Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions

Jinkun Pan
College of Computer
National University of Defense Technology
Changsha, China
pan_jin_kun@163.com

Xiaoguang Mao
College of Computer
Laboratory of Science and Technology on Integrated Logistics Support
National University of Defense Technology
Changsha, China
xgmao@nudt.edu.cn

***Abstract*—In recent years, with the advances in JavaScript engines and the adoption of HTML5 APIs, web applications begin to show a tendency to shift their functionality from the server side towards the client side, resulting in dense and complex interactions with HTML documents using the Document Object Model (DOM). As a consequence, client-side vulnerabilities become more and more prevalent. In this paper, we focus on DOM-sourced Cross-site Scripting (XSS), which is a kind of severe but not well-studied vulnerability appearing in browser extensions. Comparing with conventional DOM-based XSS, a new attack surface is introduced by DOM-sourced XSS where the DOM could become a vulnerable source as well besides common sources such as URLs and form inputs. To discover such vulnerability, we propose a detecting framework employing hybrid analysis with two phases. The first phase is the lightweight static analysis consisting of a text filter and an abstract syntax tree parser, which produces potential vulnerable candidates. The second phase is the dynamic symbolic execution with an additional component named shadow DOM, generating a document as a proof-of-concept exploit. In our large-scale real-world experiment, 58 previously unknown DOM-sourced XSS vulnerabilities were discovered in user scripts of the popular browser extension Greasemonkey.**

***Keywords*-Web security; browser extension vulnerability; DOM-sourced XSS; dynamic symbolic execution; shadow DOM; JavaScript**

I. Introduction

Modern web applications are increasingly moving program code to the client in the form of JavaScript. Meanwhile, more and more vulnerabilities occurring purely on the client through interacting with the Document Object Model (DOM) can be discovered. The DOM is a treelike structure providing APIs for accessing, traversing, and mutating the structure and content of HTML elements at runtime. One prevalent and severe class of client-side vulnerabilities is DOM-based cross-site scripting (XSS), which is caused by unsafe data flows from attacker-controlled sources, such as URLs and form inputs, into security sensitive APIs, resulting in malicious JavaScript code injection and execution. In a recent study, it is shown that approximately 10% of the Alexa Top 5000 websites carry at least one DOM-based XSS vulnerability [1]. In the research community, DOM-based XSS has attracted much attention [1–7].

However, there is a special kind of client-side XSS that has not been taken seriously and well-studied. We name it DOM-sourced XSS to distinguish it from conventional DOM-based XSS. DOM-based XSS emphasizes the vulnerability occurs during manipulating the DOM, whereas DOM-sourced XSS stresses the DOM as a new kind of attack surface. For regular website scripts, they might take the website DOM as an input, but the website DOM could never be controlled by the attacker. Meanwhile, the attacker-controllable website is not able to access the resources of the target website due to the Same-Origin Policy (SOP) complied by modern browsers which constrains the access to website resources within the same domain. However, this is not the case for browser extension scripts. The browser extension mechanism allows third parties to modify the browser's behavior, enhance its functionality and GUI, and integrate it with popular web services by injecting a piece of script. Such scripts are provided with many powerful privileges such as violating the SOP. In this way, a vulnerable browser extension script is able to take the attacker-controllable website DOM as a source, and execute the malicious code injected by the attacker accessing resources of the target website across the domain through the privilege, leading to a DOM-sourced XSS. Comparing with conventional DOM-based XSS, DOM-sourced XSS in browser extensions has two attacker-favorable features. First, DOM sources are more trusted by the developer than other sources, resulting in less sanitization. Second, the attacker-injected scripts are equipped with more powerful privileges.

In our work, we focus on a more specific kind of DOM-sourced XSS appearing in Greasemonkey. Greasemonkey is a very popular cross-browser extension enabling users to write lightweight JavaScript programs called user scripts (e.g. the example shown in Fig. 1) to manipulate loaded web pages on the client-side as desired. Large number of user scripts are created and shared among the community. Unfortunately, besides the convenience, user scripts bring about new security issues as well. Although Greasemonkey makes use of sandboxing to protect the privileged functions from possibly malicious scripts running on a website, a poorly-written user script can still introduce unsafe code vulnerable to DOM-sourced XSS in the sandboxed environment by executing a string from a malicious page without performing the proper sanity checks. More specifically, the directive of Greasemonkey specifying the target scope might be overly generic, allowing attacker-

978-1-5386-0992-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSME.2017.11

controlled web page DOM to be taken as the input for the user script. (e.g. the first wildcard used in line 8 of Fig. 1) Moreover, the Greasemonkey script is privileged with cross-domain requests, allowing attacker-controlled scripts to access resources from arbitrary domains by sending arbitrary requests towards any website with the user's cookies embedded in them, and reading the corresponding responses. (e.g. the privileged API *GM_xmlhttpRequest* granted in line 10 of Fig. 1 has the ability to perform cross-domain requests) Therefore, the DOM-sourced XSS in Greasemonkey is able to affect a broader range of targets, making more severe impact.

Unfortunately, there are no existing approaches which are able to discover DOM-sourced XSS in Greasemonkey. In this paper, we propose a detecting framework employing both static and dynamic techniques. First, a suspicious script is analyzed by the lightweight static analysis consists of a text filter and an abstract syntax tree (AST) parser. The text filter analyses the metadata to check overly generic directives and granted privileges. The AST parser searches the script code for potential attacker-controllable sources and security-sensitive sinks. If the script remains suspicious after the static analysis, the dynamic symbolic execution is then employed to detect potential exploitable flows. To tackle the problem of generating the exploiting hierarchical document (e.g. the one shown in Fig. 2) which is not supported by existing techniques, we introduce an additional shadow DOM component maintaining the hierarchical structure during the dynamic symbolic execution. The shadow DOM is able to avoid errors due to accessing non-existent DOM elements. Combining the structure from the shadow DOM and the values solved from the symbolic constraints by the SMT solver, a proof-of-concept document exploiting the vulnerable script can be constructed to demonstrate the presence of DOM-sourced XSS without any false positives.

To practically validate our approach, we conducted a large-scale experiment. Using a data set of 154,065 Greasemonkey user scripts, we successfully discovered 58 DOM-sourced XSS vulnerabilities with total 676,174 installations.

To summarize, this paper makes the following contributions:

- The notion of DOM-sourced XSS which emphasizes the hierarchical document source and distinguishes it from DOM-based XSS.
- An additional shadow DOM component during dynamic symbolic execution maintaining the hierarchical document structure, which is helpful in detecting DOM-sourced vulnerabilities.
- A hybrid analysis framework detecting DOM-sourced XSS in user scripts of the browser extension Greasemonkey.
- An empirical experiment which discovers 58 real-world vulnerabilities, demonstrating the existence of DOM-sourced XSS and the effectiveness of our detecting approach.

The remainder of this paper proceeds as follows: Section II describes the browser extension Greasemonkey. The notion of DOM-sourced XSS is introduced in Section III. In Section IV,

```
1.  // ==UserScript==
2.  // @name Demo
3.  // @author Author Name
4.  // @namespace http ://www. namespace. com
5.  // @version 1.0
6.  // @description This is a demo
7.  // @include http://target.com/*
8.  // @include http://*.target.com/*
9.  // @exclude http://login.target.com/*
10. // @grant GM_xmlhttpRequest
11. // @grant GM_log
12. // ==/UserScript==
13.
14. GM_log(" This is a demo ");
15. GM_xmlhttpRequest ({
16.       method: "GET",
17.       url: "http://www.test.com/",
18.       onload: function (response) {
19.             GM_log ( response.responseText );
20.       }
21. });
22. …
23. var main = document.getElementById("main");
24. var input = main.getElementsByTagName("input")[0];
25. var value = input.getAttribute("value");
26. eval(value);
27. …
```

Fig. 1. An example of the vulnerable user script *vulnerable.user.js*

```
<html>
  <body>
    <div id= "main">
      <input value= "doEvil()" ></input>
    </div>
  </body>
</html>
```

Fig. 2. An example of the exploit document *exploit.html*

we present our detecting approach aiming at DOM-sourced XSS. The implementation is illustrated in Section V. The experiment setup and results are described in Section VI. Section VII discusses the prevention measures and Section VIII presents related works. We conclude in Section IX.

II. GREASEMONKEY

A. Greasemonkey Browser Extension

Greasemonkey[1] is a very popular browser extension, allowing users to customize the way a web page displays or behaves. This is achieved by user-written JavaScript code injected into target web pages and executed in a context possessing access to privileged functionality. With the help of Greasemonkey, various user demands can be satisfied such as removing ads, adding auxiliary functionalities and meshing up contents from different sources.

While Greasemonkey itself is a browser extension, a user script run in Greasemonkey can be considered as a special kind of browser extension as well. Comparing with a conventional

[1]https://addons.mozilla.org/firefox/addon/greasemonkey

browser extension, a user script has a lot of advantages. First, it is lightweight and easy to develop. Unlike complex structure organization requirement of conventional browser extensions, the user script is self-described and only a single JavaScript file is needed. It can be inspected and edited just in the Greasemonkey extension, facilitating development to a great extent. Second, it is browser-independent. Greasemonkey was a Firefox extension originally, however, it is now implemented in all mainstream browsers, such as IEScripts for IE, Tampermonkey for Chrome, NinjaKit for Safari and Violentmonkey for Opera. Moreover, the community of user scripts is very active and a variety of useful scripts are shared in various forums and websites.

B. Greasemonkey User Scripts

A Greasemonkey user script consist of two parts, i.e., the metadata and the actual code. Figure 1 shows an example of the user script.

The metadata locates in the comments between "==UserScript==" and "==/UserScript==", which is in the form of "@*key value*". The metadata is read and used by the Greasemonkey engine to perform corresponding tasks. Here we introduce several important keys affecting the security of the script.

@*include*, @*exclude* and @*match* are used to define the target where the user script should be injected and executed. The script will execute if it matches any include rule, as long as it does not match an exclude rule. The rules are URLs, which can have a wildcard asterisk (*) matching any string including the empty string. If no include rule is provided, @*include* * is assumed. That is to say, every URL will be matched. The @*match* directive is very similar to @*include*, however it is safer. It sets stricter rules on what the * character means. @*match* wildcard is context-sensitive. A URL will be split into three contexts, i.e., the scheme, the host and the path. The * wildcard is limited in each context and cannot match characters across the context boundary. In cases where both @*include* and @*match* directives are used, the @*include* directive is handled first.

@*grant* is used to specify the privileged APIs used in the script. If a script does not specify any @*grant* values, Greasemonkey 1.0-1.9 will attempt to auto-detect the right settings. From Greasemonkey 2.0, @*grant* none is assumed by default, if no other values are specified. If a script specifies any values, then it will be provided with only those API methods that it declares. Otherwise the script will be granted no special API privileges, and thus run without the security constraints Greasemonkey scripts have traditionally had.

The actual code starts where the metadata ends. Besides regular functionalities provided by the browser, the code is augmented by special Greasemonkey-specific privileged APIs which have the prefix of *GM_*. The most notable privileged API is *GM_xmlhttpRequest*. This method performs a similar function to the standard *XMLHttpRequest* object, but allows these requests to cross the Same Origin Policy (SOP) boundaries. SOP essentially states that if content from one site is granted permission to access resources on the system, then any content from that site will share these permissions, while content from another site will have to be granted permissions separately. SOP is a core security policy implemented and complied by modern browsers, which is loosen by *GM_xmlhttpRequest*, making such API security-critical.

III. DOM-Sourced XSS in Greasemonkey

Cross-site scripting (XSS) [8] is a type of vulnerability in web applications. It is caused by inappropriate processed user input data flowing into security-critical sinks. XSS enables attackers to inject client-side script into web pages viewed by other users to bypass access controls such as SOP. XSS occurring in the client side script is referred as DOM-based XSS in previous researches. They focus on XSS appearing in the regular website scripts where only primitive data specifically strings can be used as script inputs, such as the URL and the form input. However, in the context of browser extensions, the DOM can also be treated as the attacker-controllable input source besides primitive data. We name the XSS sourced from the DOM input as DOM-sourced XSS.

In order to make the DOM controlled by the attacker to be the taint source of XSS, there are two obstacles. One is the limited target website scope restricted by the browser extension specified by the developer. In other words, the attacker-controlled website is usually out of the scope of the browser extension, making it unable to play the role of taint source. Another is the SOP complied by the browser, restricting the ability of the script in the attacker-controlled website, making it not security critical.

Nevertheless, they both can be overcame in the context of Greasemonkey. On the on hand, as we explained previously, the @*include* directive allows wildcard * to match any character, which might be used in a too generic way, leading to security problems. For instance, in the example of Figure 1, @*include http://*.target.com/** is an overly generic @*include*. The developer intends to match it against all sub domains of *target.com*. However, the attacker is able to abuse it by crafting a web page with the URL of *http://attacker.com/#.target.com/* which is hosted in the attacker's website and matches the @*include* rule as well. In this way, the limit of the restricted target scope can be compromised, enabling the DOM in the attacker-controlled website to be treated as a taint source. On the other hand, the privileged API *GM_xmlhttpRequest* has the ability to perform cross-domain requests, therefore the script in the attacker-controlled website is able to access resources of other websites, thus becoming security-critical. With these two conditions, the DOM can be treated as another attacker-controllable taint source of XSS.

Technically speaking, a DOM-sourced XSS vulnerability in a Greasemonkey user script should satisfy the following conditions:

- *Condition 1 (Global)*: The user script where the XSS happens has at least one overly generic @*include*.

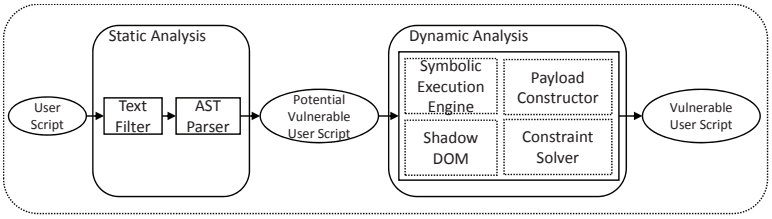

Fig. 3. Overview of our approach

TABLE I
OVERLY GENERIC *@include* RULE PATTERNS

Rule Pattern	Intention	Example	Exploit URL
http://*.*[the rest]*	use *.domain.com to capture all sub domains	http://*.target.com	http://attacker.com/exploit.html#.target.com
https://*.*[the rest]*	use *.domain.com to capture all sub domains	https://*.target.com	https://attacker.com/exploit.html#.target.com
htt*://*[the rest]*	use htt* to capture both http and https protocols	htt*://www.target.com	http://attacker.com/exploit.html#://www.target.com

- *Condition 2 (Privileged)*: The user script has the privilege of *GM_xmlhttpRequest* to perform cross-domain requests. For older versions of Greasemonkey, the privilege will be granted as long as it is invoked. Newer versions require the privilege should be granted explicitly in the metadata using the *@grant* directive.
- *Condition 3 (DOM-sourced)*: The input leading to the XSS comes from the DOM of the attacker-controlled web page.
- *Condition 4 (Eval-sink)*: The sink leading to the XSS is the function *eval* which executes arbitrary code from a string tainted by the attacker-controlled source. The execution context is inside the Greasemonkey sandbox which exposes the privileged Greasemonkey API.
- *Condition 5 (Exploitable Flow)*: There is a taint flow propagated from the DOM source to the *eval* sink which is not sanitized properly, making it exploitable.

Figure 1 shows an example user script vulnerable to DOM-sourced XSS. Line 8 defines an overly generic *@include* rule. Line 10 grants the privilege of *GM_xmlhttpRequest* explicitly. The value of the variable *source* in line 23 comes from the text content of a DOM element, which flows into the *eval* function as a parameter in line 26.

If such vulnerable script is installed in a user's browser, an attacker will be able to attack the victim through the following steps. First, the attacker crafts an exploiting document such as the one shown in Figure 2, which is designed to exploit the vulnerable user script. The *doEvil()* represents the payload which will be executed on the victim's browser with the ability to perform cross-domain requests through *GM_xmlhttpRequest*. Then, he uploads it to a website controlled by him, say *http://attacker.com/exploit.html*. Next, he crafts a URL such as *http://attacker.com/exploit.html#.target.com/* to match the overly generic *@include* rule of the vulnerable script and allures the victim to visit the crafted URL through methods like social engineering. Once the victim visits the URL, the payload *doEvil()* will be executed on the victim's browser to perform the malicious actions.

IV. DETECTING APPROACH

A. Overview

To discover the DOM-sourced XSS, we propose a detecting framework, the overview of which is shown in Figure 3. Our approach employs hybrid analysis, combining static analysis with dynamic analysis. The core component is the dynamic symbolic execution with an additional component named shadow DOM, which is able to generate document exploits accurately without false positives. However, the overhead is comparatively high. To reduce candidates for the dynamic analysis, we adopt the lightweight static analysis to filter out scripts not satisfying vulnerable conditions.

B. Lightweight Static Analysis

The lightweight static analysis consists of two components: the text filter and the AST parser.

1) Text Filter: The text filter treats the scripts as texts and processes them using string and regular expression operations. The text filter checks a script whether it satisfies the following prerequisites:

- The script should have at least one overly generic *@include* rule. We define patterns and exploit URLs of overly generic *@include* rules in Table I. These patterns are commonly appeared exploitable patterns in user scripts which are summarized through our investigation.
- The script should contain at least one "GM_xmlhttpRequest" either in the *@grant* directive or in the source code. If it only appears in the source code, it is directly vulnerable in older versions of Greasemonkey which will grant the privilege on demand. For newer versions, since the use of *GM_xmlhttpRequest* is intended by the script developer, the privilege has to be granted explicitly in order to make the script functional. Either way can guarantee that the privilege is granted.

Therefore, we consider a script is privileged as long as the "GM_xmlhttpRequest" appears.

All unsatisfied scripts will be filtered by the text filter. In this way, the *Condition 1* and *Condition 2* of the DOM-sourced XSS can be fulfilled.

2) AST Parser: To identify the DOM source and the *eval* sink, we employ a JavaScript abstract syntax tree (AST) parser. By traversing the AST, we are able to recognize the use of the global object *document* and the invokation of the *eval* function. A script will be filtered out due to the absence of either *document* use or *eval* invokation. By doing so, we eliminate scripts unsatisfying the *Condition 3* or *Condition 4* of the DOM-sourced XSS.

Both the text filter and the AST parser are lightweight techniques. They are able to eliminate most of unnecessary candidates for the dynamic analysis. We planned to employ more heavyweight static analysis technique like static taint analysis to reduce the candidates further. However, after several trials, we found existing static JavaScript analysis tools (e.g. SAFE [9], WALA [10] and TAJS [11, 12]) are not able to deal with the complex JavaScript features fully such as dynamic code evaluation, document and browser related APIs and the event system. Possible workarounds will result in too many false positives due to over-approximation, making the overhead of the heavyweight analysis unworthy. Therefore, we adopt only lightweight techniques in our approach.

C. Dynamic Analysis

Considering the last condition of the DOM-sourced XSS, we would like to check whether there is an exploitable flow from the source to the sink. If so, we want to get an exploiting input like the one shown in Figure 2. To achieve this, both the document structure and element values such as attribute values and text contents of elements should be generated.

Since the element values are strings, they can be generated by dynamic symbolic execution [13] cooperated with a string-supported constraint solver. The key idea of symbolic execution is to represent the values of program variables as symbolic expressions rather than concrete data values. A path condition is maintained and updated whenever a branch instruction is executed, encoding the constraints on the inputs that reach that program point. The constraints can be solved through a Satisfiability Modulo Theories (SMT) [14] solver to obtain satisfiable inputs or counterexamples in violation cases. In dynamic symbolic execution, when constraints cannot be solved symbolically, concrete values are provided to ensure completion of the process. As for the document structure, it is beyond the ability of the conventional dynamic symbolic execution.

1) Dynamic Symbolic Execution with Shadow DOM: To deal with the problem of hierarchical document structure, we propose an enhanced dynamic symbolic execution which is augmented with an additional component named shadow DOM. During the symbolic execution, the shadow DOM is responsible for maintaining the document structure and

Algorithm 1 The algorithm for maintaining the shadow DOM during trace analysis

Input: The execution trace of the script $Trace$ and the input document $OriginDOM$
Output: The updated shadow document $ShadowDOM$ reflecting the required structure

```
1: function TRACEANALYSIS(Trace)
2:     ShadowDOM ← CLONE(OriginDOM)
3:     for each Statement in Trace do
4:         Type ← GETTYPE(Statement)
5:         Error ← GETERROR(Statement)
6:         if Type == "GetField" then
7:             Field ← GETFIELD(Statement)
8:             Value ← GETVALUE(Statement)
9:             ValueType ← GETTYPE(Value)
10:            if ISDOMRELATEDFIELD(Field) then
11:                if ISACCESSINGNULL(Error) then
12:                    UPDATE(ShadowDOM, Field)
13:                end if
14:                if ISPRIMITIVE(ValueType) then
15:                    Value ← SYMBOLIC(Value)
16:                end if
17:            end if
18:        else if Type == "InvokeFunction" then
19:            Fun ← GETFUNCTION(Statement)
20:            Args ← GETARGUMENTS(Statement)
21:            if ISDOMRELATEDFUNCTION(Fun) then
22:                if ISACCESSINGNULL(Error) then
23:                    UPDATE(ShadowDOM, Fun)
24:                end if
25:            else if ISEVAL(Fun) then
26:                if ISSYMBOLIC(Args[0]) then
27:                    PayloadConstraint ← GENPAYLOADCONSTRAINT(Args[0])
28:                    Res ← SOLVE(PayloadConstraint)
29:                    if Res != null then
30:                        UPDATE(ShadowDOM, Res)
31:                        return ShadowDOM
32:                    end if
33:                end if
34:            end if
35:        else if Type == "Branch" then
36:            PC ← UPDATEPATHCONSTRAINTS( )
37:            NegatedPC ← NEGATECONSTRAINTS(PC)
38:            if NegatedPC != null then
39:                Res ← SOLVE(NegatedPC)
40:                if Res != null then
41:                    UPDATE(ShadowDOM, Res)
42:                    return ShadowDOM
43:                end if
44:            end if
45:        end if
46:    end for
47: end function
```

updating element values by interacting with the SMT string solver.

The procedure of our technique is illustrated in Algorithm 1, which emphasizes the difference from the conventional dynamic symbolic execution. The script is instrumented to generate the trace of dynamic execution. Given the trace, the script will be executed once again symbolically.

In the beginning of the symbolic execution, the shadow DOM is a clone of the input document. Once a document-related field of primitive type is encountered, which represents the attribute value or text content of a document element, it will be treated as a symbolic variable. (Line 15 in Algorithm 1) As the script is executed statement by statement, both the symbolic expression and the shadow DOM will be updated accordingly. The update of the symbolic expression is just as the conventional dynamic symbolic execution. As for the shadow DOM, there are four places where it needs to be updated. The first is when accessing a non-existing document-related filed. (Line 10-13) The second is when invoking a document-related function which accesses a non-existing document element. (Line 21-24) The shadow DOM will be augmented with the missing element in the first two cases. The third is when a branch statement is encountered. (Line 35-44) In order to explore different paths of the program, a negated constraint will be generated and solved through an SMT constraint solver for unvisited branches. If the constraint is solvable, then the shadow DOM will be updated by replacing all symbolic variables with solved values and taken as the next input and the analysis will start over again. The last one is when the *eval* function is encountered. (Line 25-34) If the argument is symbolic, then a payload constraint will be generated for it. If the payload constraint is satisfiable, a document exploit can be constructed through updating all solved symbolic variables.

2) Shadow DOM: Table II presents the details of how the shadow DOM is updated according to the document-related functions and fields. When a missing element is accessed, the shadow DOM will be updated by augmenting a relevant element which satisfies the accessing conditions. We focus on the most commonly used document-accessing functions and fields which are shown in the first column.

For *getElementById*, a *div* element is created with the specified id and appended to the shadow DOM. The value is set to be "placeholder" to avoid an empty value which might cause failure in presence check.

As for *getElementsByTagName*, an element of the specified tag is appended. A random id is necessary for later reference, especially when finding relevant elements in the shadow DOM.

For *getAttribute*, before the missing attribute is added, it should be checked that whether the element is able to hold that attribute. This is because for those elements generated by the shadow DOM instead of the script code, the *div* element is the default like the one created for *getElementById*, which might not suit for the attribute. Therefore, the tag of the element should be changed when necessary. For the example presented in the table, the *div* element is replaced with the *input* element since there is no "value" attribute for a *div* element.

The next group of fields refer to elements with specified structure relationship. *children*, *firstElementChild* and *lastElementChild* are elements of the under layer. *parentNode* refers to the element in the upper layer. *previousSibling* and *nextSibling* indicate elements in the same hierarchy. The created element should be placed to the appropriate position of the shadow DOM accordingly using proper functions such as *appendChild* and *insertBefore*.

The XPath (parameter of *evaluate*) and the CSS selector (parameter of *querySelector* and *querySelectorAll*) are two kinds of expressive expressions used to locate elements satisfying specified structure and value conditions. Each of them has plenty of complex features. At present, the shadow DOM supports basic commonly used features of them. For other fields, if they have corresponding element attributes, they will be processed just like the *getAttribute*.

Other element-specific document-related fields are processed case by case, such as the special field *elements* of the *form* element in the example shown in the table.

3) Payload Constructor: The basic form of the payload is straightforward which invokes the privileged API to access resources across domains and does something evil. However, there are two more problems should be considered to make the payload actually work, i.e., the sanitization and the context.

In some situations, the value might be sanitized by certain sanitization process before flowing into the *eval* sink. Although it is possible to model such process through constraints, some complex operations such as transducers using regular expressions are not supported by existing constraint solvers. To tackle such problem, we adopt several heuristic escaping methods to transform the basic payload into various forms, trying to bypass the sanitization. For instance, characters in the payload might be replaced with corresponding JavaScript octal escape characters in the form of $\backslash OOO$ (O represents an octal digit) or hexadecimal escape characters in the form of $\backslash xHH$ or $\backslash uHHHH$ (H represents a hexadecimal digit).

Moreover, in some cases, the taint values flowing into the *eval* sink are surrounded by certain contexts. Some context-dependent break out sequences should be generated to ensure that the payload is executed correctly without throwing errors. For example, for the code *eval("a=["+taint+"]")*, the break out sequence *];* is needed before the payload as well as a comment symbol after the payload, resulting in the execution of *a=[];payload//]* where the payload is executed properly. To construct the context-dependent break out sequences, we parse the parameter string into an AST. By traversing the AST, the position of the payload can be located. According to the hierarchy, we construct closing sequences for each level to break out that level, and concatenate them from bottom to top to form the final break out sequences. At last, the payload will be able to be executed in the top level.

TABLE II
PROCESSES AND EXAMPLES OF SHADOW DOM MAINTENANCE FOR CORRESPONDING DOM-RELATED OPERATIONS

DOM-related Operation	Shadow DOM Maintenance	Example
getElementById(*identifier*)	1. check whether there exists an element with the id of *identifier*. If so, return; 2. create an element with the tag of div; 3. set the element id to be *identifier*; 4. set the text content of the element to be "placeholder"; 5. append the element to the body of the document.	getElementById("abc") <body></body> => <body> placeholder </body>
getElementsByTagName(*tagName*)	1. check whether there exist elements with the tag of *tagName*. If so, return; 2. create an element with the tag of *tagName*; 3. set the element id randomly; 4. set the text content of the element to be "placeholder"; 5. append the element to the body of the document.	getElementsByTagName("input") <body></body> => <body><input id="564987"> placeholder</input></body>
ele.getAttribute(*attrName*)	1. find the element *eleShadow* in the shadow DOM corresponding to *ele*; 2. check whether there exists an attribute *attrName* for *eleShadow*. If so, return; 3. check whether the type of *eleShadow* is able to hold the *attrName*. If not, change the tag of *eleShadow* to a tag which is able to have *attrName*; 4. set the value of attribute *attrName* for *eleShadow* to be "placeholder".	getElementById("abc").getAttribute("value") <body> </body> => <body><input id="abc" value="placeholder"> </input></body>
ele.children *ele*.firstElementChild *ele*.lastElementChild *ele*.parentNode *ele*.previousSibling *ele*.nextSibling	1. find the element *eleShadow* in the shadow DOM corresponding to *ele*; 2. check whether there exists an element satisfying the relationship with respect to *eleShadow* described by the field. If so, return; 3. create an element with the tag of div; 4. set the element id randomly; 5. set the text content of the element to be "placeholder"; 6. put the created element in appropriate place with respect to *eleShadow* using DOM API such as appendChild and insertBefore.	getElementById("abc").children <body> </body> => <body> placeholder </body>
evaluate(*xpath*, ...) querySelector(*selector*) querySelectorAll(*selector*)	1. check whether there exists a node satisfying the *xpath* or *selector*. If so, return; 2. For the querySelector and querySelectorAll, translate the CSS selector *selector* to the form of XPath *xpath*; 3. parse the *xpath* and generate all required elements, attributes and text contents along the parsed path iteratively.	evaluate("//div[@class='xyz']/input", ...) <body></body> => <body> placeholder<input id="278565"> placeholder</input> </body>
ele.fieldName	1. find the element *eleShadow* in the shadow DOM corresponding to *ele*; 2. check whether there exists a field *fieldName* for *eleShadow*. If so, return; 3. if there is a corresponding attribute for the field *fieldName*, process it as the getAttribute; 4. if the field *fieldName* is related with the document structure, process it case by case for different types of elements.	getElementById("abc").elements <body><form id="abc"></form></body> => <body><form id="abc"><input id="495843"> placeholder</input></form></body>

V. IMPLEMENTATION

The text filter is written in Python. The Esprima[2], a high performance standard-compliant ECMAScript parser written in JavaScript, is adopted to implement the AST Parser. The dynamic symbolic execution with shadow DOM is based on the concolic testing platform of Jalangi[3]. Jalangi is a selective record-replay and dynamic analysis framework for JavaScript with a built-in concolic testing engine written in JavaScript. The CVC3 SMT solver[4] is employed by Jalangi to solve string constraints. The Java-written automata[5] is used to encode regular expressions. All JavaScript code is executed in the context of Node.js, therefore a mock browser[6] library is adopted to simulate the browser and document related APIs.

We would like to make our approach completely automatic. However, during the experiment, manual analysis is adopted due to two reasons. On the one hand, some issues are caused by limitation of our current implementation. On the other hand, some problems are not solvable through existing techniques automatically. Specifically, the following issues are resolved by manual analysis.

- Due to the event-driven nature of web applications, the execution of callback functions such as event listeners rely on the trigger of special events. That is to say, besides the program space, the event space should also be explored in order to detect vulnerabilities. In practice, we generate event sequences for each script manually to compose suspicious flows. Techniques such as SymJS [15] might be adopted in the future to solve the event space exploration problem.
- The support for regular expression operations of existing SMT string solvers is limited in membership checking. Unfortunately, there are many transduce operations such as replace strings using regular expressions, which are not supported. We have to solve them manually.

VI. EXPERIMENT

A. Setup

To evaluate our approach, we collected user scripts from three sources: *userscripts*[7], *greasyfork*[8] and *openuserjs*[9], which are online community websites hosting shared Greasemonkey user scripts. We crawled all scripts available from these websites and related meta information such as the number of installations. In total, we collected 154,065 scripts as the raw dataset for evaluation.

We run all the experiments on a desktop computer with an Intel Core i7 3.5 GHz processor and 4GB RAM using Ubuntu 14.04 LTS.

[2]http://esprima.org

[3]https://people.eecs.berkeley.edu/∼gongliang13/jalangi_ff

[4]http://www.cs.nyu.edu/acsys/cvc3

[5]http://www.brics.dk/automaton/automaton.jar

[6]http://www.npmjs.com/package/mock-browser

[7]http://userscripts-mirror.org. This is a mirror site of http://userscripts.org. We collected data from it because the original website is not available.

[8]https://greasyfork.org

[9]https://openuserjs.org

TABLE III
STATISTICS OF EACH STEP

Step	# of Input Scripts	# of Remaining Scripts	Total Time(s)	Average Time(s)
Text Filter	154,065	1,195	49.794	3.232E-04
AST Parser	1,195	838	1201.739	1.006
Dynamic Symbolic Execution with Shadow DOM	838	58	106613.306	127.224

B. Result

We evaluated all collected user scripts using our approach, the number of input and remaining scripts and the time cost in each step are shown in Table III. The text filter eliminated a majority of the dataset, retaining only 1,195 scripts. After the AST parser, 838 scripts were considered suspicious. In the last step of the dynamic symbolic execution with shadow DOM, 58 scripts were proved to be vulnerable. Note that our approach will not introduce false positives since every detected vulnerability is verified by the real execution. In other words, once a script is detected by our approach, it is definitely vulnerable. However, our approach might miss some vulnerable scripts due to the incompleteness of the support on DOM-manipulation operations in our shadow DOM. Supporting more operations to decrease the false negatives is one of our future work.

The timeout is set to be one minute each script for the first two steps, and ten minutes for the last step. The text filter and AST parser are lightweight techniques, so the time spent on each script is not much, which are around 0.3 milliseconds and one second, respectively. The dynamic symbolic execution with shadow DOM is the comparatively most time-consuming step, the average time of which is around two minutes each.

To explore detected vulnerable scripts further, we present the detailed statistics of them in Table IV. We cluster the detected 58 vulnerable scripts into 18 groups. Scripts in each group have similar vulnerable code. The main cause of such similarities is the fact that they are different versions of the same functionality written by the same developer. For example, the group No.5 has 31 versions of the BB-Code Editor, sharing the same vulnerable code. The first six columns are information about the vulnerable scripts, and the last three columns are about the generated exploit documents. The first column is the group number. The second column is the name of a representative script in a group, which gives a hint of what the scripts are designed for. As we can see, the vulnerable scripts are developed for various purposes. The lines of code (LOC) representing the code complexity is shown in the third column. The number of similar scripts in a group is presented in fourth column. The number of accumulated installations as a measure of the potential impact caused by the vulnerable script is displayed in the fifth column. The sixth column indicates whether the vulnerable script is granted with the *GM_xmlhttpRequest* privilege explicitly by the developer.

TABLE IV
STATISTICS OF DETECTED VULNERABLE SCRIPTS AND GENERATED DOCUMENTS

Grouped Script						Generated Document		
Group No.	**Name**	**LOC**	**# of Scripts**	**# of Installations**	**Explicitly Granted**	**# of Tags**	**# of Attributes**	**Time(s)**
1	FreeRice Hacked: ajax++	286	1	1,934	No	7	9	3.734
2	Mako video	412	1	2,255	No	4	1	3.759
3	LJ Thread Expander	571	1	49,070	No	5	2	2.122
4	vu_karakterlap	1,434	1	2,104	No	30	30	19.925
5	BBCode-Edit	1,307	31	177,084	No	5	3	3.572
6	Travian Report Processor	2,422	2	47,378	No	7	6	4.846
7	Free Youtube!	261	2	69,130	Yes	5	4	3.703
8	YouTube HD Download Button	1,614	4	39,285	No	7	5	6.355
9	MyGoogleMonkeyR	2,335	1	4,140	No	36	34	273.987
10	Mob Wars Turbo - Jonatan	2,085	1	270	No	16	13	12.861
11	Planetromeo Gayromeo Filter	1,124	2	53,340	No	44	37	43.587
12	Comunio - SCRIPT - Beta	7,140	1	143,644	No	9	6	11.309
13	Youtube Remove Ads In-Video and Outside-Video	622	5	24,912	No	6	3	23.676
14	View image links	1,304	1	2,920	No	29	27	27.067
15	oroszfordit	1,392	1	1,905	No	25	29	360.468
16	Conquer Club Assault Odds	625	1	36,411	No	11	8	7.559
17	vu_omni	1,585	1	17,323	Yes	14	11	8.331
18	Travian	547	1	3,069	No	23	26	14.296

The seventh and eighth columns show the number of HTML tags and attributes in the document generated by the shadow DOM, representing the complexity of the document. The last column shows the average time cost by generating the exploit document, which varies a lot from script to script. Generally speaking, the time is correlated with the complexity of the script and the generated document.

The total number of installations of all these 58 vulnerable user scripts is up to 676,174, indicating a great potential influence. Among them, three scripts (two in group 7 and one in group 17) are designed for the latest Greasemonkey which grant the privilege explicitly, thus directly exploitable in the latest Greasemonkey. These three scripts sum to 86,453 installations. The rest scripts are design for older versions of Greasemonkey, therefore they use the *GM_xmlhttpRequest* without granting the privilege explicitly. That is to say, the privilege will be inferred and granted automatically by the older versions of Greasemonkey, making the scripts exploitable. Alternatively, the scripts might be upgraded to be explicitly granted by the user in order to achieve the intended functionality in the latest Greasemonkey. In this way, they will become exploitable as well. The total installations of these scripts are counted up to 589,721.

VII. PREVENTION

As we described previously, there are five conditions for DOM-sourced XSS in Greasemonkey user scripts. Accordingly, we can prevent the vulnerability through the following measures.

- Avoid overly generic *@include* rules. We can either define *@include* rules more carefully and specifically or use the safer directive *@match* instead. In this way, the attacker will not able to control the DOM input.
- Upgrade the Greasemonkey to the latest version to take advantage of the new security mechanism. Latest Greasemonkey isolates privileged user script code from insecure content pages, restricting the powerful privileged API within the sandbox. Moreover, it provides least privileges by default unless additional privileges are specified by the user explicitly.
- Avoid using *GM_xmlhttpRequest* if not necessary. Based on the least privilege principle, powerful functions like the privileged *GM_xmlhttpRequest* API should be available to user scripts only if they are absolutely necessary, avoiding being abused by the attacker.
- Avoid taking data dependent on values from user-controlled sources as the parameter of *eval* function. If it is necessary, ensure that the input data is sanitized properly.

VIII. RELATED WORK

DOM-based XSS: DOM-based XSS was first proposed by Klein [2] in 2005. Saxena et al. [3] built FLAX adopting black box fuzzing enhanced with taint information to detect client-side validation vulnerabilities, including DOM-based XSS. Lekies et al. [1] conducted a large-scale empirical study on DOM-based XSS. They modified the JavaScript engine to track byte-level taint information dynamically and

proposed a validation system. Considering the dynamic nature of JavaScript code, Tripp et al. [16] proposed JSA, which combines static string analysis with partially evaluated JavaScript code. To protect the client against DOM-based XSS, Stock et al. [4] proposed a protection mechanism which tracks the insecure data precisely and stops the tainted data during parse time. Weissbacher et al. [17] employed invariant techniques to defend clients from attacks through a proxy named ZigZag. Parameshwaran et al. [5] synthesized patches for DOM-based XSS by constructing safe DOM manipulating API to replace insecure dynamic generated code. Stock et al. [6] presented a large-scale empirical study to analyze the causes of DOM-based XSS in depth. Gupta et al. [7] proposed JS-SAN, which generates a template by clustering various payloads and places appropriate sanitizers. Unlike all these existing researches, in this work, we focus on DOM-sourced XSS rather than DOM-based XSS.

Browser Extension Security: Bandhakavi et al. [18] presented VEX, a static information flow analysis framework to detect vulnerabilities in Firefox extensions. Guha et al. [19] proposed IBEK for verifying the security of extensions through formal analysis. Wang [20] conducted an empirical study of unsafe behaviors related with web sessions of Firefox extensions. Carlini et al. [21] performed a large-scale study of security mechanisms in Chrome extensions. Liu et al. [22] revealed several serious threats of Chrome extensions through bot-based attacks. Kapravelos et al. [23] presented Hulk, which employs HoneyPages adapted to expectations of extensions and a fuzzer to trigger various events. Marston et al. [24] focused on risks of Firefox extension in Android platform. Zhao et al. [25] proposed LvDetector to discover information leakage vulnerabilities in browser extensions. Sentinel [26] enforces the user-provided fine-grained policies for legacy Firefox extensions. Jagpal et al. [27] reported the trends of malicious extensions over three years and lessons learned in fighting against them. Saini et al. [28] presented a new attack taking advantage of colluding browser extensions. CrossFire [29] detects a new kind of extension-reuse vulnerability in Firefox through a multi-stage and lightweight static analysis. The most relevant research is [30], which performs an empirical analysis of user scripts for Greasemonkey and discovers several malicious extensions and vulnerabilities. However, they deal with a wide range of vulnerabilities and their straightforward static analysis is very coarse and inaccurate, resulting in a lot of false positives. In our work, we focus on DOM-sourced XSS and perform a much more precise analysis providing proof-of-concept exploits.

JavaScript Symbolic Execution: Kudzu [31] is a symbolic execution framework, aiming at detecting JavaScript code injection vulnerabilities in AJAX applications. Jalangi [32] is a flexible dynamic analysis framework incorporating selective record-replay and shadow values. A concolic execution component is implemented in Jalangi, on which our work is based. Li et al. [15] proposed SymJS, a JavaScript symbolic execution engine augmented with an automatic event explorer to explore the event space in a more efficient way. MultiSE [33] enhances conventional dynamic symbolic execution by representing states using value summaries instead of merged paths. To eliminate the redundant inputs, Dhok et al. [34] proposed a type-aware approach by grouping and inferring type preconditions of variables. Fard et al. [35] considered the DOM as input for JavaScript symbolic execution engine to increase coverage of unit testing, which is very inspiring. However, they rely on an existing XML constraint solver, inheriting its limitation thus supporting limited DOM manipulating operations. Moreover, they concentrate on structural constraints not thinking much of other constraints. In addition, they aim at the application context of unit testing. Instead, we employ the shadow DOM supporting various operations and interacting with the sophisticated SMT string solver to detect vulnerabilities in the whole program.

IX. Conclusion

DOM-sourced XSS is a noteworthy vulnerability which has not been paid much attention to in the research community. In this work, we have proposed the notion and a hybrid detection approach combining lightweight static analysis and dynamic symbolic execution. The shadow DOM technique is introduced to tackle the problem of hierarchical document inputs. Experiment results validate the presence of DOM-sourced XSS and the effectiveness of our detecting approach. At present, our technique needs manually analysis assistance. In the future, we plan to automatize it and make it more robust. Moreover, we would like to explore more potential vulnerable browser extensions other than Greasemonkey.

Acknowledgment

This research was supported in part by grants from National Natural Science Foundation of China (Nos. 61672529, 61379054 and 91318301).

References

[1] S. Lekies, B. Stock, and M. Johns, "25 million flows later - large-scale detection of dom-based xss," in *Proceedings of the ACM Sigsac Conference on Computer & Communications Security*, 2013, pp. 1193–1204.

[2] A. Klein, "Dom based cross site scripting or xss of the third kind," *Web Application Security Consortium*, vol. 4, pp. 365–372, 2005.

[3] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[4] 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 USENIX Security Symposium*, 2014.

[5] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching dom-based xss at scale," in *Proceedings of the ACM Sigsoft Symposium on the Foundations of Software Engineering*, 2015, pp. 272–283.

[6] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: Exploring client-side cross-site scripting," in *ACM Sigsac Conference on Computer and Communications Security*, 2015, pp. 1419–1430.

[7] S. Gupta and B. B. Gupta, "Js-san: defense mechanism for html5-based web applications against javascript code injection

vulnerabilities," *Security & Communication Networks*, vol. 9, no. 11, pp. 1477–1495, 2016.

[8] V. Nithya, S. L. Pandian, and C. Malarvizhi, "A survey on detection and prevention of cross-site scripting attack," *International Journal of Security & Its Applications*, vol. 9, no. 3, pp. 139–152, 2015.

[9] C. Park, S. Won, J. Jin, and S. Ryu, "Static analysis of javascript web applications in the wild via practical dom modeling," in *Proceedings of the International Conference on Automated Software Engineering*, 2015, pp. 552–562.

[10] M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," *Acm Sigplan Notices*, vol. 48, no. 48, pp. 165–174, 2013.

[11] S. H. Jensen, A. Moller, and P. Thiemann, "Type analysis for javascript." in *Proceedings of the International Static Analysis Symposium*, 2009, pp. 238–255.

[12] S. H. Jensen, M. Madsen, and A. Moller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the ACM Sigsoft Symposium on the Foundations of Software Engineering*, 2011, pp. 59–69.

[13] C. Cadar, P. Godefroid, S. Khurshid, C. S. Reanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 1066–1071.

[14] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.

[15] G. Li, E. Andreasen, and I. Ghosh, "Symjs: Automatic symbolic testing of javascript web applications," in *Proceedings of the ACM Sigsoft Symposium on the Foundations of Software Engineering*, 2014, pp. 449–459.

[16] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web javascript code via dynamic partial evaluation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 49–59.

[17] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Zigzag: Automatically hardening web applications against client-side validation vulnerabilities," in *the Proceedings of the USENIX Security Symposium*, 2015.

[18] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: vetting browser extensions for security vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2010, p. 339C354.

[19] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011, pp. 115–130.

[20] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng, "An empirical study of dangerous behaviors in firefox extensions," in *Proceedings of the International Conference on Information Security*, 2012, pp. 188–203.

[21] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the google chrome extension security architecture," in *Proceedings of the USENIX Security Symposium*, 2012.

[22] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2012.

[23] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proceedings of the USENIX Security Symposium*, 2014.

[24] J. Marston, K. Weldemariam, and M. Zulkernine, "On evaluating and securing firefox for android browser extensions," in *Proceedings of the International Conference on Mobile Softeware Engineering and Systems*, 2014, pp. 27–36.

[25] R. Zhao, C. Yue, and Q. Yi, "Automatic detection of information leakage vulnerabilities in browser extensions," in *International Conference on World Wide Web*, 2015.

[26] K. Onarlioglu, A. S. Buyukkayhan, W. Robertson, and E. Kirda, "Sentinel: Securing legacy firefox extensions," *Computers & Security*, vol. 49, pp. 147–161, 2015.

[27] N. Jagpal, E. Dingle, J. P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, "Trends and lessons from three years fighting malicious extensions," in *Proceedings of the USENIX Security Symposium*, 2015.

[28] A. Saini, M. S. Gaur, V. Laxmi, and M. Conti, "Colluding browser extension attack on user privacy and its implication for web browsers," *Computers & Security*, vol. 63, pp. 14–28, 2016.

[29] A. Buyukkayhan, Salih, K. Onarlioglu, W. Robertson, and E. Kirda, "Crossfire: An analysis of firefox extension-reuse vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[30] V. Acker, Steven, Nikiforakis, Nick, Desmet, Lieven, Piessens, Frank, Joosen, and Wouter, "Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets," *Cw Reports*, 2014.

[31] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. Mccamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the IEEE Symposium on Security & Privacy*, vol. 41, no. 3, 2010, pp. 513–528.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[33] K. Sen, G. Necula, G. Liang, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2015, pp. 842–853.

[34] M. Dhok, M. K. Ramanathan, and N. Sinha, "Type-aware concolic testing of javascript programs," in *Proceedings of the International Conference on Software Engineering*, 2016.

[35] A. M. Fard, A. Mesbah, and E. Wohlstadter, "Generating fixtures for javascript unit testing," in *Proceedings of the International Conference on Automated Software Engineering*, 2015, pp. 190–200.