as C store the taint metadata in a separate namespace (e.g., shadow memory). This is not a viable solution, since JavaScript cannot access the address of any variable.

Storage in a global namespace. Taint metadata can be stored in the same namespace of the program using alpha renaming of metadata variables to resolve namespace conflicts. However, since scoping rules are inconsistent across different browser implementations (as we show in our evaluation), reasoning about the namespace of a variable is not straightforward. Further, storing the taint metadata in the global namespace quickly blows up the program's memory usage. Using a fixed set of variables (analogous to fixed set of registers available in the CPUs) to store the taint metadata leads to overly complex *spilling* management in the instrumentation [22]. We adopt a more efficient solution that works generically for global and function-scoped data.

Per-String Storage. In all the above strategies, the taint information of each variable is stored in separate variables. Storing the taint information as a property of the string would preclude costly lookups in taint maps. This is an elegant solution which solves scoping as well as performance challenges. We adopt this approach with one caveat — attaching properties to primitive data types like string literals fails silently in JavaScript (See Listing 1).

Listing 1: Infeasibility of attaching taint with string literals.

```
1 var a = "foo"; // a is a string literal
2 a.taint = true;
3 console.log(a.taint); // prints undefined
```

Listing 2: Conversion to Objects to track taint information.

```
1 var a = new String("foo"); // a is string object
2 a.taint = true;
3 console.log(a.taint); // prints true
```

Although, JavaScript does not allow adding properties to string literals, objects have no such limitation. Therefore, a viable strategy is to box the string literals to string objects and then attach the taint property to them [4]. Further this mechanism implicitly propagates the taint in cases of assignments and function calls.

Hence, we convert all the string literal to string objects and then use them to store the taint information. Note that there are subtle differences in the semantics of String objects and String literals, as discussed in Section 4.3.

4.2.2 Inlining Propagator Logic

The exact point of insertion of the taint propagation statements in the instrumented code is also a crucial design choice. Naïve approaches like inserting such statements after every statement of the original program leads to re-execution of code in our taint propagation statements, breaking the semantics of the program. Our taint propagation logic has to be embedded at a more fine-grained level within the original program.

To solve this challenge, DEXTERJS uses a standard feature of JavaScript called immediately invoked function expression (or IIFE). This feature allows a function to be defined and immediately invoked within a single block-scope construct, and has consistent semantics in all browser implementations. An example of our instrumentation for the code statement in Figure 5(a) is shown in Figure 5(b).

```
1 var a = b() + "foo";
```

(a)

```
1 var a = (function() {
2 var rhs = b();return rhs + "foo"
3 })();
```

(b)

Figure 5: (a) Un-instrumented input program. (b) IIFE used to group taint analysis statements.

Our mechanism of using IIFE has several advantages in achieving memory efficiency and nesting of instrumentation. First, it uses function scoping, so temporary variables used in the instrumentation (`rhs`) are scoped within the IIFE and can be garbage collected immediately after the IIFE finishes executing. Second, since IIFE is a function closure, it has access to all the variables declared outside the IIFE. For example, `b` can be used within the IIFE even though it wasn't declared within it. This preserves the original scope of the un-instrumented statement. Third, IIFE encapsulation also extends naturally to nesting of IIFE scopes that is necessary to compute taint for multiple sub-expressions of a larger expression.

[4] Our definition is similar to the concept of *Boxing* in Java.

4.3 Completeness in Handling JavaScript

DEXTERJS handles a variety of practical cases for JavaScript in the wild. We discuss the interesting challenges here and our solution to handle these.

4.3.1 Dynamic Evaluation

JavaScript supports various dynamic code generation constructs, like `eval` and the `Function` constructor, as well as constructs for fetching data from external sources dynamically. At times, the arguments to these functions are only known during runtime and hence can be instrumented only then. To this end, the arguments to these functions are sent back to the DEXTERJS's instrumentation server using `XMLHttpRequest`. This returns the instrumented version of the code which is then passed to the respective function. The input program using `eval` and a naïve instrumented program of the same is shown below, with some details elided for brevity.

Listing 3: Sample code using `eval`

```
var code = "var a = 2;";
eval(code); // The value of a is 2 here
```

Listing 4: Naïve instrumentation strategy for `eval`

```
var code = "var a = 2;";
(function(){(
  var _$temp1 = (code);
  _$temp2 = new XMLHttpRequest();
  _$temp2.open('POST', instrumentationServerIP,false);
  _$temp2.send(_$temp1);
  // _$temp2.responseText contains instrumented code
  return eval(_$temp2.responseText);
})(); // a is not defined here
```

Recall that DEXTERJS wraps `eval()` using an IIFE as shown in Listing 4. However, `eval()` function poses an interesting case: any variable that is created using `eval()` will bound to the scope where `eval()` is called. Therefore, the variable `a` would only be defined within the IIFE and accessing `a` outside it would result in a reference error, thereby breaking the original program semantics. For such cases (e.g., instrumenting `eval`), where IIFE instrumentation results in incorrectness, we instead use a *sequence expression* to group statements without creating a function scope. Sequence expressions (syntactically `(expr,..., expr)`) combines multiple expressions into a single expression [5]. The final correct instrumentation logic for Listing 4 is shown in Listing 5.

Listing 5: `Sequence Expressions` to instrument `eval()`

```
var code = "var a = 2;";
( _$temp1 = (code),_$temp2 = new XMLHttpRequest(),
  _$temp2.open('POST',instrumentationServerIP,false),
  _$temp2.send(_$temp1),
  // _$temp2.responseText contains instrumented code
  _$temp3 = eval(_$temp2.responseText),_$temp3);
// The value of a is 2 here
```

4.3.2 Reflection

JavaScript has constructs for reflection, permitting application code to self-inspect and manipulate program code during runtime. For example, it can be used for listing all properties of an object, or searching for a particular string inside a function definition. As an example, in Listing 6, if the function `foo` is transformed into another function called `wrap_foo` by the instrumentation, `foo.caller` will return `wrap_foo` instead of `bar`.

Listing 6: Function `foo()` accesses its caller's object through `caller` property. `temp` is the string representation of `bar`

```
function foo() { var temp = foo.caller.toString(); }
function bar() { foo(); }
bar();
```

To handle reflection, DEXTERJS hooks reflection constructs and returns values that match the original application semantics. Reflection is used quite commonly on popular websites; we discuss two common idioms and how DEXTERJS handles such cases. As a first example, `Function.caller` property returns the function that invoked the specified function. Since we use IIFE to aid in the instrumentation process, this changes the value of the `caller` property and may lead to a change in the intended program behavior. We ensure that we return the correct value of this property by recursively calling it till we find an un-instrumented function. As a second example, consider a snippet of code inspecting the `toString` property of a function. The `toString` property of a function returns a String representation of the function's declaration as shown in Listing 7. In this example, `a.toString()` returns the String representation of the instrumented function whereas the expected behavior would be to return the String representation of the un-instrumented function. Since there is no simple way to convert the instrumented code snippet back to its original version, the results of each instrumentation is stored on the server. The `toString` function of each function is then overridden to request the un-instrumented version from the server, which is then used in the program. DEXTERJS handles these and many such real-world cases in a similar fashion.

Listing 7: Program using `toString` property of a function

```
function a() { foo = bar; }
a.toString(); // "function a(){ foo = bar; }"
```

4.3.3 Built-in constructs and Native Functions

Special JavaScript constructs such as value of `this`, `for-in`, `arguments` and re-definitions of native functions must be preserved even in the instrumented version of the JavaScript program. We discuss how DEXTERJS wraps references to these constructs to preserve semantics.

The `this` variable. The value of `this` variable in a function depends on how the function is called. Consider the example we discuss in Listing 8. A naïve implementation as shown in Listing 9 can lead to incorrect behavior since the value of `this` will be the global window object. This problem is fixed by using the `apply` function. Using the `apply` function the IIFE can be passed the correct value of the `this` variable as shown in Listing 10.

Listing 8: Sample program showing usage of `this`.

```
var obj = {};
obj.foo = function() { return this; }
console.log(obj.foo()); //prints obj
```

Listing 9: Naïve instrumentation for Listing 8.

```
var obj = {};
obj.foo = function() {
   return (function() { return this; })(); }
console.log(obj.foo()); //prints window
```

Listing 10: Correct instrumentation for Listing 8.

```
var obj = {};
obj.foo = function() {
return (function() {return this;}).apply(this,[]);};
console.log(obj.foo()); //prints obj
```

The `arguments` variable. Using an IIFE changes the expected value of the `arguments` variable. This is solved by passing the correct value to the IIFE using the `apply` function, similar to the way the `this` variable is handled.

The `for-in` loop. The naïve method of attaching taint to a String object causes problems when the string object is enumerated using a for-in loop. To overcome this problem, the taint property is set to be non-enumerable using the `defineProperty` function.

Listing 11: Sample Program using `for-in loop`.

```
document.cookie = "ab";
var a = document.cookie;
for(foo in a) console.log(foo); // prints 0, 1
```

Listing 12: Naïve Instrumented for Listing 11.

```
1 document.cookie = "ab";
2 var a = (function() {
3    var rhs = new String(document.cookie);
4    rhs.taint = 1; //attaching taint
5    return rhs; })();
6 //uninstrumented version for simplicity
7 for(foo in a) console.log(foo);//prints 0, 1 and taint
```

Listing 13: Correct instrumentation for Listing 11.

```
1 var a = (function() {
2    var rhs = new String(document.cookie);
3    Object.defineProperty(rhs, 'taint',
4     {enumerable: false, value: 1}); //attaching taint
5    return rhs; })();
6 //uninstrumented version for simplicity
7 for(foo in a) console.log(foo); //prints 0, 1
```

Re-definition of Native Functions. JavaScript allows developers to override the behavior of native functions. For example, assume that the developer augments the `Function.prototype` object to modify the `bind` function, which DEXTERJS also uses internally as a part of the instrumentation logic. This can lead to an infinite recursion since the overridden body of the `bind` function would also have a call to `bind`. To avoid this recursive execution, DEXTERJS memoizes the `bind` function before any script is executed and uses this cached function in the rest of the instrumentation logic. This is applicable to all native JavaScript functions and DEXTERJS caches all the native functions it uses in its instrumentation logic before using them.

Handling type and comparison operators. All string literals in the program are converted to string objects in the instrumented program to store taint information. String literal and objects differ in subtle ways with respect to some operators as shown in Listing 14. To preserve the original semantics, we need to modify the behavior of operators such as the `typeof` operator. Since JavaScript does not support operator overriding, we modify the behavior of `typeof` by replacing it with a function `dexterTypeof`. Specifically, `dexterTypeof` function returns 'string' when an instrumented string object is passed to it and mirrors the behavior of `typeof` operator for rest of the cases. DEXTERJS attaches a custom property to all the instrumented string objects. This helps it to distinguish between the string objects created in the original program (e.g., `a`) vs. those created by instrumented code (e.g., `b`). In a similar way, DEXTERJS preserves the semantic behavior of other operators such as ==, ===, ! =, ! ==, the `instanceof` operator, and the implicit comparisons in the `switch-case` operator. Specifically, DEXTERJS converts the aforementioned operators to DEXTERJS function calls in the desugaring phase before instrumentation, as shown in Listing 15 corresponding to the Listing 14.

Listing 14: `typeof` behavior for String objects and literals.

```
1 var a = new String("foo");
2 var b = "b";
3 typeof a; // "object"
4 typeof b; // "string"
```

Listing 15: Modifying behavior of `typeof` operator.

```
1 var a = new String("foo");
2 var b = new String("b"); //converted to object by
      instrumentation
3 dexterTypeof(a); // "object"
4 dexterTypeof(b); // "string"
```

5. IMPLEMENTATION

We implement DEXTERJS as a proxy server based on node-http-proxy and mitmproxy adding 238 lines of code [1, 14]. We build DEXTERJS's instrumentation, taint analysis, exploit verifier, and auto-patching engine using the Node.js platform. Table 1 shows the list of all the sources and sinks supported by DEXTERJS. We implement our source-to-source rewriting logic with 1237 lines of code on top of Esprima [10] and Cheerio [11] node modules.

Table 3: # Exploits / #Flows for usnsafe HTML generating API

		SOURCE		
		Location. href	Location. hash	Location. search
SINKS	Document.write	451/67228	37/2160	204/1398
	Document.writeln	22/536	2/15	39/109
	HTMLElement.innerHTML	57/7890	0/23	8/401

Crawler. We utilize the Selenium framework [19] to implement a web crawler and a simple GUI fuzzing tool. Our Python based crawler is written in 808 lines of code and supports multiple OSes such as Linux, Windows and OS X. We have tested the crawler on mainstream browsers such as Chrome (v35), Firefox (v30), Internet Explorer (IE 10) and Safari (v7). Our crawler crawls the web application in a depth-first manner up-to a maximum depth of 5 and is capable of configuring the browser proxy, enforcing a page-load time-out, browsing in private-mode and controlling browser-specific features like the Chrome's XSS-auditor. It is also able to perform a wide range of interactions on websites, such as extracting data from DOM nodes, filling out forms based on the input type, triggering different events — which are necessary to dynamically analyze features in the instrumented web application [37, 47].

Attack Generator and Verifier. We implement an attack generator using Node.js platform in 1908 lines of code. We develop a custom parser based on the PEG.js node module to analyze the context of taint sink pattern [18]. We identify various attack patterns using the XSS filter evasion cheat sheet [21] and evaluate taint sink patterns against these to generate the actual exploit URLs.

Auto-patching. Benign templates for each page of the web application are stored on the server as a JSON file. These templates along with a 3KB JavaScript patch is then injected into all vulnerable pages. The patch function is responsible for protecting the patch-points and notifying the server of any attack attempts at runtime through a `XMLHttpRequest` call.

6. EVALUATION

In this section, we show the scalability and correctness of DEXTERJS by instrumenting popular real-world websites and testing it against benchmarks for DOM-based XSS. We focus on how often insecure coding practices are being used in the websites we analyzed. We then measure the performance of our auto-patching tool and show its efficacy in safeguarding web applications. Lastly we analyze the source of browser-specific flows showing the need for a tool which is able to capture flows across diverse browser backends. All the experiments were conducted on a Intel Xeon® 2.0 Ghz CPU with 64 GB RAM. We set a timeout of 3 minutes for testing each instrumented webpage.

6.1 Scalability & Correctness of DEXTERJS

In order to test the scalability of DEXTERJS, we tested the Alexa Top 1000 websites which were run on 4 mainstream browsers — Chrome, Firefox, Internet Explorer and Safari in June 2014.

Micro-Benchmarking. To test the accuracy of our taint analysis engine, we benchmarked DEXTERJS with the IBM JavaScript Security Test Suite [13] and the Firing Range testbed [12] containing 136 and 29 test cases respectively. DEXTERJS is able to detect taint flows in all these tests expected from a dynamic analysis tool. Exploits were automatically generated for all DOM-based XSS with URL sources with a zero false positive rate. DEXTERJS has a zero false negative rate with respect to these benchmarks.

Zero-day vulnerabilities in Alexa Top 1000 Websites. DEX-

Table 2: Summary of distinct flows detected by DEXTERJS by crawling URLs recursively starting from the Alexa Top 1000 websites. Each flow represents the use of data from a untrusted source (listed as columns) to a critical sink (rows).

		SOURCES											
		Web Storage	Location. search	Location	Cookie	Referrer	Window. top	Window. parent	Window. name	Location. href	Location. hash	post Message	**Total**
SINKS	Web Storage	4020	1764	340	4176	198	706	59	0	3168	25	106	14562
	Script.src	160	3548	13761	5721	29742	3407	319	440	15111	191898	105	264212
	Image.src	8204	5562	56497	117245	4817	6976	1396	981	74649	8421	1601	286349
	Anchor.href	0	681	811	26	3929	227	0	0	8987	316	48	15025
	Script.text	1	0	3	0	0	14	0	0	69	0	502	589
	Cookie	560	629	15150	59467	804	381	181	9443	2295	39	80	89029
	Location	0	106	296	7	18	22	10	0	114	0	0	573
	setTimeout	4	1	52	17	2	256	64	1	22	0	26	445
	eval	820	135	276	253	17	763	174	337	63	1	124	2963
	write/writeln	826	1507	10410	3407	6032	4880	3724	2	57354	2160	0	90302
	innerHTML	476	401	4356	982	35	1156	2057	2	3534	23	11	13033
	Total	15071	14334	101952	191301	45594	18788	7984	11206	165366	202883	2603	777082

```
1 a = document.createElement('img');
2 if (a.hasOwnProperty('src')) console.log("Chrome v35.0");
3 else   console.log ("Firefox v30.0");
```

(a)

```
1 try {  test();
2  function test(){ console.log("Chrome v35.0");}
3 } catch (e){ console.log("Firefox v30.0"); }
```

(b)

```
1 a = "(090) 4-4"
2 var b = a.replace(/[()-\s]+/g, '');
3 console.log("working on Chrome");
```

(c)

```
1 var status = false;
2 if (status === false) console.log("Firefox v30.0");
3 if (status === "false") console.log("Chrome v35.0");
```

(d)

Figure 6: Browser-specific Taint Flows: (a) Prototype vs. Instance Property. (b) Variable Scope Hoisting. (c) Notation for Regular Expressions. (d) Non-standard assignment to global variables.

TERJS crawled 228,541 URLs starting from the Alexa Top 1000 websites using its built-in crawler. The total instrumentation time was 143 hours. DEXTERJS successfully instrumented 13,255,378 HTML and 15,769,329 JavaScript files. At the end of the crawl 390,359,132 functions were executed which yielded a total of 777,082 unique taint flows. The complete breakdown of flows by sources and sinks is presented in Table 2. Of these, we focus only on the sinks listed in Table 3 as these are the ones directly relevant to dynamic markup generation. We restrict ourselves to 79,760 flows which are derived from the `location` object and its properties since verification of exploits from this source is easy to automate. In this category, DEXTERJS automatically generates working exploits for 820 distinct patch-points from 89 different domains. These domains include many high-profile websites such as *comodo.com*, *wsj.com*, *washingtonpost.com*, *bloomberg.com*, *usagc.org* (the US Green Card lottery website). The prevalence of DOM-based XSS among these popular websites motivate tools like DEXTERJS which finds and patches vulnerabilities automatically.

Robustness. In both our micro-benchmarks and large-scale evaluation, we verify the correctness of all the instrumented webpages by ensuring that the DOM-tree of the webpage remains unchanged after instrumentation automatically using the Selenium driver. The Selenium driver checks that the original DOM tree rendered and all sub-resources loaded by the benign input is the same as in the instrumented application. Further, we confirm that the transformed application does not throw any errors in the browser console.

6.2 Prevalence of Unsafe Coding Practices

We investigate the use of unsafe coding practices in Alexa Top 1000 sites. Out of 777,082 tainted data flows we obtained from the experiment, 13.3% use attacker-controlled data to generate HTML using functions such as `innerHTML`, `document.write` and `document.writeln`. This is potentially unsafe since the attacker can generate arbitrary HTML content or evaluate JavaScript code, given his control over a portion of the input to these critical code-evaluating constructs. More specifically, 0.97% of calls to `document.write`, 0.78% of `innerHTML` calls and 9.5% of `writeln` calls were confirmed to be exploitable by DEXTERJS automatically, which justifies the need for a tool that can auto-patch these vulnerabilities. The details about the fraction of exploitable sinks is presented in Table 3.

On the positive, we also find several instances of safe programmatic DOM construction methods used by developers. As Table 2 depicts, 566,175 (72.3%) of the flows utilize (to a minimalistic extent) programmatic DOM methods. We observe that even minimalistic programmatic DOM construction prevents exploitability to a large extent. For example, using the src property to set the location of an image (`getElementById('foo').src`) is safer than string interpolation of HTML (e.g. `document.write('<img src="'+...)`) since it prevents the attacker from escaping the attribute context.

6.3 Performance of Auto-patched Sites

DEXTERJS successfully synthesizes patches for all the DOM-based XSS exploits that were discovered. First, we verify the correctness of the patch by running each auto-patched website on IE, Chrome, Safari and Firefox web browsers via the crawler. Each time the vulnerable sink is reached, the patch is applied. We compare the DOM tree generated in the browser post-patching with the tree in the unpatched site using our Selenium-enabled infrastructure. We find that all the rendered DOM trees are similar to the original ones under benign inputs.

We measure the page load time for the auto-patched webpages. The overhead averaged over 10 runs is modest — ranging from 5.2% for Google Chrome, 6.45% for Internet Explorer, and 8.07% for Mozilla Firefox. This performance overhead is due to the replacement of the functions at the patch-points with the `patch` function and the increased number of DOM operations as compared to the vulnerable program. For example, in our running example, one DOM operation in the unpatched application (Line 5 of Figure 2(a)) corresponds to 7 DOM operations in the patched version (Lines 5 - 11 in Figure 2(b)). DOM construction API are presently unoptimized in browser implementations, which is the primary reason for these overheads.

DEXTERJS-generated patches are small in size. The average size of the patch is 3KB, whereas the average webpage size is 1.21MB.

Therefore, the code size impact to the original webpage is less than 0.3%. To test our patch security, we re-ran our initial set of attack patterns on the auto-patched site [21]. We report that all the tested attacks are nullified after auto-patching, as expected.

6.4 Browser-specific Taint Flows

The browser-agnostic nature of our instrumentation allows us to measure how sensitive the results of such taint analyses are to browser variance. To determine this, we collect a random sample of 100 taint flows that exhibit only on some browsers for manual investigation. A large fraction of these were missed in one browser versus the other because of our timeout — sites execute faster in certain browsers, thereby generating a greater number of flows in a given time interval. However, we found 3 instances of flows that exhibit only on Firefox and 2 only on Chrome out of a 100, due to differences in semantics of their JavaScript implementations. Though a 5% deviation is small, it explains how JavaScript implementations differ; therefore, formal analyses based on abstract language semantics may be easily inconsistent with the ground truth. We reduced the browser-unique flows to simple test cases (Figure 6) that exhibit subtle language implementation differences.

- *Prototype vs. Instance Property*: The specification dictates that DOM attributes should be a part of the DOM object instead of being part of their prototype. However, Chrome did not follow the spec before version 43 (Figure 6(a)).
- *In-Scope Variable Hoisting*: When a function is called before it is defined in a block scope, Chrome automatically does the hoisting, whereas Firefox doesn't (Figure 6(b)).
- *Notation in Regular Expressions*: When "-" occurs in the middle of the regular expression, Chrome treats it as hyphen but Firefox considers it as a range character (Figure 6(c)).
- *Non-standard assignment to global variables*: The behavior of Chrome and some versions of Firefox are different when global variables which are expected to be strings such as `window.status`, `window.name` are assigned non-string values (Figure 6(d)).

7. RELATED WORK

Preventing DOM-based XSS. Several solutions have been proposed to defeat DOM-based XSS. Though solutions such as Content Security Policy (CSP) and a capability controlled DOM are powerful [39], they require CSP or ES6 features to be supported by the browser and hence leave older browsers vulnerable. Implementing CSP in an existing application also requires extensive rewriting, further limiting its adoption [34, 60].

Lekies *et al*. use precise taint analysis to find flows which are vulnerable to DOM-based XSS similar to DEXTERJS [54]. However, since their taint analysis engine is implemented by modifying Chromium, they can miss flows due to the implementation differences of JavaScript semantics, as we show in our evaluation. One needs to carry out such analysis on various browser backends to account for such variation and porting it across a diverse range of HTML5 platforms and browser versions is an onerous task. In contrast to these solutions, DEXTERJS's taint analysis engine is not tied to any specific browser. Our patching technique employs a learning phase to figure out the expected document structure to be rendered on the client-side whereas the prevention technique proposed by the same authors is carried out by simply blocking the tainted flows and requires significant effort in specifying declassifiers to avoid false positives [56]. DEXTERJS requires minimal developer effort to patch the application.

Several defenses such as auto-sanitization have been integrated in web frameworks that emit JavaScript code [30, 52]. If the sanitization routines in auto-sanitized code are correct, and if all client-side code is generated using such web frameworks, these defenses serve as a panacea to the problem. However, a majority of existing frameworks either offer no auto-sanitization or incomplete sanitization [59]. Further, auto-sanitization approaches rely on the correctness of sanitizers which has been a problematic assumption [24, 40]. Mechanisms for using templatized or programmatic construction of the DOM [45], which our approach also relies on, sidesteps the assumption of correct sanitization.

JavaScript Analysis Tools. Static analysis techniques [25, 31, 35, 36, 46], dynamic analysis techniques [31, 42, 44, 48, 54] and mixed analyses [57, 58] for JavaScript applications have been extensively investigated. Static analysis approaches are notoriously difficult for JavaScript applications which lack static typing, have mutable class hierarchies, and a host of dynamic evaluation constructs, resulting in missing taint sources/sinks due to aliasing [46, 50]. Further, static analysis tools often have high false positives and do not generate concrete exploits. Dynamic analysis, therefore, has proven to be useful for taint-style security analyses. Most of dynamic taint-style analyses have been implemented in specific browser versions [9, 32, 48, 54]. These approaches require no modification to the original application and have been useful for conducting large-scale analyses. However, they tend to suffer from the same set of limitations as [54]. Source-to-source rewriting techniques offer the ability to sidestep these limitations as pointed in several previous works [36, 42, 44, 49, 61, 62].

Template generation and Auto-patching. Blueprint employs a set of *models* (akin to our *templates*) to preserve the integrity of intended document structure [45]. Similar to DEXTERJS, Blueprint requires no browser modification. However, it is built as a defense to secure HTML content generated by server-side code. Therefore, it does not account for dynamic HTML generated via JavaScript at the client-side which is required to prevent DOM-based XSS.

Our mechanism for auto-patching is conceptually related to several solutions for preventing reflected and persistent XSS in server-side languages. Previous studies such as ScriptGard and XSSGuard employ server-side parsing of HTTP responses [27,53]. They rewrite the application to enforce the intended structure. However, all of these techniques are insufficient in protecting against DOM-based XSS since they focus on patching server side code and not insecure JavaScript.

8. CONCLUSION

We presented DEXTERJS, a tool for auto-patching DOM-based XSS vulnerabilities. DEXTERJS is robust and scales to the Alexa Top 1000 on multiple browser backends. Using DEXTERJS, we patch hundreds of exploitable DOM-XSS vulnerabilities with a reasonable performance impact, thus making analysis-driven patching a practical alternative for securing JavaScript applications.

9. ACKNOWLEDGMENTS

We thank Benjamin Livshits, Adi Yoga Sidi Prabawa, Xinshu Dong and Amarnath Ravikumar for their constructive feedback on the paper. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work is supported in part by a university research grant from Intel.

10. REFERENCES

[1] A full-featured http proxy for node.js. http://goo.gl/fpPfum.

[2] A new modular browser. https://github.com/breach/breach_core.

[3] A Twitter DomXSS, a wrong fix and something more. http://goo.gl/dHF457.

[4] Analyzing a Dom-Based XSS in Yahoo! http://goo.gl/yXKtf4.

[5] Comma Operator - JavaScript | MDN. http://goo.gl/M738e.

[6] Crafty - JavaScript HTML5 Game Engine. craftyjs.com/.

[7] DexterJS online. https://dexterjs.io/.

[8] DOM XSS on Google Plus One Button. http://goo.gl/ohRAkM.

[9] DominatorPro: Securing Next Generation of Web Applications. https://dominator.mindedsecurity.com/.

[10] ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/.

[11] Fast, flexible, and lean implementation of core jQuery designed specifically for the server. https://github.com/cheeriojs/cheerio.

[12] Firing Range. http://public-firing-range.appspot.com/.

[13] LaBaSec: Language-based Security. http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1598.

[14] Mitmproxy: a man-in-the-middle proxy. http://mitmproxy.org/.

[15] Mobile/Tablet Market Share. http://goo.gl/Pcu492.

[16] node-os: First operating system powered by npm. http://node-os.com/.

[17] OS.js: JavaScript Cloud/Web Desktop Platform. http://osjsv2.0o.no/.

[18] PEG.js - Parser Generator for JavaScript. http://pegjs.majda.cz/.

[19] SeleniumHQ Browser Automation. http://seleniumhq.org/.

[20] The Chromium Blog. http://goo.gl/MIEOTW.

[21] XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[22] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[23] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 736–754, 2013.

[24] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 387–401. IEEE Computer Society, 2008.

[25] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 339–354. USENIX Association, 2010.

[26] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 91–100. ACM, 2010.

[27] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings*, volume 5137 of *Lecture Notes in Computer Science*. Springer, 2008.

[28] S. Bratus, J. Oakley, A. Ramaswamy, S. W. Smith, and M. E. Locasto. Katana: Towards Patching as a Runtime Part of the Compiler-Linker-Loader Toolchain. *IJSSE*, 1(3):1–17, 2010.

[29] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang. You Can't Be Me: Enabling Trusted Paths and User Sub-origins in Web Browsers. In *Research in Attacks, Intrusions and Defenses*, pages 150–171. Springer, 2014.

[30] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A Data-Centric Web Application Security Framework. In *2nd USENIX Conference on Web Application Development, WebApps'11, Portland, Oregon, USA, June 15-16, 2011*. USENIX Association, 2011.

[31] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 50–62. ACM, 2009.

[32] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 382–391. IEEE Computer Society, 2009.

[33] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1205–1216. ACM, 2013.

[34] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting csp to web applications. In *Proceedings of the 37th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2015)*, 2015.

[35] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

[36] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*. ACM, 2009.

[37] W. G. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 285–296, New York, NY, USA, 2009. ACM.

[38] W. G. J. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006, pages 175–185. ACM, 2006.

[39] M. Heiderich. *Towards elimination of xss attacks with a trusted and capability controlled dom*. Ruhr-University Bochum, 2012.

[40] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

[41] Jensen, Simon Holm and Jonsson, Peter A. and Møller, Anders. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44. ACM, 2012.

[42] E. Kiciman and V. B. Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, 2007.

[43] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium*, 2005.

[44] Koushik Sen and Swaroop Kalasapur and Tasneem G. Brutch and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013.

[45] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 331–346. IEEE Computer Society, 2009.

[46] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 499–509. ACM, 2013.

[47] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by inferring user interface state changes. In *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*, pages 122–134. IEEE, 2008.

[48] Prateek Saxena and Steve Hanna and Pongsin Poosankam and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010.

[49] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. volume 1, 2007.

[50] S. Saha, S. Jin, and K.-G. Doh. Detection of dom-based cross-site scripting by analyzing dynamically extracted scripts. In *The 6th International Conference on Information Security and Assurance*, 2012.

[51] M. Samuel and Ú. Erlingsson. Let's Parse to Prevent Pwnage. In *5th USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '12, San Jose, CA, USA, April 24, 2012*. USENIX Association, 2012.

[52] M. Samuel, P. Saxena, and D. Song. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 587–600. ACM, 2011.

[53] P. Saxena, D. Molnar, and B. Livshits. ScriptGard: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 601–614. ACM, 2011.

[54] Sebastian Lekies and Ben Stock and Martin Johns. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1193–1204. ACM, 2013.

[55] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 921–930. ACM, 2010.

[56] 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 Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 655–670. USENIX Association, 2014.

[57] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 49–59. ACM, 2014.

[58] S. Wei and B. G. Ryder. Practical Blended Taint Analysis for JavaScript. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 336–346. ACM, 2013.

[59] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. C. R. Shin, and D. Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2011.

[60] M. Weissbacher, T. Lauinger, and W. K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 212–233. Springer, 2014.

[61] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association.

[62] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 237–249, 2007.