A Measurement Study of Insecure JavaScript Practices on the Web

CHUAN YUE, University of Colorado Colorado Springs
HAINING WANG, The College of William and Mary

JavaScript is an interpreted programming language most often used for enhancing webpage interactivity and functionality. It has powerful capabilities to interact with webpage documents and browser windows, however, it has also opened the door for many browser-based security attacks. Insecure engineering practices of using JavaScript may not directly lead to security breaches, but they can create new attack vectors and greatly increase the risks of browser-based attacks. In this article, we present the first measurement study on insecure practices of using JavaScript on the Web. Our focus is on the insecure practices of JavaScript inclusion and dynamic generation, and we examine their severity and nature on 6,805 unique websites. Our measurement results reveal that insecure JavaScript practices are common at various websites: (1) at least 66.4% of the measured websites manifest the insecure practices of including JavaScript files from external domains into the top-level documents of their webpages; (2) over 44.4% of the measured websites use the dangerous eval() function to dynamically generate and execute JavaScript code on their webpages; and (3) in JavaScript dynamic generation, using the document.write() method and the innerHTML property is much more popular than using the relatively secure technique of creating script elements via DOM methods. Our analysis indicates that safe alternatives to these insecure practices exist in common cases and ought to be adopted by website developers and administrators for reducing potential security risks.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; H.4.3 [**Information Systems Applications**]: Communications Applications—*Information browsers*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Scripting languages*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms: Design, Experimentation, Languages, Measurement, Security

Additional Key Words and Phrases: JavaScript, execution-based measurement, security, same origin policy, AST tree matching, Web engineering

ACM Reference Format:
Yue, C. and Wang, H. 2013. A measurement study of insecure JavaScript practices on the web. ACM Trans. Web 7, 2, Article 7 (May 2013), 39 pages.
DOI: http://dx.doi.org/10.1145/2460383.2460386

1. INTRODUCTION

Security is an important aspect of Web engineering, and it should be taken into serious consideration in the development of high-quality Web-based systems [Ceri et al. 2002; Kappel et al. 2006; Mendes and Mosley 2005; Murugesan and Deshpande 2001; Powell et al. 1998]. In many cases, however, security does not receive sufficient attention due to the complexity of Web-based systems, the ad hoc processes of system development,

---

A short version of this article appears in *Proceedings of the International World Wide Web Conference* [Yue and Wang 2009].
Authors' addresses: C. Yue (corresponding author), University of Colorado, Colorado Springs; email: cyue@uccs.edu; H. Wang, The College of William and Mary.

DOI: http://dx.doi.org/10.1145/2460383.2460386

and even the fact that many designers or developers lack security knowledge on Web development techniques. It is not a surprise, therefore, that website security breaches are common [Suh 2005] and Web applications are more susceptible to malicious attacks than traditional computer applications [Rossi et al. 2007].

Browser-based attacks have posed serious threats to the Web in recent years. Exploiting the vulnerabilities in Web browsers [Reis et al. 2006; Chen et al. 2007] or Web applications [Huang et al. 2004; Kals et al. 2006], attackers may directly harm a Web browser's host machine and user through various attacks such as drive-by download [Moshchuk et al. 2006; Wang et al. 2006; Provos et al. 2008; Egele et al. 2009; Cova et al. 2010; Canali et al. 2011], cross-site scripting [CERT 2000; Fogie et al. 2007], cross-site request forgery [SANS 2007; Barth et al. 2008a], phishing [Jakobsson and Myers 2006; Dhamija et al. 2006; Yue 2012; Yue and Wang 2010], password cracking or stealing [Florêncio and Herley 2007; Stone-Gross et al. 2009; Komanduri et al. 2011; Zhao and Yue 2013], and Web privacy attacks [Jackson et al. 2006; Bortz et al. 2007]. Attackers may even use browsers to indirectly launch large-scale distributed attacks against Web servers [Lam et al. 2006] or propagate Internet worms [Livshits and Cui 2008].

Most of these browser-based attacks are closely tied with JavaScript, which is an interpreted programming language most often used for client-side scripting. JavaScript code embedded or included in HTML pages runs locally in a user's Web browser, and is mainly used by websites to enhance the interactivity and functionality of their webpages. However, because JavaScript is equipped with a powerful and diverse set of capabilities in Web browsers [Flanagan 2006], it has also become the weapon of choice for attackers.

Modern Web browsers impose two restrictions to enforce JavaScript security: the *sandbox* mechanism and the *same-origin* policy. The former limits JavaScript to execute only in a certain environment without risking damage to the rest of the system, while the latter prevents JavaScript in a document of one origin from interacting with another document of a different origin [Flanagan 2006; WikiSOP 2011]. Unfortunately, most JavaScript-related security vulnerabilities are still the breaches of either of these two restrictions [WikiJS 2011]. Some of these vulnerabilities are due to Web browser flaws, but the majority of them have been attributed to the flaws and insecure practices of websites [SANS 2007; Symantec 2008].

A great deal of attention has been paid to the JavaScript-related security vulnerabilities such as cross-site scripting [Fogie et al. 2007; SANS 2007; Vogt et al. 2007; Wassermann and Su 2008; Symantec 2008; Kirda et al. 2009; Hooimeijer et al. 2011; WikiXSS 2011] that could directly lead to security breaches. However, little attention has been given to websites' insecure practices of using JavaScript on their webpages. Similar to websites' other insecure practices such as using the customers' social security numbers as their login IDs [Falk et al. 2008], insecure JavaScript practices may not necessarily result in direct security breaches, but they could definitely cultivate the creation of new attack vectors.

In this work, we present the first measurement study on insecure practices of using JavaScript at different websites. We mainly focus on two types of insecure practices: *insecure JavaScript inclusion* and *insecure JavaScript dynamic generation*. We define the first type of practices as using the `src` attribute of a <script> tag to directly or indirectly include a JavaScript file from an external domain into the *top-level document* of a webpage. A top-level document is the document loaded from the URL displayed in a Web browser's address bar. By "directly", we mean that the <script> tag belongs to the top-level document, and by "indirectly", we mean that the <script> tag belongs to a sublevel frame or iframe document whose origin is the same as that of the top-level document. Either directly or indirectly, the included scripts will inherit the origin of

the top-level document and will obtain maximum permissions to control the top-level document and the browser window; therefore, both cases should be considered in our measurement to ensure the completeness of our analysis. We define the second type of practices as using dangerous techniques such as the eval() function to dynamically generate new scripts. Both types of insecure practices create new vectors for attackers to inject malicious JavaScript code into webpages and launch attacks such as cross-site scripting and cross-site request forgery.

The primary objective of our work is to examine the severity and nature of these two types of insecure JavaScript practices on the Web. To achieve this goal, we devised an execution-based measurement approach. More specifically, we instrumented the Mozilla Firefox 2 Web browser and visited the homepages of 6,805 popular websites in 15 different categories. The instrumented Firefox nonintrusively monitors the JavaScript inclusion and dynamic generation activities on these webpages, and it precisely records important information for offline analysis.

Our measurement results reveal that insecure JavaScript inclusion and dynamic generation practices are widely prevalent among websites. At least 66.4% of the measured websites have the insecure practices of including scripts from external domains into the top-level documents of their homepages. Over 74.9% of the measured websites use one or more types of JavaScript dynamic generation techniques, and insecure practices are quite common. For example, eval() function calls exist at 44.4% of the measured websites. Using the document.write() method and the innerHTML property is much more popular than using the relatively secure method of creating JavaScript elements via DOM (Document Object Model) [W3CDOM 2011] methods. Our results also show that around 94.9% of the measured websites register various event handlers on their homepages, implying that the captured insecure JavaScript practices in inclusion and dynamic generation are likely conservative estimates.

The main contribution of our work is threefold. First, we introduce a browser instrumentation framework that enables us to capture essential JavaScript execution behavior on webpages. Not only can this framework measure the insecure JavaScript practices, it can also examine other JavaScript execution characteristics such as function call patterns and code (de)obfuscation activities. Second, we present a classification method to analyze and classify different types of dynamically generated JavaScript code. By extracting the AST (Abstract Syntax Tree) trees of scripts and performing AST signature creation and matching, our classification method can effectively assist us in understanding the structural information of the hundreds of thousands of dynamically generated scripts. Third, our measurement study sheds light on the insecure JavaScript practices and especially reveals the severity of insecure JavaScript inclusion and dynamic generation practices on the Web. Our in-depth analysis further indicates that safe alternatives to these insecure practices do exist in common cases. We therefore suggest website developers and administrators pay serious attention to these insecure engineering practices and use safe alternatives to avoid them.

The remainder of this article is structured as follows. Section 2 explains why the two types of JavaScript practices are insecure. Section 3 introduces our measurement and analysis methodologies. Section 4 describes the dataset of this study. Section 5 presents and analyzes our measurement results. Section 6 reviews related work, and finally, Section 7 concludes the article.

2. BACKGROUND

In the same-origin policy, the origin of a document is defined using the protocol, domain name, and port of the URL from which the document is loaded. It is important to realize that this policy does not limit the origin of a script itself. Although JavaScript code cannot access another document loaded from a different origin, it can fully access the

document in which it is embedded or included even when the code has a different origin than the document [Flanagan 2006]. Including scripts from an external domain into the top-level document of a webpage is very dangerous because it grants the scripts the maximum permissions allowed to control the webpage and the browser window. Therefore, if the author of a script file or the administrator of a script hosting site is insincere or irresponsible, insecure JavaScript inclusion practices could lead to serious security and privacy breaches. Moreover, script hosting sites could become attractive targets of attacks, especially when their JavaScript files are included by multiple websites. To lower the potential risks, websites should avoid external JavaScript inclusion by using internal JavaScript files from the same sites when possible. Otherwise if external inclusion is really inevitable, for example, some advertising sites or traffic analysis sites may necessitate it [Oda et al. 2008], external included scripts should be retrieved using HTTPS connections and should be restricted within a sublevel HTML frame or iframe document whose origin is different from that of the top-level document.

The eval() function takes a string parameter and evaluates it as JavaScript code. This function is dangerous because it executes the passed script code with the privileges of the function's caller [EvalMDC 2011]. Therefore, attackers may endeavor to inject malicious code into the evaluated string in order to take advantage of this capability. Meanwhile, since scripts are dynamically generated and evaluated, it is very challenging to effectively filter out maliciously injected code [Jim et al. 2007; Reis et al. 2006; Yu et al. 2007]. Eval() should be avoided[1] if at all possible, and its safe alternatives should be used [Willison 2005; EvalMDC 2011]. Other JavaScript dynamic generation techniques such as using the document.write() function and the innerHTML property also pose similar security risks, as discussed in Section 5.

Once attackers have successfully exploited these insecure practices and injected their malicious JavaScript code, they can easily launch severe attacks such as cross-site scripting and cross-site request forgery. These attacks can be used to conduct many malicious activities such as account hijacking, user behavior tracking, denial-of-service attacking, and website defacing. Therefore, insecure engineering practices of using JavaScript should be thoroughly investigated, their risks should be highlighted to Web developers, and safe alternatives should be used to avoid them.

3. METHODOLOGY

We devised an execution-based measurement approach to study the insecure JavaScript practices on the Web. Our strategy is to first use an instrumented Web browser to obtain actual JavaScript execution trace information on different webpages, and then use offline analysis to characterize and understand various JavaScript practices. An alternative approach is to simply perform static analysis on webpages. However, this approach suffers the problem of undecidability and is unable to precisely determine which scripts will be generated and executed. In contrast, our approach allows us to effectively capture the dynamics of webpages and JavaScript code in their real runtime environments. Figure 1 gives an overview of our instrumentation framework and analysis toolkit.

3.1. Instrumentation Framework

To achieve an accurate and efficient measurement, we employed the source-code instrumentation technique and instrumented the most popular open-source Web browser, Mozilla Firefox. Our instrumentation method is similar to program tracing, which is a well-known approach for monitoring program behavior and measuring program performance. We followed a few rules suggested in Ball and Larus [1994] to minimize

[1]Searching "eval is evil" on the Web for many discussions.

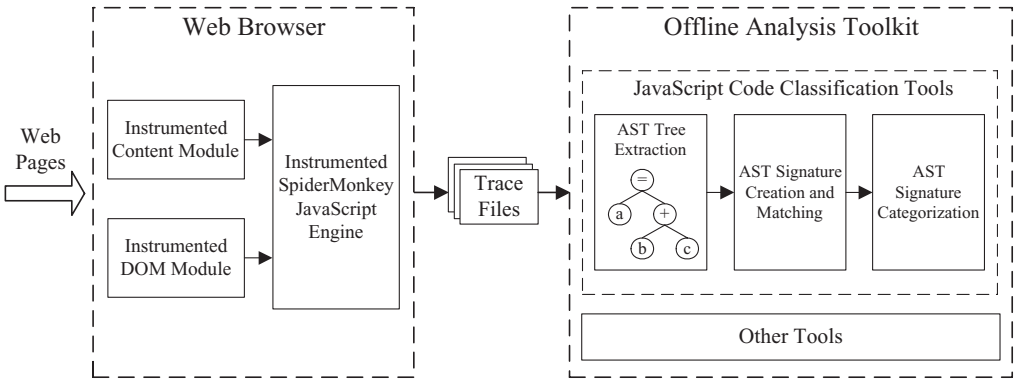

Fig. 1. Overview of the instrumentation framework and analysis toolkit.

instrumentation overhead. More specifically, we attempted to insert less instrumentation code and place the code only at necessary points with low execution frequency. In terms of "less instrumentation code", we mean that the code will be mainly used to monitor the JavaScript execution and record the essential information to trace files; no or the least number of computation or analysis tasks should be performed by the instrumented code. In terms of "necessary points with low execution frequency", we mean that the instrumented code should not be unnecessarily executed; for example, although instrumenting low-level functions in the JavaScript engine can reduce the total number of instrumentation points, it often incurs more overhead because low-level functions are executed more frequently than high-level functions.

We mainly instrumented three modules of Firefox 2 source code: the JavaScript engine, the content module, and the DOM module. Firefox uses SpiderMonkey as its JavaScript engine [SpiderMonkey 2012]. SpiderMonkey JavaScript engine is written in C programming language and is a relatively independent module in Firefox. The major interface between SpiderMonkey and other modules in Firefox is the SpiderMonkey JSAPI [JSAPI 2011]. JSAPI facilitates other modules in Firefox to use the core JavaScript data types and functions of SpiderMonkey, and it also allows other modules to expose some of their objects and functions to JavaScript code.

Inside the SpiderMonkey, our instrumented code written in C consists of three parts. First, eight trace logging functions were integrated into the JSAPI interface. These functions facilitate the trace collection in a consistent manner, recording various information such as script text, function calls, and event handler registrations. Second, we added code to the byte-code interpreter of SpiderMonkey so that we can record the execution information of any global scripts and function scripts. Third, we instrumented the object system implementation of SpiderMonkey to monitor the calls to the eval() function and collect both the calling context information and the evaluated content information. The trace files generated in the preceding instrumentation points enable us to analyze the practices of JavaScript inclusion and the practices of JavaScript dynamic generation using eval().

We also need to monitor the practices of other JavaScript dynamic generation techniques. Originally we attempted to fulfill this task by still instrumenting inside the SpiderMonkey and monitoring the engine's native callbacks to the content and DOM modules. However, we found that this approach incurs high overhead, could easily introduce errors in instrumentation, and could only record partial information. For example, the js_Interpret() function in the jsinterp.c of SpiderMonkey is the essential function where JavaScript interpretation happens. We tried to instrument this

function to record other JavaScript dynamic generation techniques; however, this low-level function uses threaded interpretation via computed *goto* statements to achieve high performance and is very complex. Instrumenting this function can incur seconds of overhead on many webpages. Meanwhile, the complexity of its interrupt jump table and lots of *goto* statements make it very difficult for us to properly add the code to monitor all of our interested information. Therefore, we decided to directly instrument the content module and the DOM module of Firefox to monitor the practices of other JavaScript dynamic generation techniques. To identify and instrument all the entry points from other modules to SpiderMonkey, we carefully inspected all the major source-code modules of Firefox 2, and we also used the Mozilla cross-reference tool [MXR 2012] for Firefox 2 and our own source-code search tool.

In the content module of Firefox, we integrated C++ code to measure the other three types of JavaScript dynamic generation techniques. We instrumented the document.write() method[2] and the method for setting the innerHTML property of an HTML element to track their invocations. Both techniques can be used to add new content to an HTML document, and the added content may contain new JavaScript code. We also added code to monitor the method for replacing, inserting, or appending a new DOM element, which could be created by using DOM methods such as document.createElement() and document.createTextNode(). Our instrumentation code can identify the script type of elements and record their source and text information. Other techniques such as the insertAdjacentHTML() method or the outerHTML property are supported in the Internet Explorer Web browser only, and we cannot measure them in Firefox.

In the DOM module and the content module, we added C++ code to measure various event handler registration techniques supported in Firefox. Event handlers can be triggered by user interaction or timer events. We collected event handler registration information to show that further JavaScript inclusion and execution could happen and our captured insecure practices are likely conservative estimates. Event handler registration and other aspects of information described before are written into a set of six different trace files to assist our offline analysis.

Since many internal user interface components of Firefox also heavily use JavaScript, special care is needed to ensure that the preceding instrumentation code only records the JavaScript execution activities of a visited webpage. Our code checks the JSPrincipals [2011] information of an object or script to guarantee this requirement. We also ensured that our instrumentation code only monitors and records essential information and does not change the execution logic of Firefox and SpiderMonkey. Our instrumentation is specific to the Firefox 2 Web browser and its SpiderMonkey JavaScript engine. If we want to achieve the same instrumentation objective in other browsers and their JavaScript engines such as Google Chrome and its V8 JavaScript engine, we would like to take a very similar approach as what we did in this work. That is, we will first examine whether the whole instrumentation can be purely performed in the corresponding JavaScript engine; if a pure engine-based instrumentation is not practical, we should only perform those practical instrumentation tasks in the JavaScript engine, and should identify and perform those impractical instrumentation tasks in other relevant modules of the browser.

3.2. Analysis Toolkit

We took an offline analysis approach so that we can sufficiently analyze the trace information without interfering with the actual measurement process. We developed an offline analysis toolkit that consists of a set of tools written in approximately 9,000

[2]In this work, it also includes the document.writeln() method.

```
                        new_off.src = dir + "new_off.gif";
                                    (a)

document.getElementById("noticia").innerHTML = NoticiaStr[posNot] +
  " <a href=" + LigaNotStr[posNot] + " class=LigaNN>(Ir al comunicado)</a>";
                                    (b)

                    document.getElementById("num_img" + imgNo);
                                    (c)

                       loadBanner(http_request.responseXML);
                                    (d)
```

Fig. 2. Four real JavaScript examples: (a) a simple assignment statement; (b) a complex assignment statement; (c) a method call; (d) a function call.

lines of Java code, 200 lines of C code, 500 lines of Linux shell script code, and 300 lines of Matlab script code. Most of the tools are used for classifying dynamically generated JavaScript code, and the others are used for processing trace records and calculating statistical information. The detailed description of the JavaScript code classification tools is as follows.

The motivation for developing these classification tools is to automate the challenging task of understanding a large amount of dynamically generated JavaScript code. To achieve this goal, we explored the concepts in software engineering and developed an AST-based classification method. As illustrated in Figure 1, the key idea is to first extract the AST trees of scripts, then create and match AST signatures, and finally merge signatures into different categories. We devised such an AST-based approach in that ASTs have been demonstrated effective in program understanding [Baxter et al. 1998; Welty 1997].

The AST tree extraction tool is a stand-alone C program that embeds the SpiderMonkey 1.7 [SpiderMonkey 2012]. This is the same version of the SpiderMonkey as used in our instrumented Firefox 2 Web browser. Therefore, our extraction tool can create a token stream and parse the stream into a syntax tree for a script in the same manner as in the instrumented Firefox. The tool finally constructs the essential structure of a syntax tree as an AST tree and writes the tree into an XML file to facilitate further comparison.

To implement this AST tree extraction tool, we partially ported the source-code provided in the js.c, jsapi.c, jsscan.h, jsscan.c, jsparse.h, and jsparse.c of SpiderMonkey 1.7 [SpiderMonkey 2012] and an example provided at siliconforks.com [SiliconForks 2012]. In more details, 87 token types are defined in the jsscan.h of SpiderMonkey 1.7. These token types represent different operators, keywords, operand types, statement separators or terminators, and special tags, etc., used in the JavaScript programming language. A dynamically generated script (could be a list of JavaScript statements) will be parsed into a syntax tree, and each node in the tree is a JSParseNode structure defined in the jsparse.h. A constructed AST tree has the same structure as the corresponding syntax tree, but each of its nodes is simply represented by the token type value and some other associated properties of the corresponding JSParseNode.

Figure 2 illustrates four real JavaScript examples (a), (b), (c), and (d) recorded into our trace files. Figure 3 illustrates the four real AST trees (a), (b), (c), and (d) extracted from the corresponding JavaScript examples in Figure 2. These four extracted AST trees were written into XML files.

In these four AST tree examples shown in Figure 3, TOK_* XML tags are the token types defined in the jsscan.h. TOK_SEMI represents the semicolon statement terminator ";"; TOK_ASSIGN represents any assignment operator such as "=", "+=", or

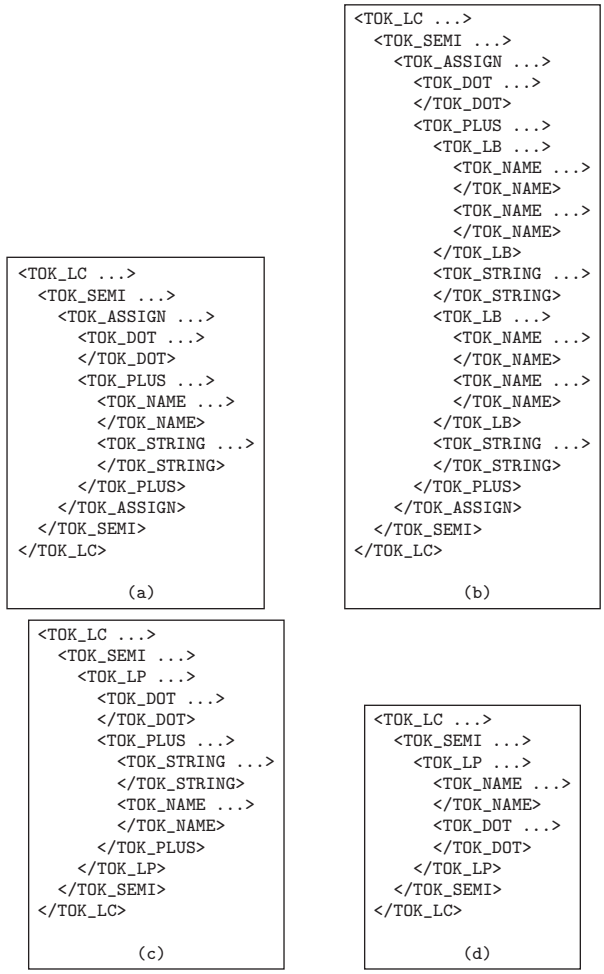

Fig. 3. Four real AST trees (a), (b), (c), and (d) extracted from the corresponding JavaScript examples in Figure 2. Their heights (visualized by their maximum indentation levels) are 5, 6, 5, and 4, respectively.

"−="; TOK_DOT represents the member operator "."; TOK_PLUS represents the addition operator "+"; TOK_NAME represents an identifier; TOK_STRING represents a string constant; TOK_LB represents the left bracket "["; TOK_LP represents the left parenthesis "(". The root of these AST trees is an element defined by the TOK_LC tag, which is the left curly brace "{" and is a special token type used by SpiderMonkey to represent a list of statements. This root is common to all the extracted AST trees. The "..." represents other properties of tokens such as their line and column numbers in the original JavaScript code; these properties can be used to manually compare an AST tree with its original JavaScript code for verifying the correctness of the AST extraction. The heights of the four AST trees are 5, 6, 5, and 4, respectively.

We applied *top-down tree matching* techniques to perform AST signature creation and matching, and the high-level procedure is illustrated in Figure 4. First, an empty AST signature set S is initialized. Next, for each AST tree in the XML files, its top N-level structure is used to generate an AST signature, denoted as *thisSig*. Then, top-down tree comparisons are made to seek a match between the *thisSig* and an existing

```
SigCreateMatch (XMLfiles, N)
1. Initialize an empty AST signature set S;
2. for each AST tree in the XML files do
3.    thisSig=the top N level structure of the AST tree;
4.    if thisSig matches an existing signature in S then
5.       Record the information of this matching;
6.    else
7.       S = S ∪ {thisSig};
8.    endif
9. endfor
10. return the result set S;
```

Fig. 4. High-level AST signature creation and matching procedure.

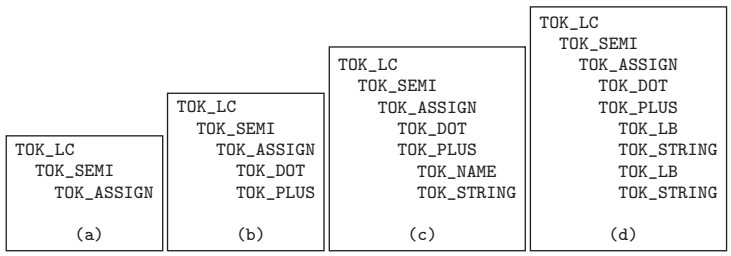

Fig. 5. AST signature examples based on the first two AST trees (a) and (b) in Figure 3: (a) is based on the top three-level AST trees (a) and (b) in Figure 3; (b) is based on the top four-level AST trees (a) and (b) in Figure 3; (c) is based on the top five-level AST tree (a) in Figure 3; (d) is based on the top five-level AST tree (b) in Figure 3.

signature in the set S. If a match exists, this procedure keeps a record of the related information; otherwise, the *thisSig* is added to the set S as a new AST signature. Finally, this procedure returns the signature set S as its output.

To be accurate and representative, an AST signature only keeps the type information of an operator node, and it also only keeps the type information of an operand node. Other associated properties such as line and column numbers of tokens are also discarded in an AST signature. Top-down tree matching techniques can capture the key structural differences between trees, and they have been used in several Web-related projects [Reis et al. 2004; Yue et al. 2010; Zhai and Liu 2005]. The signature comparison algorithm used in line 4 of this procedure is adapted from the STM (Simple Tree Matching) algorithm presented in Yang [1991]. STM is an efficient top-down tree distance comparison algorithm, and our adaptation is to only compare the top N levels of trees. Our adapted algorithm is similar to the RSTM algorithm presented in Figure 9 of Yue et al. [2010].

Such a top N-level adaptation is effective in striking a good balance between retaining the accuracy and reducing the total number of signatures. Figure 5 illustrates four AST signature examples based on the first two AST trees (a) and (b) in Figure 3. Figure 6 illustrates four AST signature examples based on the last two AST trees (c) and (d) in Figure 3.

On the one hand, if the input parameter N for the AST signature creation and matching procedure (Figure 4) is too large, then the same category of JavaScript code may not be directly summarized by the same AST signature. For example, if the top three levels of AST trees are compared (i.e., $N = 3$), then the same AST signature shown in Figure 5(a) will be created or matched for the two AST trees (a) and (b) in

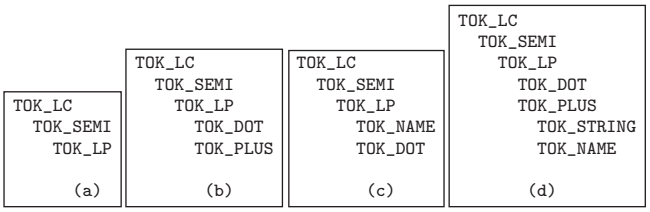

Fig. 6. AST signature examples based on the last two AST trees (c) and (d) in Figure 3: (a) is based on the top three-level AST trees (c) and (d) in Figure 3; (b) is based on the top four-level AST tree (c) in Figure 3; (c) is based on the top four-level AST tree (d) in Figure 3; (d) is based on the top five-level AST tree (c) in Figure 3.

Figure 3. Similarly, if the top four levels of AST trees are compared, then the same AST signature shown in Figure 5(b) will be created or matched for the two AST trees (a) and (b) in Figure 3. However, if the top five levels of AST trees are compared, then the AST signature shown in Figure 5(c) will be created or matched for the AST tree (a) in Figure 3, and a different AST signature shown in Figure 5(d) will be created or matched for the AST tree (b) in Figure 3. In such a case, the two assignment statements ((a) and (b) in Figure 2) cannot be directly summarized by the same AST signature.

On the other hand, if the input parameter N for the AST signature creation and matching procedure (Figure 4) is too small, then different categories of JavaScript code may not be directly differentiated by distinct AST signatures. Although the AST signature in Figure 5(a) ($N = 3$) can still well represent the two assignment statement AST trees (a) and (b) in Figure 3, a larger N value should be used in some other cases. For example, if the top three levels of AST trees are compared (i.e., $N = 3$), then the same AST signature shown in Figure 6(a) will be created or matched for the two AST trees (c) and (d) in Figure 3. However, these two AST trees represent different categories of JavaScript code: one is a method call and the other is a function call ((c) and (d) in Figure 2). If the top four levels of AST trees are compared, then the AST signature shown in Figure 6(b) will be created or matched for the AST tree (c) in Figure 3, and a different AST signature shown in Figure 6(c) will be created or matched for the AST tree (d) in Figure 3. In Figure 6(b), the first child of TOK_LP is TOK_DOT, which means the left side of the left parenthesis "(" is a method name; in Figure 6(c), the first child of TOK_LP is TOK_NAME, which means the left side of the left parenthesis "(" is a function name. Therefore, with $N = 4$, these two AST trees can be properly summarized by distinct AST signatures. If $N = 5$, the AST signature shown in Figure 6(d) will be created or matched for the AST tree (c) in Figure 3, but using $N = 5$ is not necessary because we do not want to further consider the input to the method call.

Different selections of the input parameter N for the AST signature creation and matching procedure will also result in different total numbers of the AST signatures. Examples will be provided in Section 5.3.3. In addition, because our AST tree extraction tool is implemented based on the SpiderMonkey 1.7, the selection of the input parameter N is closely related to this specific JavaScript engine as illustrated by the aforementioned examples. However, the basic principle should be the same; that is, N should not be too large as to summarize many AST trees of the same category using different AST signatures, and it should not be too small as to summarize many AST trees of different categories using the same AST signature.

The AST signature categorization tool was developed to further merge AST signatures into different categories. We simply defined categories according to different types of JavaScript expressions and statements such as arithmetic expressions and assignment statements. Such a categorization can help us understand the use purposes

Table I. Category Breakdown by Top-Level Domain

Category	com	org	gov	net	edu	cc	other	Total
arts	417	16	0	27	1	39	0	500
business	430	7	10	4	0	49	0	500
computers	432	29	1	21	1	15	1	500
games	428	13	0	43	0	14	2	500
health	277	107	41	8	33	30	4	500
home	415	28	22	14	2	18	1	500
news	412	24	6	12	3	43	0	500
recreation	409	19	12	19	0	40	1	500
reference	116	17	11	4	192	158	2	500
regional	292	23	21	6	3	152	3	500
science	209	96	68	8	47	64	8	500
shopping	479	2	0	2	0	17	0	500
society	302	84	34	11	3	58	8	500
sports	403	13	0	21	0	62	1	500
world	199	15	1	23	0	262	0	500
Total	5220	493	227	223	285	1021	31	7500
Uniq-Total	4727	445	170	212	276	950	25	6805

of JavaScript code from a programming language perspective. This tool is especially useful for analyzing dynamically generated scripts, most of which have specific use purposes in terms of programming language functionality as revealed in Section 5.3.

4. DATASET

To obtain a representative dataset, we followed a similar method as used in Krishnamurthy and Wills [2006] and selected top websites listed by Alexa.com. We chose 15 categories and then top 500 sites from each of these categories. Table I gives the breakdown of 15 categories by DNS Top-Level Domain (TLD). JavaScript is often used for enhancing webpage interactivity and functionality. Different types of websites could be developed by different types of developers and could have different interactivity and functionality requirements. Considering sites from different categories and TLDs allows us to measure whether the prevalence of insecure JavaScript practices changes with the types of websites. This website selection method is also similar to the one used in Krishnamurthy and Wills [2006]. Because some sites appear in multiple categories, the total number of unique sites is 6,805 in our study. This number is over five times larger than that in Krishnamurthy and Wills [2006], and we also only visited the homepages of those sites so that we can have a consistent measurement. Meanwhile, measuring the insecure JavaScript practices on homepages is sufficient to illustrate the severity of the problem. Table I shows that the majority of the 6,805 sites come from the .com TLD and the country code (denoted as the cc) TLD. The former contributes 4,727 unique sites and the latter contributes 950 unique sites.

The execution of JavaScript on a webpage can be roughly divided into two phases: the document loading and parsing phase and the event-driven phase [Flanagan 2006]. When the document loading and parsing phase ends, the event-driven phase starts and event handlers can be asynchronously executed in response to various user interaction and timer events. In our study, we developed a browser extension to automatically visit each of the 6,805 webpages using our instrumented Firefox Web browser. On each page, our browser extension waits for the end of the document loading and parsing phase and then stays in the event-driven phase for 10 seconds. Our browser extension has no intention to trigger the execution of any specific event handlers on a page. In other words, we did not intentionally click buttons, fill in forms, or do any specific actions

Table II. JavaScript Presence by Category and Top-Level Domain

Category/ TLD	Pages with any JS			Pages with DJS
	embedded JS	included JS	Total	
arts	484(96.8%)	483(96.6%)	491(98.2%)	437(87.4%)
business	482(96.4%)	473(94.6%)	492(98.4%)	380(76.0%)
computers	471(94.2%)	465(93.0%)	484(96.8%)	374(74.8%)
games	471(94.2%)	473(94.6%)	488(97.6%)	375(75.0%)
health	467(93.4%)	451(90.2%)	481(96.2%)	330(66.0%)
home	479(95.8%)	471(94.2%)	487(97.4%)	389(77.8%)
news	477(95.4%)	475(95.0%)	483(96.6%)	430(86.0%)
recreation	477(95.4%)	467(93.4%)	487(97.4%)	389(77.8%)
reference	455(91.0%)	443(88.6%)	476(95.2%)	286(57.2%)
regional	479(95.8%)	457(91.4%)	492(98.4%)	401(80.2%)
science	421(84.2%)	405(81.0%)	449(89.8%)	274(54.8%)
shopping	487(97.4%)	486(97.2%)	493(98.6%)	393(78.6%)
society	441(88.2%)	435(87.0%)	466(93.2%)	329(65.8%)
sports	492(98.4%)	482(96.4%)	496(99.2%)	456(91.2%)
world	481(96.2%)	438(87.6%)	489(97.8%)	377(75.4%)
com	4551(96.3%)	4504(95.3%)	4629(97.9%)	3838(81.2%)
org	401(90.1%)	378(84.9%)	422(94.8%)	247(55.5%)
gov	150(88.2%)	137(80.6%)	160(94.1%)	75(44.1%)
net	194(91.5%)	189(89.2%)	204(96.2%)	153(72.2%)
edu	239(86.6%)	223(80.8%)	250(90.6%)	122(44.2%)
cc	863(90.8%)	817(86.0%)	902(94.9%)	654(68.8%)
other	23(92.0%)	22(88.0%)	24(96.0%)	9(36.0%)
All	6421(94.4%)	6270(92.1%)	6591(96.9%)	5098(74.9%)

on a page. This is because the event handlers registered on different webpages could be very diverse, and it is difficult to trigger their executions in a consistent manner on a large number of webpages. Therefore, the JavaScript execution dataset collected in our measurement study covers the whole document loading and parsing phase and 10 seconds of the event-driven phase for each of the 6,805 homepages. The dataset was collected in the second week of July 2008.

5. RESULTS AND ANALYSIS

We present and analyze our measurement results in this section. We first briefly present the results on JavaScript presence. Second, we detail the results on the insecure practices of JavaScript inclusion and dynamic generation. Third, we summarize the results on event handler registrations. Finally, we discuss the limitations of our measurement study and their implications.

5.1. Overall JavaScript Presence

Table II lists the results of overall JavaScript presence for the 6,805 measured homepages. We use *JS* to represent any JavaScript code, and we use *DJS* to represent the JavaScript code that is dynamically generated by using one of the four dynamic generation techniques measured in our instrumented Firefox Web browser. The *embedded* JS indicates that the executed JavaScript code is embedded within an HTML document, and the *included* JS indicates that the executed JavaScript code is included from a separate file.

Overall, JavaScript execution has been widely observed on 6,591(96.9%) homepages. Both the JS embedding and JS inclusion are very common, and they are practiced on 6,421 and 6,270 pages, respectively. For the percentage of webpages containing JavaScript execution within a category, the lowest percentage is 89.8% for science, the

highest percentage is 99.2% for sports, and the range is 9.4%. For the percentage of webpages containing JavaScript execution within a TLD, the lowest percentage is 90.6% for .edu, the highest percentage is 97.9% for .com, and the range is 7.3%. JavaScript dynamic generation is also very popular, and there are 5,098 (74.9%) sites containing DJS on their homepages. For the DJS presence within a category, the lowest percentage is 54.8% for science, the highest percentage is 91.2% for sports, and the range is 36.4%. For the DJS presence within a TLD, the lowest percentage is 36.0% for `other` domains such as .mil and .info, the highest percentage is 81.2% for .com, and the range is 45.2%. It is interesting to note that the range of DJS presence within a category is much larger than the range of JS presence within a category (36.4% versus 9.4%); meanwhile, the range of DJS presence within a TLD is also much larger than the range of JS presence within a TLD (45.2% versus 7.3%). These results demonstrate that the prevalence of JavaScript dynamic generation practices indeed changes with the types of websites. We can also see that the prevalence of included JS does not change dramatically with the types of websites. For the included JS presence within a category, the lowest percentage is 81.0% for science, the highest percentage is 97.2% for shopping, and the range is 16.2%. For the included JS presence within a TLD, the lowest percentage is 80.6% for .gov, the highest percentage is 95.3% for .com, and the range is 14.7%.

5.2. Insecure JavaScript Inclusion

Among all the 6,270 webpages with the included JS, we identify and analyze insecure practices of JavaScript inclusion. Note that we defined the insecure JavaScript inclusion as the practices of using the `src` attribute of a <script> tag to directly or indirectly include a JavaScript file from an external domain into the top-level document of a webpage. Keeping JavaScript code separate from HTML markups is actually a good engineering practice, advocated especially in the unobtrusive JavaScript programming paradigm [Flanagan 2006; Heilmann 2011]. Therefore, there is no need to analyze the good practices of including JavaScript files from the same host or domain, and we only focus on the insecure inclusion practices.

5.2.1. Results and Analysis. To our surprise, insecure JavaScript inclusion is very prevalent. Around 66.4% (4,517 out of 6,805) of websites directly or indirectly include JavaScript files from external domains into the top-level documents of their homepages. Note that our analysis tool applies a conservative standard to compare the domain name of a JavaScript file and that of its including homepage. Two domain names are regarded as different only if, after discarding their top-level domain names (e.g., .com) and the leading name "www" (if existing), they do not have any common subdomain name[3]. Therefore, this 66.4% result is basically an objective estimate of the severity of insecure JavaScript inclusion practices. We provide a simple manual verification of the effectiveness of this domain name comparison method at the end of this subsection.

After further analyzing the domain name relationship between JavaScript file inclusion sites and JavaScript file hosting sites, we found that those 4,517 sites include JavaScript files from a diverse set of 1,985 external domains. We can use a directed graph to characterize the domain name relationship between these sites. Different vertices represent different domain names, and a direct edge from vertex A to vertex B means that the homepage in domain A includes at least one JavaScript file from domain B. Therefore, 4,517 vertices have a greater than zero outdegree value, and 1,985 vertices have a greater than zero indegree value.

[3]For example, two domain names www.d1sub2.d1sub1.d1tld and d2sub3.d2sub2.d2sub1.d2tld are regarded as different only if the intersection of the two sets {d1sub2, d1sub1} and {d2sub3, d2sub2, d2sub1} is empty.

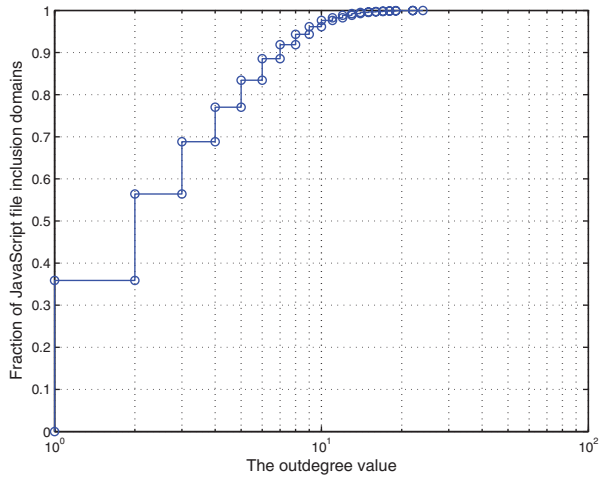

Fig. 7. Cumulative distribution of the 4,517 JavaScript file inclusion domains in terms of their outdegree values.

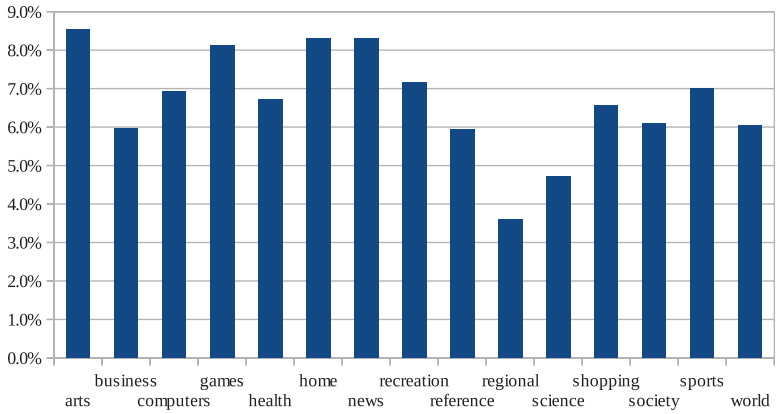

Fig. 8. Distribution of the 4,517 JavaScript file inclusion domains in terms of their categories.

Figure 7 illustrates the CDF (Cumulative Distribution Function) of the 4,517 JavaScript file inclusion domains in terms of their outdegree values. We can see that approximately 43.6% of the 4,517 sites include JavaScript files from at least three external domains. While the mean value of outdegree is 3.1, the maximum value of outdegree reaches 24. These results indicate that not only 66.4% of measured sites are at the risk of having their homepages under the control of the included JavaScript code, but many of them also face higher risks from multiple sources.

Figure 8 further illustrates the distribution of the 4,517 JavaScript file inclusion domains in terms of their categories. We can see that only 3.6% and 4.7% of the 4,517 sites belong to the regional and science categories, respectively; most of the 4,517 sites belong to other categories. Figure 9 further illustrates the distribution of the 4,517 JavaScript file inclusion domains in terms of their TLDs. From those absolute bars (the ratio between the number of JavaScript file inclusion domains within a TLD and 4,517), we can see that the majority (76.0%) of the 4,517 sites belong to the .com TLD. This is mainly because the majority of the measured websites as shown in Table I belong to the .com TLD. From those relative bars (the ratio between the number of JavaScript file inclusion domains within a TLD and the total number of unique sites

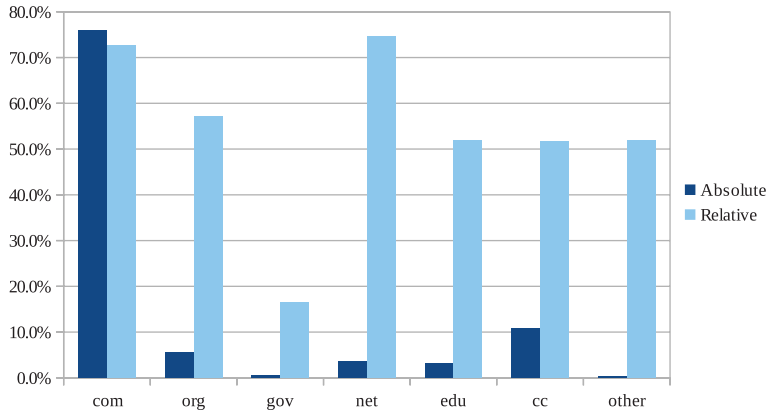

Fig. 9. Distribution of the 4,517 JavaScript file inclusion domains in terms of their TLDs.

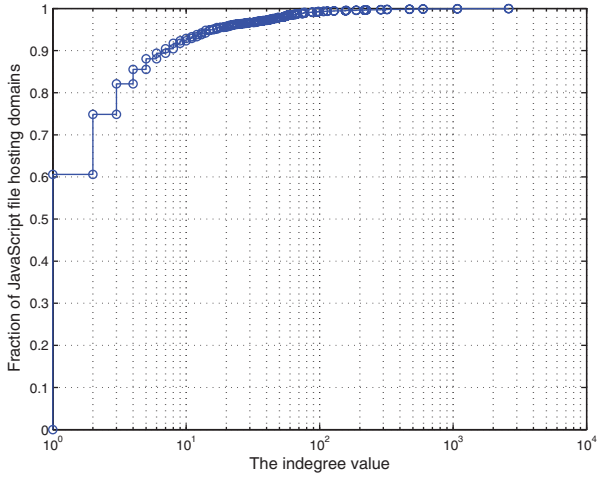

Fig. 10. Cumulative distribution of the 1,985 JavaScript file hosting domains in terms of their indegree values.

within a TLD as shown in the last row of Table I), we can see that insecure JavaScript inclusion practices appeared on 74.5% of the .net sites and 72.6% of the .com sites, but only appeared on 16.5% of the .gov sites.

From a different perspective, Figure 10 depicts the CDF of the 1,985 JavaScript file hosting domains in terms of their indegree values. We can observe two interesting phenomena. On the one hand, JavaScript files in approximately 60.6% of the hosting domains are only included by one of our visited homepages. On the other hand, JavaScript files in approximately 7.7% of the hosting domains are included by at least 10 of our visited homepages, and JavaScript files in 14 sites are even included by at least 100 of our visited homepages. The mean value of indegree is 7.2, but the maximum value of indegree reaches a very high value of 2,606. What we need to emphasize is that external JavaScript file hosting sites, especially those high-profile ones, create new vectors for large-scale browser-based attacks. Even a single compromised JavaScript file could directly cause security breaches on thousands of websites.

We further inspected these high-profile JavaScript file hosting domains and many other low-profile domains, and we found that a few of them are popular traffic analysis service sites and advertising servers. However, most of them are the type of “hidden”

Table III. Twenty High-Profile JavaScript File Hosting Domains with the Largest Indegree Values

Hosting Domains	Indegree Value	Meaningful information on root URL?	Example URL paths and names of the included JavaScript files
www.google-analytics.com	2606	Yes	urchin.js, ga.js, siteopt.js
pagead2.googlesyndication.com	1076	No	pagead/show_ads.js, pagead/ads?***
ad.doubleclick.net	596	Yes	adj/***, click***
edge.quantserve.com	471	Yes	quant.js
m1.2mdn.net	320	No	***/flashwrite_1_2.js, ***/MotifExternalScript_01_01.js, ***/DartRichMedia_1_03.js, ***/EventBin_01_17.js, ***/WMPlayer_01_17.js
an.tacoda.net	287	No	an/12426/slf.js
js.revsci.net	224	No	gateway/gw.js, common/pcx.js
anrtx.tacoda.net	216	No	opt/r.js, dastat/ping.js
a.tribalfusion.com	188	No	j.ad?***, h.click/***
tags.expo9.exponential.com	158	No	tags/GamingSource/ROS/tags.js
www.google.com	156	Yes	coop/cse/brand?***, jsapi, uds/api/search/***/default.I.js, cse/api/overlay.js, uds/Gfeeds?***, uds/?***
ssl.google-analytics.com	128	Yes	urchin.js, ga.js, siteopt.js
partner.googleadservices.com	114	No	gampad/google_service.js, gampad/google_ads.js, gampad/slotdata.js, gampad/cookie.js, gampad/ads?***
statse.webtrendslive.com	107	No	***/wtid.js
www.statcounter.com	98	Yes	counter/counter.js, counter/counter_xhtml.js, counter/frames.js
media.fastclick.net	97	No	w/get.media?***, w/pop.cgi?***
uac.advertising.com	95	No	wrapper/aceUAC.js
servedby.advertising.com	88	Yes	site=***
js.adsonar.com	77	No	js/adsonar.js, js/tw_adsonar.js, js/tw_dfp_adsonar.js, js/tw_cnn_adsonar.js
s7.addthis.com	77	No	js/***/addthis_widget.js, js/addthis_widget.php?***, custom/***/addthis_widget.js, js/widget.php?***

The "***" in the last column represents varying (or long) path names or parameters.

sites that provide no meaningful information on their root URLs but just point to some stored JavaScript files using URL paths. For one example, Table III lists the 20 high-profile JavaScript file hosting domains with the largest indegree values. From the third column, we can see that 13 (i.e., cells with "No" values) out of 20 sites are the type of "hidden" sites. The last column provides the example URL paths and names of the JavaScript files hosted on these domains. These JavaScript files were insecurely included by the homepages of our measured websites. For another example, Table IV lists 20 examples of low-profile JavaScript file hosting domains with indegree value one. From the second column, we can see that 15 (i.e., cells with "No" values) out of 20 sites are the type of "hidden" sites. In addition, in the last 10 examples, IP addresses instead of domain names were used by those JavaScript file hosting sites, and we observed 29 more such type of examples.

Table IV. Twenty Examples of Low-Profile JavaScript File Hosting Domains with Indegree Value One

Hosting Domains	Meaningful information on root URL?	Example URL paths and names of the included JavaScript files
app.communicatorcorp.com	No	public/scripts/ConversionTracking.js
w00tpublishers.wootmedia.net	No	delivery/ajs.php?***
houston.backpage.com	Yes	gyrobase/classifieds/include?***
mxmtrack.com	No	tag/floating_icon/discuss.js
fwevents.planetdiscover.com	Yes	GreatJobs/fw_scripts/rounded_corners.js
server2.web-stat.com	No	wtslog.js
openads.aawsat.com	No	adx.js
www.fidweb.net	No	reciprok/Cartouche?***
www.tsgonline.com	Yes	studies/zd/INVITES/pcm200805/invite.js
ads2.vortexmediagroup.com	Yes	advertpro/servlet/view/banner/javascript/zone?***
204.2.168.8	No	deliverjs.nmi?***
65.215.109.89	No	dcs8u0ga210000478ev8s20t9_1m4z/wtid.js
216.187.72.70	No	phpAdsNew/www/delivery/ajs.php?***
72.20.109.42	Yes	gyan/g.php
213.8.192.102	No	e-dologic/red/codeRed.js
63.214.178.49	No	abm.aspx?***
72.3.233.244	No	feed2js/feed2js_nobullets.php?***
213.239.222.7	No	publisher/gulli.php?***
122.70.135.87	No	rm6_stat.do?***
80.87.128.204	No	rwtag.js

The "***" in the last column represents varying or long path names or parameters.

We checked the trustworthiness of all the 30 domain names[4] listed in Table III and Table IV using a very popular community-based safe browsing tool WOT (Web of Trust) [WOT 2012] in April 2012. The ratings provided by WOT "are based on real user ratings and they tell you how much other users trust this site" [WOT 2012]. WOT did not have the records for w00tpublishers.wootmedia.net and www.fidweb.net. Among the other 28 domains, nine domains received poor or very poor ratings as shown in Table V. WOT displayed the warning message "Warning! This site has a poor reputation." for all these nine domains. The first eight of these nine domains are indeed high-profile domains listed in Table III. Therefore, from real users' point of view, many JavaScript file hosting domains are even untrustworthy. We can imagine that including JavaScript files from untrustworthy domains could be more risky than from trustworthy domains. However, we want to emphasize that no matter JavaScript file hosting domains are trustworthy or not, insecure inclusion practices themselves are risky. A recent large-scale evaluation of remote JavaScript inclusions conducted by Nikiforakis et al. provides more insights on the risks of insecure inclusion practices [Nikiforakis et al. 2012].

Among the 4,517 sites that include JavaScript files from external domains, we also observed that 125 sites only use the HTTPS protocol to retrieve JavaScript files and 138 sites use both the HTTP protocol and the HTTPS protocol to retrieve different JavaScript files. In total, there are 263 sites using HTTPS to include scripts from 72 JavaScript file hosting sites. These observations imply that some JavaScript file hosting sites do provide the secure transmission service for accessing their hosted

[4]WOT did not return results for those 10 IP addresses listed in Table IV.

Table V. Nine JavaScript File Hosting Domains with Poor or Very Poor WOT (Web of Trust) Rating

Hosting Domains	Trustworthiness Rating
edge.quantserve.com	poor
an.tacoda.net	very poor
js.revsci.net	very poor
anrtx.tacoda.net	poor
a.tribalfusion.com	poor
tags.expo9.exponential.com	poor
statse.webtrendslive.com	very poor
js.adsonar.com	poor
ads2.vortexmediagroup.com	very poor

JavaScript files, and some of our measured sites do use this service. However, this secure JavaScript transmission service is not popular. Only 3.6% (72 out of 1,985) of the JavaScript file hosting sites provide the service, and only 5.8% (263 out of 4,517) of the JavaScript file inclusion sites use the service. Also note that HTTPS protects data in transit, but it does not guarantee that a JavaScript file is uncompromised in a hosting site.

In contrast to these 4,517 sites, we did find that there are 324 other sites, in which an external included JavaScript file is always restricted within a sublevel HTML frame or iframe document whose origin is different from that of the top-level document. This observation implies that some sites do limit the permissions of external included JavaScript code within sublevel documents and provide a protection to the top-level documents of their homepages. However, such a relatively secure practice is exclusively followed by only 324 measured sites, and those 4,517 sites still use a very insecure way to include external JavaScript files.

Finally, we performed a simple manual verification of the effectiveness of the domain name comparison method described at the beginning of this subsection. We found that among the 40 sites listed in Table III and Table IV, only www.google.com and www.statcounter.com belong to our 6,805 measured websites. By further searching domain name registration information and by using common knowledge, we found that www.google-analytics.com, pagead2.googlesyndication.com, and ad.doubleclick.net belong to www.google.com that we measured; meanwhile, uac.advertising.com and servedby.advertising.com belong to www.advertising.com that we measured. All of these overlapped seven sites are high-profile ones listed in Table III, and none of the sites listed in Table IV belongs to our 6,805 measured websites. The other 33 sites do not overlap with our measured 6,805 websites; thus, we can say that they are indeed *external* JavaScript file hosting sites. We further manually examined which of these 4,517 JavaScript file inclusion sites included scripts from the seven overlapped sites. We found that six Google-related sites included www.google-analytics.com, and none of Google-related sites included either pagead2.googlesyndication.com or ad.doubleclick.net. Meanwhile, no misclassification happened on www.google.com, www.statcounter.com, uac.advertising.com, and servedby.advertising.com. Overall, we only clearly observed six misclassification cases based on those 40 examples of JavaScript file hosting sites; therefore, our domain name comparison method works well for our purpose.

5.2.2. Safe Alternatives to Insecure Inclusion. Our results show that insecure JavaScript inclusion is widely practiced by the majority (66.4%) of our measured sites. Our in-depth analysis on the domain name relationship between JavaScript file inclusion sites and hosting sites further reveals the severity and nature of these insecure practices. Although HTTPS and sublevel documents are used by a small portion of sites to

Table VI. DJS Presence by Category and Top-Level Domain

Category/ TLD	eval-generated	write-generated	innerHTML-generated	DOM-generated
arts	258(51.6%)	403(80.6%)	76(15.2%)	83(16.6%)
business	253(50.6%)	295(59.0%)	73(14.6%)	56(11.2%)
computers	205(41.0%)	307(61.4%)	55(11.0%)	55(11.0%)
games	203(40.6%)	327(65.4%)	58(11.6%)	57(11.4%)
health	190(38.0%)	276(55.2%)	35(7.0%)	34(6.8%)
home	240(48.0%)	357(71.4%)	57(11.4%)	73(14.6%)
news	314(62.8%)	412(82.4%)	161(32.2%)	110(22.0%)
recreation	229(45.8%)	310(62.0%)	67(13.4%)	57(11.4%)
reference	144(28.8%)	214(42.8%)	44(8.8%)	22(4.4%)
regional	258(51.6%)	337(67.4%)	97(19.4%)	58(11.6%)
science	137(27.4%)	234(46.8%)	39(7.8%)	35(7.0%)
shopping	245(49.0%)	307(61.4%)	37(7.4%)	38(7.6%)
society	163(32.6%)	283(56.6%)	42(8.4%)	42(8.4%)
sports	322(64.4%)	424(84.8%)	114(22.8%)	95(19.0%)
world	212(42.4%)	341(68.2%)	92(18.4%)	55(11.0%)
com	2359(49.9%)	3359(71.1%)	724(15.3%)	656(13.9%)
org	109(24.5%)	195(43.8%)	25(5.6%)	22(4.9%)
gov	32(18.8%)	50(29.4%)	9(5.3%)	9(5.3%)
net	77(36.3%)	135(63.7%)	26(12.3%)	21(9.9%)
edu	50(18.1%)	92(33.3%)	17(6.2%)	7(2.5%)
cc	393(41.4%)	558(58.7%)	130(13.7%)	79(8.3%)
other	4(16.0%)	7(28.0%)	3(12.0%)	0(0.0%)
All	3024(44.4%)	4396(64.6%)	934(13.7%)	794(11.7%)

enhance the security of external JavaScript file inclusion, we believe that the majority of measured JavaScript file inclusion sites and hosting sites have not paid sufficient attention to the potential risks of insecure JavaScript inclusion. For JavaScript file inclusion sites, we suggest they: (1) avoid external JavaScript inclusion by using internal JavaScript files from the same sites, if at all possible; (2) restrict the permission of external included scripts by placing them within a sublevel HTML frame or iframe document whose origin is different from that of the top-level document, if external inclusion is really inevitable; and (3) retrieve external JavaScript files using HTTPS connections, if the HTTPS service is available. The third suggestion needs a hosting site to provide the HTTPS service for accessing its JavaScript files, but the first two suggestions can be easily adopted by JavaScript file inclusion sites. Regarding our second suggestion, the JavaScript file inclusion sites may use some secure client-side frame communication techniques [Jackson and Wang 2007; Wang et al. 2007, 2009; Barth et al. 2008b; HTML5comm 2012; Zalewski 2012] if they need to support the interactions between sublevel and top-level HTML documents. We review these techniques in Section 6. Regarding our third suggestion, using HTTPS can at least protect the transmission of JavaScript files from external domains, especially if these JavaScript files are high-profile ones that are more likely to be targeted by attackers.

5.3. Insecure JavaScript Dynamic Generation

Since 74.9% of measured sites (5,098 out of 6,805) contain DJS scripts on their homepages, we now characterize all the DJS scripts based on their generation techniques and analyze insecure practices.

5.3.1. DJS Presence by Category and TLD. Table VI lists the overall DJS presence by category and TLD for the four different DJS generation techniques. We can see that the eval() function and the document.write() method are widely used on 44.4% and

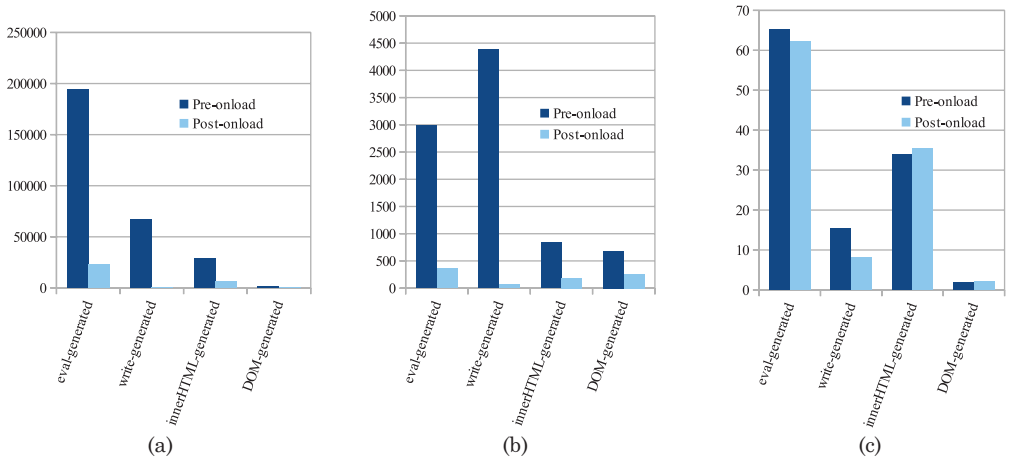

Fig. 11. DJS instance summary for pre-onload/post-onload phases: (a) total number of DJS instances; (b) total number of webpages on which those DJS instances are identified; (c) mean value of IPP (Instance Per Page).

64.6% of webpages, respectively. In contrast, the innerHTML property and the DOM methods (i.e., replacing, inserting, or appending a new created script element) are only used on 13.7% and 11.7% of webpages, respectively. It is also interesting to notice that the categories with the highest DJS presence values are news and sports for all the four generation techniques. The TLDs with the highest DJS presence values are .com, .net, and country code domains. These results indicate that JavaScript dynamic generation is more likely to be used on those sites that have more dynamic contents.

5.3.2. DJS Instance Summary. We now examine the generated DJS instances on each webpage. A DJS instance is identified using different methods for different generation techniques. For the eval() function, the whole evaluated string content is regarded as a DJS instance. Within the written content of the document.write() method and the value of the innerHTML property, a DJS instance can be identified from three sources: (1) between a pair of <script> and </script> tags; (2) in an event handler specified as the value of an HTML attribute such as onclick or onmouseover; and (3) in a URL that uses the special `javascript:protocol` specifier [Flanagan 2006]. For the DOM methods, each new script element is identified as a DJS instance. These identification methods are used no matter the DJS instances are obfuscated or not.

Figure 11 gives a summary of DJS instances for the document loading and parsing phase, denoted as the pre-onload phase, and the event-driven phase, denoted as the post-onload phase. Figure 11(a) illustrates the total number of DJS instances identified in the two execution phases for the four different techniques. Figure 11(b) illustrates the total number of webpages on which these DJS instances are identified. Figure 11(c) illustrates the average IPP (Instance Per Page) values.

It is evident that the eval() function generates the largest number of DJS instances in both phases (194,676 in the pre-onload phase and 22,632 in the post-onload phase). The mean value of IPP for eval-generated DJS instances is 65.2 in the pre-onload phase and 62.3 in the post-onload phase. The maximum value of IPP for eval-generated DJS instances reaches 2,543 in the pre-onload phase and 6,350 in the post-onload phase. These numbers indicate that eval() may be misused or abused. The document.write() method also generates a large number of DJS instances (67,446 DJS instances on 4,385 pages) in the pre-onload phase, but it only generates 519 DJS instances on

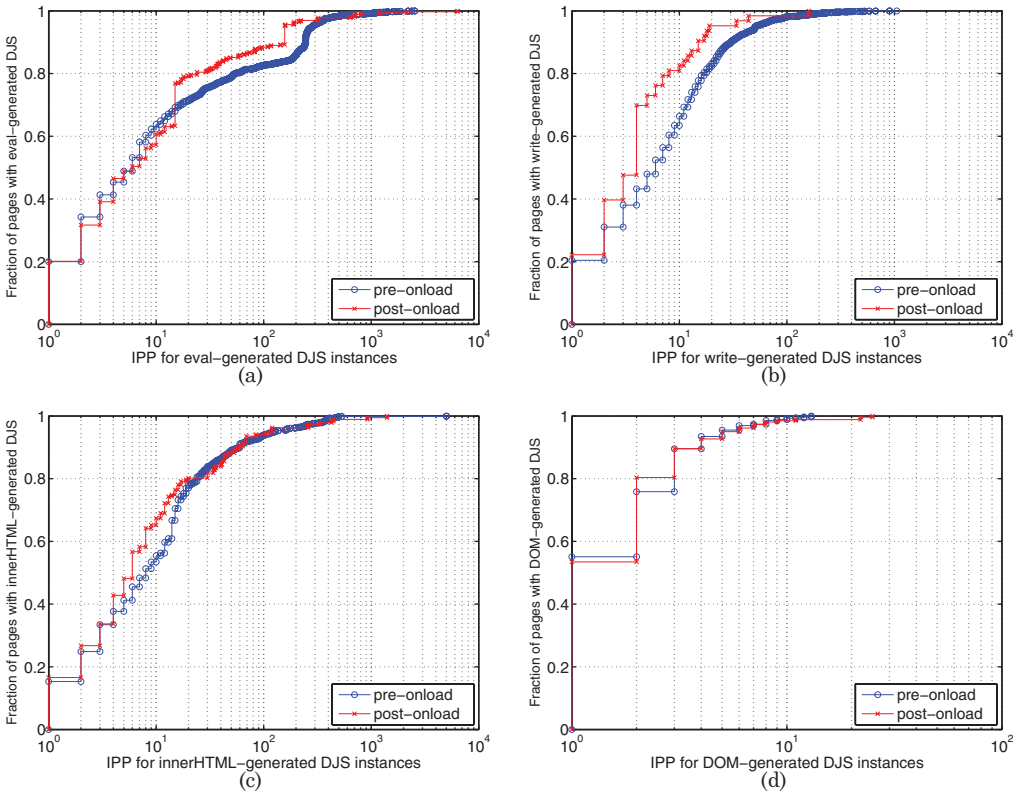

Fig. 12. Cumulative distribution of the webpages in terms of IPP (Instance Per Page) for (a) eval-generated; (b) write-generated; (c) innerHTML-generated; (d) DOM-generated DJS instances.

63 pages in the post-onload phase. Calling document.write() in the post-onload phase is usually not desirable because it will overwrite the current document with the written content. In both phases, the innerHTML property also generates a large number of DJS instances with 28,717 DJS instances on 844 pages in the pre-onload phase and 6,626 DJS instances on 187 pages in the post-onload phase. However, DOM methods generate much fewer DJS instances with 1,370 DJS instances on 680 pages in the pre-onload phase and 557 DJS instances on 260 pages in the post-onload phase.

For the four JavaScript dynamic generation techniques, Figures 12(a) to 12(b) further illustrate the cumulative distribution of the webpages in terms of IPP. In each of these four figures, the "o" curve is for the pre-onload phase and the "$*$" curve is for the post-onload phase. Note that the total number of pages is different for the two phases (as shown in Figure 11(b)), and we present the two curves together for ease of comparison. We can see that the indication of misuse or abuse is especially evident for the eval() function. While the majority (about 60%) of webpages have 10 or less eval-generated DJS instances, nearly 17% and 11% of webpages have 100 or more eval-generated DJS instances for the pre-onload phase and the post-onload phase, respectively.

5.3.3. Structural Analysis of Eval-Generated DJS. The prevalence of DJS on various categories of webpages and the high IPP values (Figure 12) motivate us to further understand the use purposes of the large number of DJS instances. Using our JavaScript code classification tools, we now uncover the use purposes of eval-generated DJS instances in terms of programming language functionality.

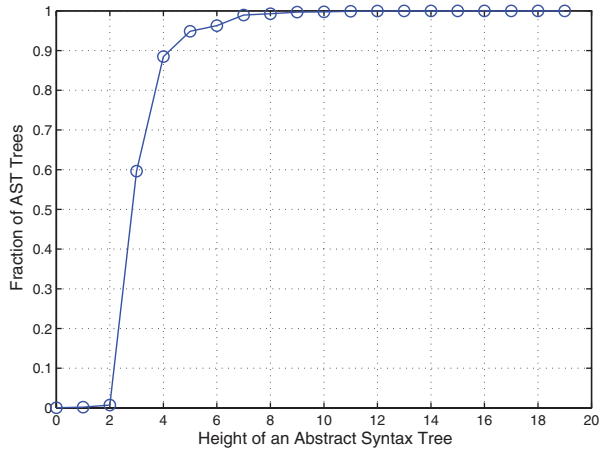

Fig. 13. Cumulative distribution of the AST trees in terms of the height of an AST tree for eval-generated DJS instances.

From the total 217,308 (both the pre-onload phase and the post-onload phase) eval-generated DJS instances, 217,308 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 19. Figure 13 shows the cumulative distribution of the AST trees in terms of the height of an AST tree. Nearly 90% of AST trees have a height less than or equal to 4. Therefore, distinct AST signatures can be accurately created or matched for 90% of AST trees if top four levels of AST trees are compared. Meanwhile, as exemplified in Section 3.2, selecting a smaller N value may compromise accuracy, while selecting a larger N value may unnecessarily create more AST signatures. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top four-level structure of AST trees to create and match AST signatures. Indeed, all the four JavaScript examples (a), (b), (c), and (d) illustrated in Figure 2 are eval-generated DJS instances recorded from one arts website homepage, one science website homepage, one regional website homepage, and one health website homepage, respectively.

With $N = 4$, a total number of 647 AST signatures are created and matched from the 217,308 AST trees. These 647 AST signatures capture the essential structural information of the 217,308 AST trees, and they greatly facilitate our further analysis. Using more levels of AST tree structure is unnecessary because lower-level AST tree nodes only contain less important structural information; meanwhile, more AST signatures will be created and matched. For example, with $N = 5$, about 808 AST signatures will be created and matched from the 217,308 AST trees.

Finally, AST signatures with the same programming language functionality are merged into the same category by using our AST signature categorization tool. For example, two AST signatures representing two types of function calls with different number or type of parameters are merged into the same *function calls* category. Table VII lists the final 17 categories of DJS instances classified from the 647 AST signatures, and in turn from the 217,308 DJS instances.

We can see that 0.05% of the DJS instances have *parse error* when AST trees are extracted, and 0.13% of the DJS instances have *empty content*. The majority (around 98.6%) of the eval-generated DJS instances are classified into the 14 categories from *simple expression* to *try-catch statements*. The DJS instances in these 14 categories all have specific use purposes in terms of programming language functionality. Only 1.2% of the DJS instances have mixed programming language functionalities, and they are

Table VII. The 17 Categories of Eval-Generated DJS Instances

Category	Presence in Pages	Number of DJS instances	Average DJS length
parse error	20(0.7%)	111(0.05%)	1175.5
empty content	124(4.1%)	291(0.13%)	0.0
simple expression	1209(40.0%)	134251(61.8%)	13.9
arithmetic expression	12(0.4%)	158(0.1%)	67.9
relational expression	79(2.6%)	3246(1.5%)	31.7
logical expression	159(5.3%)	3249(1.5%)	75.8
object/array literal	265(8.8%)	4798(2.2%)	623.0
other expression	126(4.2%)	5789(2.7%)	18.5
variable declarations	98(3.2%)	411(0.2%)	586.0
function declarations	1157(38.3%)	1929(0.9%)	13380.7
assignment statements	1289(42.6%)	42015(19.3%)	51.9
function calls	527(17.4%)	2733(1.3%)	368.9
method calls	561(18.6%)	2062(0.9%)	75.9
object/array creations	29(1.0%)	212(0.1%)	41.3
conditional statements	181(6.0%)	9371(4.3%)	519.1
try-catch statements	1075(35.5%)	4127(1.9%)	51.7
mixed statements	910(30.1%)	2555(1.2%)	6884.1

classified into the last category of *mixed statements*. The generated DJS instances in the last 15 categories are either various expressions (from *simple expression* to *other expression*) or various statements (from *variable declarations* to *mixed statements*). In general, a JavaScript expression is used only to produce a value, while a JavaScript statement normally has side-effects and is often used to accomplish some tasks. Also note that these categories do not represent the semantic meanings of DJS instances. For example, eval() is often used to convert a string into a JSON [2011] object. We found many JSON evaluation DJS instances such as ({“system”:[{“id”:“286”,“name”:“General 5 Star”}]}) in the “object/array literal” category. However, JSON evaluation itself can also be a component of more complex DJS instances that belong to other categories such as the “assignment statements” category.

From the preceding result that the majority of the eval-generated DJS instances have specific use purposes in terms of programming language functionality, we would like to suggest replacing the eval() function with DOM methods such as document.createElement() and document.createTextNode() to generate those DJS instances. This replacement could be feasibly performed since those DJS instances have specific use purposes, and it could also bring the most security benefit to us since the majority of eval-generated DJS instances will be replaced. Researchers and developers may consider to design new algorithms or tools to automate this replacement. Language designers may even consider to devise new DOM methods that can be more conveniently or efficiently used as the replacements.

5.3.4. Safe Alternatives to Eval(). To further understand whether using eval() is necessary in these different categories, we randomly sampled and inspected both the content and the calling context of 700 DJS instances. We sampled 200 DJS instances from the simple expression category and 200 DJS instances from the assignment statements category. These two categories have the largest numbers of DJS instances, accounting for 61.8% and 19.3%, respectively, of all the eval-generated DJS instances. The remaining 300 DJS instances are sampled from the other 15 categories, with each of them contributing 20 instances.

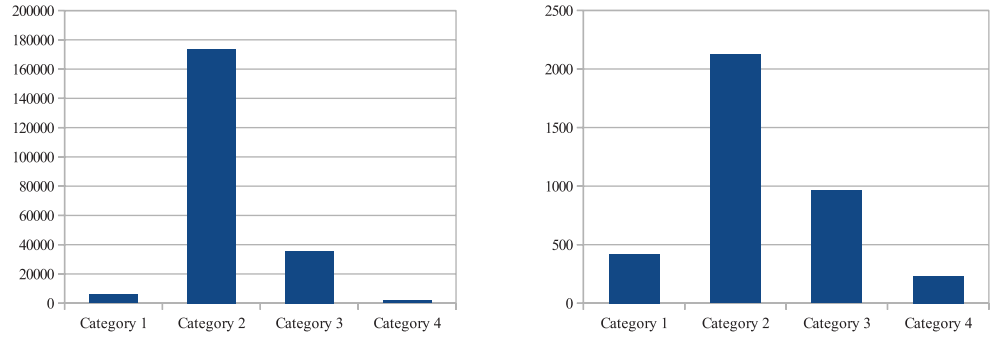

Fig. 14. Domain-based analysis of eval-generated DJS instances. Category 1 to 4 represent "embedded in the top-level document", "securely included into the top-level document from the same domain", "insecurely included into the top-level document", and "embedded or included into the frame or iframe documents with domains different from that of the top-level document", respectively. (a) illustrates the total number of DJS instances; (b) illustrates the total number of webpages on which those DJS instances are identified.

In at least 70% of the sampled cases, the eval() function is misused or abused while safe alternatives can be easily identified. Here we illustrate three representative sampled cases. The first one is: this.homePos = eval("0" + this.dirType + this.dim), in which a string simple expression "0-500" is generated. Indeed, such a kind of string concatenation directly generates a string value, and using eval() is redundant. The second one is: var ff_nav=eval("nav_"+tt[i][1]), in which a variable name "nav_20912" is dynamically accessed. A safe alternative is using the JavaScript *window* object to directly access the variable, that is, var ff_nav=window["nav_"+tt[i][1]]. The third one is: var responses = eval(o.responseText), in which the response content of an XMLHttpRequest [XHR 2011] is directly evaluated. This practice is used in many of our sampled cases to convert a *responseText* into a JSON object. However, since malicious JavaScript code could be injected into the *responseText*, it would be better to use a JSON parser rather than the eval() function to perform such a transformation [JSON 2011]. The other 30% of the sampled cases usually have complex calling context, so we do not further identify their safe alternatives.

We suggest that eval() should be avoided if at all possible. In addition to the safe alternatives exemplified earlier, DOM methods can be generally used to generate and execute various JavaScript statements. In a recent large-scale study of the use of eval() function in Web applications, Richards et al. [2011b] provided more examples of safe alternatives to the eval() function.

5.3.5. Domain-Based Analysis of Eval-Generated DJS. To estimate the extent to which the measured websites can directly replace those eval() function calls, we perform a domain-based analysis of the eval-generated DJS instances. Using the top-level document as the base, we classify the eval-generated DJS instances into four categories as shown in Figure 14.

In the first category, each eval-generated DJS instance is directly embedded in the top-level document of a homepage. We identified 5,840 this category of DJS instances on 422 homepages. Because these measured websites directly control the eval() function calls embedded in the top-level documents, they can definitely analyze their webpages to check whether safe alternatives can be used to directly replace those eval() function calls.

Table VIII. Structural Analysis of DJS Instances Generated by the Document.write() Method, innerHTML Property, and DOM Methods

Technique and Type		Presence in Pages	Number of DJS instances	Avg. length	Main usage
write	jscode	4000	26125	77	JS inclusion
	eventhandler	1773	38650	39	function call
	jsprotocol	501	3190	45	function call
innerHTML	jscode	120	503	262	JS inclusion
	eventhandler	747	31267	60	function call
	jsprotocol	336	3573	33	function call
DOM	src	779	1866	-	JS inclusion
	text	33	61	623	assignment

In the second category, each eval-generated DJS instance is securely included into the top-level document of a homepage from the same domain. This means these measured websites themselves host the included JavaScript files and completely control these JavaScript files. Therefore, these websites can also directly analyze their webpages and replace eval() function calls whenever possible. We identified 173,833 this category of DJS instances on 2,124 homepages. These instances account for the majority (80.0%) of eval-generated DJS instances.

In the third category, each eval-generated DJS instance is insecurely included into the top-level document of a homepage. In other words, these DJS instances are associated with the insecure JavaScript inclusion practices as defined in Section 5.2. We identified 35,708 this category of DJS instances on 964 homepages. These measured websites cannot directly control or replace the eval() function calls because these calls are made in the JavaScript files from external domains. Our suggestion to these websites is to first fix the insecure JavaScript inclusion problem, and then replace the eval() function calls with safe alternatives.

In the fourth category, each eval-generated DJS instance is embedded or included into a sublevel HTML frame or iframe document whose domain is different from that of the top-level document. We identified 1,927 this category of DJS instances on 233 homepages. These DJS instances are not insecurely included, but they still cannot be directly replaced by the measured websites because they belong to documents of external domains. DJS instances in this category need to be replaced by those frame or iframe document websites themselves.

Overall, the first two categories account for the majority (82.7%) of all the eval-generated DJS instances. Fortunately, these DJS instances are fully controlled by the measured websites; therefore, they can be directly replaced by these websites if corresponding safe alternatives exist. The replacement of eval-generated DJS instances in the last two categories has to be done by those external JavaScript file hosting sites or those external HTML frame or iframe document hosting sites.

5.3.6. Structural Analysis of Other Types of DJS. As mentioned before, the DJS instances generated by the document.write method() and the innerHTML property are identified from three different sources. We use *jscode* to represent a DJS instance identified between a pair of <script> and </script> tags, use *eventhandler* to represent a DJS instance identified in an event handler, and use *jsprotocol* to represent a DJS instance identified in a `javascript:protocol` URL. The DJS instances generated by the DOM methods are specified in either the `src` attribute or the `text` attribute of a script element. Table VIII gives the overall structural analysis results of the DJS instances generated by these three dynamic generation techniques. The main usage of each type of DJS instance is summarized in the last column of Table VIII.

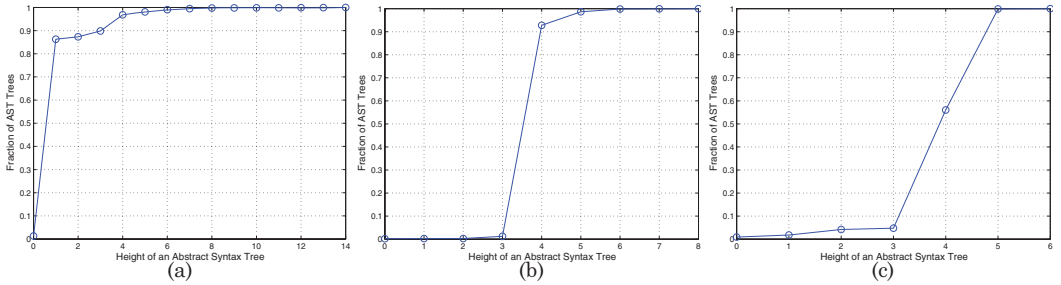

Fig. 15. Cumulative distribution of the AST trees in terms of the height of an AST tree for write-generated (a) *jscode* type of; (b) *eventhandler* type of; (c) *jsprotocol* type of DJS instances.

The detailed structural analysis of the write-generated, innerHTML-generated, and DOM-generated DJS instances is as follows.

(1) *Detailed analysis of write-generated DJS instances.* As shown in Table VIII, there are three types of write-generated DJS instances. From the total 26,125 *jscode* type of DJS instances, 26,125 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 14. Figure 15(a) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Over 96% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top four-level structure of AST trees to create and match AST signatures. A total number of 357 AST signatures are created and matched from the 26,125 AST trees. These 357 AST signatures capture the essential structural information of the 26,125 AST trees.

Similarly, from the total 38,650 *eventhandler* type of DJS instances, 38,650 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 8. Figure 15(b) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Over 92% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top four-level structure of AST trees to create and match AST signatures. A total number of 376 AST signatures are created and matched from the 38,650 AST trees. These 376 AST signatures capture the essential structural information of the 38,650 AST trees.

Furthermore, from the total 3,190 *jsprotocol* type of DJS instances, 3,190 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 6. Figure 15(c) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Over 99% of AST trees have a height less than or equal to 5. Therefore, we selected $N = 5$ as the input parameter (Figure 4) and used the top five-level structure of AST trees to create and match AST signatures. A total number of 85 AST signatures are created and matched from the 3,190 AST trees. These 85 AST signatures capture the essential structural information of the 3,190 AST trees.

Finally, AST signatures with the same programming language functionality are merged into the same category by using our AST signature categorization tool. Table IX lists the final categories of DJS instances classified from the 357, the 376, and the 85 AST signatures for the three types of write-generated DJS instances, respectively.

(2) *Detailed analysis of innerHTML-generated DJS instances.* As shown in Table VIII, there are three types of innerHTML-generated DJS instances. From the total 503 *jscode* type of DJS instances, 503 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 11. Figure 16(a) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Nearly

Table IX. Categories of Write-Generated DJS Instances

Type and Category		Presence in Pages	Number of DJS instances	Average DJS length
jscode	parse error	202(5.1%)	342(1.3%)	1095.5
	empty with src	3891(97.3%)	21678(83.0%)	0.0
	variable declarations	418(10.5%)	629(2.4%)	311.8
	function declarations	198(5.0%)	275(1.1%)	309.9
	assignment statements	435(10.9%)	1194(4.6%)	108.7
	function calls	77(1.9%)	172(0.7%)	182.8
	method calls	269(6.7%)	464(1.8%)	237.0
	object/array creations	24(0.6%)	24(0.1%)	660.0
	conditional statements	35(0.9%)	114(0.4%)	257.1
	try-catch statements	3(0.1%)	30(0.1%)	44.7
	mixed statements	649(16.2%)	1203(4.6%)	852.8
eventhandler	parse error	13(0.7%)	75(0.2%)	52.9
	dumb return statement	95(5.4%)	175(0.5%)	11.8
	variable declarations	3(0.2%)	216(0.6%)	31.5
	assignment statements	545(30.7%)	7566(19.6%)	40.9
	function calls	1227(69.2%)	22166(57.4%)	28.9
	method calls	358(20.2%)	6320(16.4%)	46.1
	conditional statements	66(3.7%)	413(1.1%)	67.9
	try-catch statements	2(0.1%)	72(0.2%)	54.5
	mixed statements	112(6.3%)	1647(4.2%)	123.7
jsprotocol	parse error	11(2.2%)	29(0.9%)	73.2
	dumb return statement	109(21.8%)	1237(38.8%)	6.7
	variable declarations	2(0.4%)	13(0.4%)	215.1
	assignment statements	5(1.0%)	5(0.2%)	49.4
	function calls	310(61.9%)	1478(46.3%)	42.9
	method calls	93(18.6%)	267(8.4%)	135.0
	try-catch statements	2(0.4%)	2(0.1%)	38.0
	mixed statements	30(6.0%)	159(5.0%)	199.3

90% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top four-level structure of AST trees to create and match AST signatures. A total number of 58 AST signatures are created and matched from the 503 AST trees. These 58 AST signatures capture the essential structural information of the 503 AST trees.

Similarly, from the total 31,267 *eventhandler* type of DJS instances, 31,267 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 8. Figure 16(b) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Over 87% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top four-level structure of AST trees to create and match AST signatures. A total number of 286 AST signatures are created and matched from the 31,267 AST trees. These 286 AST signatures capture the essential structural information of the 31,267 AST trees.

Furthermore, from the total 3,573 *jsprotocol* type of DJS instances, 3,573 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 6. Figure 16(c) shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Nearly 85% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 4) and used the top five-level structure of AST trees to create and match AST signatures. A total number of

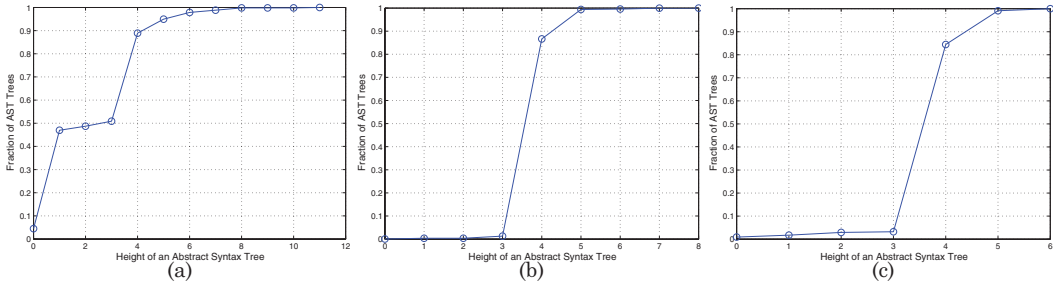

Fig. 16. Cumulative distribution of the AST trees in terms of the height of an AST tree for innerHTML-generated (a) *jscode* type of; (b) *eventhandler* type of; (c) *jsprotocol* type of DJS instances.

Table X. Categories of innerHTML-Generated DJS Instances

Type and Category		Presence in Pages	Number of DJS instances	Average DJS length
jscode	parse error	4(3.3%)	5(1.0%)	3077.4
	empty with src	53(44.2%)	175(34.8%)	0.0
	variable declarations	9(7.5%)	12(2.4%)	216.0
	function declarations	12(10.0%)	13(2.6%)	595.6
	assignment statements	26(21.7%)	45(8.9%)	203.0
	function calls	37(30.8%)	76(15.1%)	339.6
	method calls	28(23.3%)	50(9.9%)	292.2
	conditional statements	5(4.2%)	5(1.0%)	497.2
	try-catch statements	1(0.8%)	1(0.2%)	47.0
	mixed statements	37(30.8%)	121(24.1%)	395.7
eventhandler	parse error	10(1.3%)	17(0.05%)	68.7
	dumb return statement	40(5.4%)	371(1.2%)	9.2
	assignment statements	148(19.8%)	4359(13.9%)	61.2
	function calls	500(66.9%)	22154(70.9%)	34.4
	method calls	275(36.8%)	3503(11.2%)	82.6
	object/array creations	2(0.3%)	3(0.01%)	109.3
	conditional statements	29(3.9%)	105(0.3%)	112.9
	try-catch statements	7(0.9%)	12(0.04%)	248.3
	mixed statements	57(7.6%)	743(2.4%)	708.2
jsprotocol	parse error	8(2.4%)	32(0.9%)	64.8
	dumb return statement	119(35.4%)	764(21.4%)	9.2
	simple expression	2(0.6%)	3(0.1%)	63.0
	function calls	159(47.3%)	1945(54.4%)	30.7
	method calls	103(30.7%)	811(22.7%)	59.3
	conditional statements	1(0.3%)	1(0.03%)	112.0
	mixed statements	5(1.5%)	17(0.5%)	91.7

64 AST signatures are created and matched from the 3,573 AST trees. These 64 AST signatures capture the essential structural information of the 3,573 AST trees.

Finally, AST signatures with the same programming language functionality are merged into the same category by using our AST signature categorization tool. Table X lists the final categories of DJS instances classified from the 58, the 286, and the 64 AST signatures for the three types of innerHTML-generated DJS instances, respectively.

(3) *Detailed analysis of DOM-generated DJS instances.* As shown in Table VIII, there are two types of DOM-generated DJS instances. The *src* type of DJS instances contain

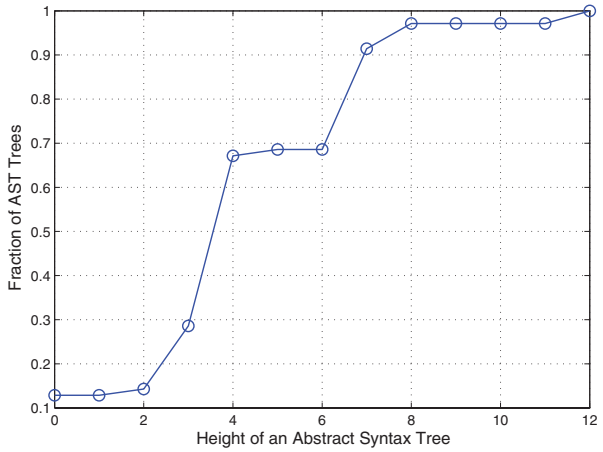

Fig. 17. Cumulative distribution of the AST trees in terms of the height of an AST tree for DOM-generated DJS instances.

Table XI. Categories of DOM-Generated DJS Instances

Type and Category		Presence in Pages	Number of DJS instances	Average DJS length
text	variable declarations	9(27.3%)	10(16.4%)	32.8
	function declarations	2(6.1%)	3(4.9%)	923.3
	assignment statements	10(30.3%)	20(32.8%)	655.3
	function calls	4(12.1%)	5(8.2%)	103.6
	conditional statements	11(33.3%)	18(29.5%)	696.8
	mixed statements	4(12.1%)	5(8.2%)	1742.0

the URL of JavaScript files, so we only further analyze the *text* type of DJS instances. From the total 61 *text* type of DJS instances, 61 AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 12. Figure 17 shows the cumulative distribution of these AST trees in terms of the height of an AST tree. Because there are only 61 DJS instances, we manually analyzed them and classified them into the six categories as shown in Table XI.

5.3.7. Safe Alternatives to Jscode Generation via Document.write() and innerHTML. For the eventhandler and jsprotocol DJS instances generated by document.write() and innerHTML, their usages are relatively safe. When new content is added to a document, event handlers are directly specified on various elements of the newly added content to respond to various events. The `javascript:protocol` scripts are often used on links to execute some statements without loading a new document.

What we emphasize is that generating jscode using document.write() and innerHTML is not desirable. For document.write(), the generated jscode is immediately executed. Multiple document.write() calls can be used to construct a jscode, and document.write() calls can be nested. All these factors make the filtering of write-generated malicious JavaScript code a very challenging task [Yu et al. 2007]. However, our results show that 26,125 instances of write-generated jscode are identified on a large number of 4,000 homepages. For innerHTML, the generated jscode is recognized by a browser, but it is not necessarily executed. For example, Firefox does not directly execute a jscode generated by innerHTML. In Internet Explorer, the `defer` attribute and some tricks need to be used to execute an innerHTML-generated jscode, but this practice is also

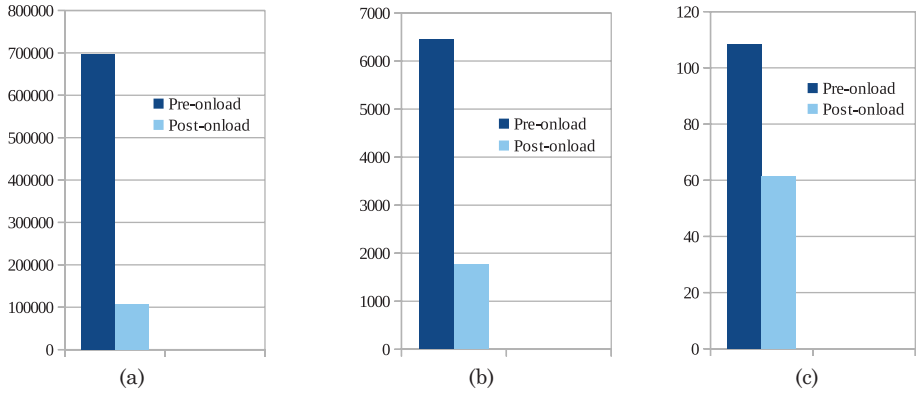

Fig. 18. Event handler registration summary for pre-onload/post-onload phases: (a) total number of event handler registration instances; (b) total number of webpages on which these registrations occurred; (c) mean value of IPP (Instance Per Page).

not recommended due to potential script-injection attacks [MSDN 2011]. Fortunately, only 503 instances of this practice are identified on 120 pages as shown in Table VIII.

The best practice is to use DOM methods to dynamically generate JavaScript code. Using DOM methods (such as createElement() and createTextNode()) to create JavaScript elements explicitly declares that the new elements are scripts. This practice can enable potential Web content protection mechanisms such as those presented in Jim et al. [2007], Reis et al. [2006], and Yu et al. [2007] to accurately define security policies and weed out potential malicious JavaScript code. Unfortunately, only 1,927(1,866 plus 61) instances of this practice are identified.

Our results show that the main usage of jscode generated by document.write() and innerHTML is for including other JavaScript files (denoted as *JS inclusion* in Table VIII). Indeed, by specifying the `src` attribute of a script element, DOM methods fit well for such a usage. By specifying the `text` attribute of a script element, DOM methods can also be used to generate and execute various statements such as assignment statements or function calls, thus safely replacing the relatively insecure practices of jscode generation via document.write() and innerHTML.

5.4. Event Handler Registration

Overall, event handler registrations occurred on 6,456 (94.9%) homepages. Figure 18 provides a detailed event handler registration summary for both the pre-onload phase and the post-onload phase. About 698,057 event handler registrations occurred on 6,451(94.8%) pages in the pre-onload phase, with an average of 108.2 registrations per page and a maximum of 5,074 registrations per page. About 108,428 event handler registrations occurred on 1,767(26.0%) pages in the post-onload phase, with an average of 61.4 registrations per page and a maximum of 2,229 registrations per page. Figure 19 further illustrates the cumulative distribution of the webpages in terms of IPP (Instance Per Page) for event handler registration. We can see that event handler registrations are very popular. Over 30% of webpages have 100 or more event handler registrations in the pre-onload phase, and over 20% of webpages have 100 or more event handler registrations in the post-onload phase.

Our measurement results include all the popular event handler registration techniques supported in Firefox. Figure 20 presents the summary of the event handler registration instances based on five different registration techniques. From bottom to top, the first technique “setting attribute event listeners” is the traditional technique

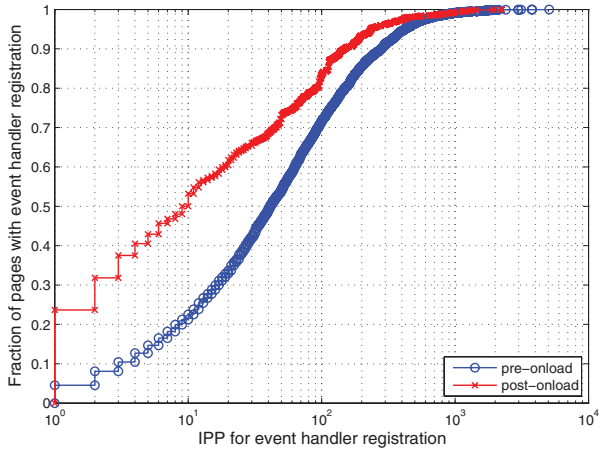

Fig. 19. Cumulative distribution of the webpages in terms of IPP (Instance Per Page) for event handler registration.

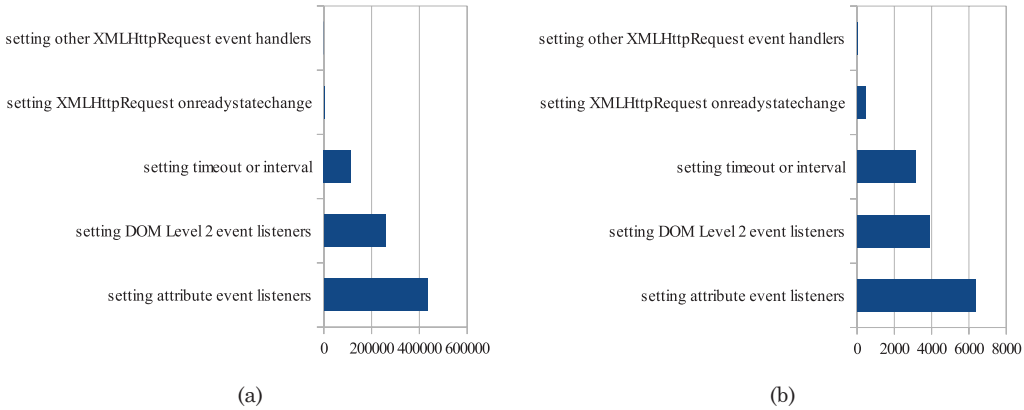

Fig. 20. Analysis of five event handler registration techniques for (a) total number of event handler registration instances; (b) total number of webpages on which those registrations occured.

that directly sets the attribute event listeners for HTML elements. It can take different forms such as the following three examples.

(1) <td id="row1" onclick="doSubmit()"> row one </td>
(2) document.getElementById('row1').onclick=doSubmit;
(3) document.getElementById('row1').setAttribute('onclick', 'doSubmit()');

Here doSubmit() is a function, and the first example directly specifies the onclick event handler for the *td* HTML element, and the last two examples specify the onclick event handler using JavaScript. Overall, this traditional technique was widely used in 434,127 instances on 6,347 (93.3%) homepages.

The second technique "setting DOM Level 2 event listeners" is the addEventListener method introduced in DOM Level 2 Events [DOM2Events 2012]. This technique allows an HTML element to register multiple event handlers for an event type and also specify the activation phase of each handler [DOM2Events 2012]. This powerful technique

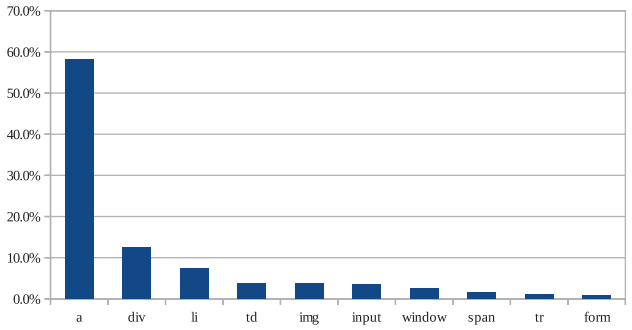

Fig. 21. Event handler registration: top 10 object types.

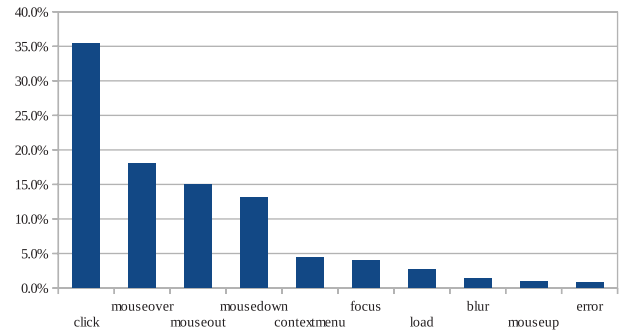

Fig. 22. Event handler registration: top 10 event types.

was also widely used in 257,282 instances on 3,879(57.0%) homepages. The third technique "setting timeout or interval" includes the using of two JavaScript functions setTimeout() and setInterval()[HTMLTimers 2012], which can schedule a timeout to run a specified handler once or periodically. These timer-based event handler registration techniques were used in 113,729 instances on 3,134(46.1%) homepages. In terms of the XMLHttpRequest [XHR 2011]-related event handler registration techniques, setting the popular onreadystatechange event handler occured in 1,278 instances on 466(6.8%) homepages, and setting other XMLHttpRequest event handlers such as onload and onerror only occured in 69 instances on 13(0.2%) homepages.

For the first two event handler registration techniques "setting attribute event listeners" and "setting DOM Level 2 event listeners", we further identified the corresponding object types and event types for all the registration instances. Figure 21 illustrates the top 10 types of objects on which event handlers were registered. We can see that the majority (58.1%) of event handler registrations were for the anchor (i.e., the $<a>$ tag) type of objects. The division (i.e., the $<div>$ tag) type of objects took the second position with about 12.6% of event handler registrations. Overall, these top 10 object types account for 95.3% of our measured event handler registrations that use the first two techniques. Figure 22 illustrates the top 10 event types. We can see that *click* is the most popular event appearing in about 35.4% of registrations. It is followed by *mouseover*, *mouseout*, and *mousedown* events appearing in about 18.0%, 15.0%, and 13.1% of registrations, respectively. Overall, these top 10 event types account for 95.5% of our measured event handler registrations that use the first two techniques.

5.5. Discussions

The insecure JavaScript practices captured in our measurement study are likely conservative estimates for two main reasons. One is that our results only include the event-driven phase for a short period of time. We described in Section 4 that our JavaScript execution dataset covers the whole document loading and parsing phase and 10 seconds of the event-driven phase for each of the 6,805 homepages. As just shown in Section 5.4, event handler registrations are very popular. Therefore, after 10 seconds of the event-driven phase, the execution of those registered event handlers may trigger further JavaScript inclusion and dynamic generation as well as event handler registration. Those JavaScript execution results are not included in our measurement study. The other reason is that we only measured the homepages of 6,805 websites so that we can have a consistent measurement, while it is possible that insecure JavaScript practices also exist on other types of webpages. These two reasons are also the limitations of our measurement study; however, due to the importance of the homepages to websites, measuring the insecure JavaScript practices on homepages is sufficient to illustrate the severity of the problem and achieve the primary objective of this work.

In a recent large-scale study of the use of eval() function, Richards et al. [2011b] measured three datasets: INTERACTIVE, PAGELOAD, and RANDOM. Their PAGELOAD dataset and RANDOM dataset were based on the 10,000 most popular websites listed by Alexa.com. The PAGELOAD dataset is similar to ours in the sense that the document loading and parsing phase and a few seconds of the event-driven phase are included but no events are intentionally triggered. Their RANDOM dataset extends the PAGELOAD dataset by further randomly triggering events such as mouse movement and link clicking. Their measurement results demonstrate that the difference in the use of eval() function between RANDOM and PAGELOAD datasets is only 2%, indicating that "sites relying on eval do so even without user interaction" [Richards et al. 2011b]. Based on their comprehensive results, we can say that our results, although conservative, are still representative of the measured popular websites.

Our measurement study focuses on examining the severity and nature of two types of insecure JavaScript practices on the legitimate websites. We do not have the data to further answer the question regarding the extent to which today's attackers are taking advantage of those insecure practices to perform malicious activities. This is another limitation of our work, and we believe the question can be properly answered only if a representative set of real-world Web-based attack examples can be sufficiently analyzed. However, even without answering this question, we should still emphasize that website developers and administrators should pay serious attention to insecure JavaScript practices and should reduce their potential security risks.

6. RELATED WORK

To the best of our knowledge, there is no directly related work on characterizing the insecure practices of JavaScript inclusion and dynamic generation. Therefore, we mainly review some related JavaScript measurement studies and JavaScript security enhancement approaches.

Krishnamurthy and Wills [2006] measured the homepages of 1,158 unique sites selected from Alexa.com to study the content delivery trade-offs in Web access. The focus of their study is on the performance impact of extraneous content, and their results show that JavaScript is often used on popular webpages to retrieve extraneous content such as images and advertisements. Richards et al. [2010] measured the dynamic behavior of JavaScript programs on 100 websites and three industry benchmark suites. The main goal of their work is to either confirm or invalidate some common

assumptions about JavaScript programs made in the programming language research area. Similarly, Ratanaworabhan et al. [2010] measured some popular Web applications and JavaScript benchmarks, and they found that real Web applications behave very differently from the benchmarks. In their following-up work, Richards et al. [2011a] further proposed a record-and-replay approach to automatically generate realistic and representative JavaScript benchmarks.

Richards et al. [2011b] also performed a large-scale study of the use of eval() function in Web applications. They categorized the behavior of the eval() function using five important metrics: mix of operations, scope affected by operations, patterns of usage, provenance of arguments, and consistence of arguments [Richards et al. 2011b]. This comprehensive study also confirms that the eval() function is pervasively used and indeed is often misused. Kiciman and Livshits [2010] designed AjaxScope, a flexible platform that can perform on-the-fly parsing and instrumentation of JavaScript code sent to users' browsers. AjaxScope can remotely monitor and report the runtime behavior of Web applications to perform many important tasks such as error reporting, performance profiling, memory leak detection, and optimization analyses. Using AjaxScope, Kiciman and Livshits [2010] analyzed the behavior of over 90 Web 2.0 applications, and one of their interesting observations is that well-behaved Web 2.0 applications do not frequently use dynamic code generation techniques mainly due to the performance penalty. The eval() function is a performance bottleneck as further highlighted in Richards et al. [2011b]. In general, the focuses of these related measurement studies are not directly on security.

In the investigation of drive-by download attacks, several execution-based measurement studies [Moshchuk et al. 2006; Wang et al. 2006; Provos et al. 2008; Cova et al. 2010] have been conducted to identify malicious webpages that contain code (in many cases, JavaScript code) for exploiting Web browser vulnerabilities and installing malware. Instead of targeting at malicious sites, our focus in this work is on legitimate websites' insecure JavaScript practices.

Some browser-instrumentation-related Web security projects have also been done recently. Finifter et al. [2010] demonstrated that existing blacklist-based static verifiers used in advertising networks do not provide perfect containment of advertisements. This is because malicious advertisements can exploit the new methods or properties exposed by the publisher websites to mount cross-site scripting attacks against publishers. The authors proposed a whitelist-based modification of the existing static verifiers, so that security can be improved while the performance and deployment advantages of static verification can be preserved. Singh et al. [2010] examined access control policies in modern Web browsers and pointed out three major incoherences. They also performed a measurement study to identify unsafe policies that could be removed by browser vendors with little compatibility cost. Our work is related to these projects in the sense that we all instrumented browsers to study different Web security problems.

A few mechanisms have been proposed to specifically improve the overall security and reliability of client-side JavaScript programs, and some of them address the insecure JavaScript dynamic generation problem. Guarnieri and Livshits [2009] proposed GATEKEEPER, a mostly static approach to sound points-to analysis for JavaScript programs. They defined nine security and reliability policies and demonstrated that their approach can effectively reason about and identify malicious or buggy JavaScript widgets. In GATEKEEPER, some JavaScript dynamic generation techniques such as the eval() function are disallowed to enable sound static analysis, and this design choice is also driven by the authors' measurement results that the eval() function is only used in 6% of their analyzed eight thousand JavaScript widgets. Later, Meyerovich and Livshits [2010] proposed ConScript, a browser-based deep advice system for security. ConScript enables fine-grained application-specific security policies to be expressed in

webpages while enforced at runtime within JavaScript engine. Curtsinger et al. [2011] proposed ZOZZLE, a low-overhead JavaScript malware detection and prevention solution that could be deployed in Web browsers. ZOZZLE is mostly a static solution and it mainly uses the Bayesian classification of hierarchical features of the JavaScript AST to detect JavaScript malware. As highlighted in their paper, the AST-based technique can capture the context information of JavaScript code and provide increased precision in comparison to the pure text-based classification [Curtsinger et al. 2011]. In ZOZZLE, the depth or height of AST is also considered because it presents a trade-off between the classification accuracy and performance.

In recent advances in new browser architectures and HTML specifications, different secure client-side frame communication techniques have been proposed [Jackson and Wang 2007; Wang et al. 2007, 2009; Barth et al. 2008b; HTML5comm 2012; Zalewski 2012], and they can be used by Websites to address the insecure JavaScript inclusion problem. Jackson and Wang [2007] proposed Subspace, which is a cross-domain communication channel that can manipulate the document domain property to pass JavaScript objects across the frames. Subspace provides a safer alternative to the insecure practice of using cross-domain <script> tags. Wang et al. [2007; 2009] highlighted that the <script> abstraction is mainly for including open content while the <frame> abstraction is mainly for supporting isolated content. They proposed two abstractions <Sandbox> and <OpenSandbox> to further provide both security and ease in creating client-side mashups [Wang et al. 2007]. In HTML 5, the postMessage API [HTML5comm 2012] is provided to enable secure cross-domain communication. Barth et al. [2008b] demonstrated that attackers can breach the confidentiality of the postMessage communication channel. They extended the postMessage API to provide confidentiality and their proposal has been adopted by the HTML 5 working group and some popular browsers [Barth et al. 2008b]. In HTML 5, a useful *sandbox* attribute is provided to flexibly restrict the behavior of the framed content with fine granularity [HTML5sandbox 2012; Zalewski 2012]; for example, JavaScript in the framed content could be completely disallowed or partially restricted by assigning different values to the sandbox attribute. Website designers and developers should consider the specific requirements of their Web applications and the availability of these mechanisms in different Web browsers to decide which mechanisms are most appropriate for them.

7. CONCLUSION

In this work, we presented the first measurement study on insecure practices of using JavaScript on the Web. We focused on investigating the severity and nature of insecure JavaScript inclusion and dynamic generation. Through an instrumented Mozilla Firefox 2 Web browser, we visited the homepages of 6,805 popular websites in 15 different categories. We found that at least 66.4% of the measured websites have the insecure practices of including JavaScript files from external domains into the top-level documents of their homepages. Our in-depth analysis on the domain name relationship between JavaScript file inclusion sites and hosting sites further reveals the severity and nature of these insecure practices. Our measurement results on JavaScript dynamic generation show that the "evil" function eval() was called on 44.4% of the measured homepages, and the document.write() method and the innerHTML property were also used to generate JavaScript code. Our AST-based structural analysis on various DJS instances further uncovers their usages with respect to programming language functionality. Our analysis indicates that in common cases, safe alternatives do exist for both the insecure JavaScript inclusion and insecure JavaScript dynamic generation. Since Web-based attacks have become more common and damaging in recent years, we

suggest website developers and administrators pay serious attention to these insecure JavaScript practices and use safe alternatives to avoid them.

ACKNOWLEDGMENTS

The authors sincerely thank anonymous reviewers for their valuable suggestions and comments.

REFERENCES

BALL, T. AND LARUS, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*, 4, 1319–1360.

BARTH, A., JACKSON, C., AND MITCHELL, J. C. 2008a. Robust defenses for cross-site request forgery. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 75–88.

BARTH, A., JACKSON, C., AND MITCHELL, J. C. 2008b. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*. 17–30.

BAXTER, I. D., YAHIN, A., MOURA, L., SANTANNA, M., AND BIER, L. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*.

BORTZ, A., BONEH, D., AND NANDY, P. 2007. Exposing private information by timing web applications. In *Proceedings of the International Conference on World Wide Web (WWW)*. 621–628.

CANALI, D., COVA, M., VIGNA, G., AND KRUEGEL, C. 2011. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the International Conference on World Wide Web (WWW)*. 197–206.

CERI, S., FRATERNALI, P., BONGIO, A., BRAMBILLA, M., COMAI, S., AND MATERA, M. 2002. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, San Fransisco, CA.

CERT. 2000. CERT advisory ca-2000-02 malicious html tags embedded in client web requests. http://www.cert.org/advisories/CA-2000-02.html.

CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND WANG, Y.-M. 2007. A systematic approach to uncover gui logic flaws for web security. In *Proceedings of the IEEE Symposium on Security and Privacy*. 71–85.

COVA, M., KRUEGEL, C., AND VIGNA, G. 2010. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the International Confeence on World Wide Web (WWW)*. 281–290.

CURTSINGER, C., LIVSHITS, B., ZORN, B., AND SEIFERT, C. 2011. Zozzle: Low-overhead mostly static javascript malware detection. In *Proceedings of the USENIX Security Symposium*.

DHAMIJA, R., TYGAR, J. D., AND HEARST, M. 2006. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 581–590.

DOM2EVENTS. 2012. Document object model (dom) level 2 events. http://www.w3.org/TR/DOM-Level-2-Events/events.html.

EGELE, M., WURZINGER, P., KRUEGEL, C., AND KIRDA, E. 2009. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Annual Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. 88–106.

EVALMDC. 2011. Eval-mdc. https://developer.mozilla.org/en/JavaScript/Reference/Global Objects/eval.

FALK, L., PRAKASH, A., AND BORDERS, K. 2008. Analyzing websites for user-visible security design flaws. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. 117–126.

FINIFTER, M., WEINBERGER, J., AND BARTH, A. 2010. Preventing capability leaks in secure javascript subsets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

FLANAGAN, D. 2006. *JavaScript: The Definitive Guide*. O'Reilly Media.

FLORENCIO, D. AND HERLEY, C. 2007. A large-scale study of web password habits. In *Proceedings of the International Conference on World Wide Web (WWW)*. 657–666.

FOGIE, S., GROSSMAN, J., HANSEN, R., RAGER, A., AND PETKOV, P. D. 2007. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress.

GUARNIERI, S. AND LIVSHITS, B. 2009. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium*.

HEILMANN, C. 2011. Unobtrusive javascript. http://www.onlinetools.org/articles/unobtrusivejavascript/.

HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., SAXENA, P., AND VEANES, M. 2011. Fast and precise sanitizer analysis with bek. In *Proceedings of the USENIX Security Symposium*.

HTML5COMM. 2012. HTML5: Communication. http://www.w3.org/TR/html5/comms.html.

HTML5SANDBOX. 2012. HTML5 <iframe>sandbox. http://www.w3schools.com/html5/att iframe sandbox.asp.

HTMLTIMERS. 2012. HTML timers. http://www.w3.org/TR/html5/timers.html.

HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. 2004. Securing web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web (WWW)*. 40–52.

JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. 2006. Protecting browser state from web privacy attacks. In *Proceedings of the International Conference on World Wide Web (WWW)*. 737–744.

JACKSON, C. AND WANG, H. J. 2007. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the International Conference on World Wide Web (WWW)*. 611–620.

JAKOBSSON, M. AND MYERS, S. 2006. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley-Interscience.

JIM, T., SWAMY, N., AND HICKS, M. 2007. Defeating script injection attacks with browser enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)*. 601–610.

JSAPI. 2011. JSAPI reference-MDC. https://developer.mozilla.org/en/JSAPI Reference.

JSON. 2011. JSON in javascript. http://www.json.org/js.html.

JSPRINCIPALS. 2011. JSPrincipals-MDC. http://developer.mozilla.org/en/JSPrincipals.

KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIC, N. 2006. SecuBat: A web vulnerability scanner. In *Proceedings of the International Conference on World Wide Web (WWW)*. 247–256.

KAPPEL, G., PROLL, B., REICH, S., AND RETSCHITZEGGER, W. 2006. *Web Engineering: The Discipline of Systematic Development of Web Applications*. John Wiley and Sons.

KICIMAN, E. AND LIVSHITS, V. B. 2010. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. *ACM Trans. Web 4*, 4, 13:1–13:52.

KIRDA, E., JOVANOVIC, N., KRUEGEL, C., AND VIGNA G. 2009. Client-side cross-site scripting protection. *Comput. Secur. 28*, 7, 592–604.

KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. 2011. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2595–2604.

KRISHNAMURTHY, B. AND WILLS, C. E. 2006. Cat and mouse: Content delivery tradeoffs in web access. In *Proceedings of the International Conference on World Wide Web (WWW)*. 337–346.

LAM, V. T., ANTONATOS, S., AKRITIDIS, P., AND ANAGNOSTAKIS, K. G. 2006. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 221–234.

LIVSHITS, B. AND CUI, W. 2008. Spectator: Detection and containment of javascript worms. In *Proceedings of the USENIX Annual Technical Conference*.

MENDES, E. AND MOSLEY, N. 2005. *Web Engineering*. Springer.

MEYEROVICH, L. AND LIVSHITS, B. 2010. ConScript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the IEEE Symposium on Security and Privacy*.

MOSHCHUK, A., BRAGIN, T., GRIBBLE, S. D., AND LEVY, H. M. 2006. A crawler-based study of spyware in the web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

MSDN. 2011. MSDN: InnerHTML property. http://msdn.microsoft.com/en-us/library/ms533897(VS.85).aspx.

MURUGESAN, S. AND DESHPANDE, Y. 2001. *Web Engineering: Managing Diversity and Complexity of Web Application Development*. Springer.

MXR. 2012. Mozilla cross-reference: Firefox 2 source code. http://mxr.mozilla.org/firefox2/.

NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. 2012. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 736–747.

ODA, T., WURSTER, G., VAN OORSCHOT, P., AND SOMAYAJI, A. 2008. SOMA: Mutual approval for included content in web pages. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 89–98.

POWELL, T. A., JONES, D. L., AND CUTTS, D. C. 1998. *Web Site Engineering: Beyond Web Page Design*. Prentice Hall.

PROVOS, N., MAVROMMATIS, P., RAJAB, M. B., AND MONROSE, F. 2008. All your iframes point to us. In *Proceedings of the USENIX Security Symposium*. 1–15.

RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. G. 2010. JSMeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*.

REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. 2006. BrowserShield: Vulnerability-driven filtering of dynamic html. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 61–74.

Reis, D. C., Golgher, P. B., Silva, A. S., and Laender, A. F. 2004. Automatic web news extraction using tree edit distance. In *Proceedings of the International Conference on World Wide Web (WWW)*. 502–511.

Richards, G., Gal, A., Eich, B., and Vitek, J. 2011a. Automated construction of javascript benchmarks. In *Proceedings of the ACMSIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 677–694.

Richards, G., Hammer, C., Burg, B., and Vitek, J. 2011b. The eval that men do - a large-scale study of the use of eval in javascript applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 52–78.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. 2010. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Rossi, G., Pastor, O., Schwabe, D., and Olsina, L. 2007. *Web Engineering: Modelling and Implementing Web Applications*. Springer.

Sans. 2007. SANS top-20 2007 security risks (2007 annual update). http://www.sans.org/top20/2007/.

Siliconforks. 2012. Parsing javascript with spidermonkey. http://siliconforks.com/doc/parsing-javascript-with-spidermonkey/.

Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. 2010. On the incoherencies in web browser access control policies. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Spidermonkey. 2012. Spidermonkey (javascript-c) engine. http://www.mozilla.org/js/spidermonkey/.

Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R. A., Kruegel, C., and Vigna, G. 2009. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 635–647.

Suh, W. 2005. *Web Engineering: Principles and Techniques*. IGI Publishing.

Symantec. 2008. Symantec internet security threat report volume XIII: April, 2008. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. 2007. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

W3cdom. 2011. W3C document object model. http://www.w3.org/DOM.

Wang, H. J., Fan, X., Howell, J., and Jackson, C. 2007. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 1–16.

Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. 2009. The multi-principal os construction of the gazelle web browser. In *Proceedings of the USENIX Security Symposium*. 417–432.

Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S. T. 2006. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Wassermann, G. and Su, Z. 2008. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 171–180.

Welty, C. A. 1997. Augmenting abstract syntax trees for program understanding. In *Proceedings of the International Conference on Automated Software Engineering*.

Wikijs. 2011. Javascript. http://en.wikipedia.org/wiki/JavaScript.

Wikisop. 2011. Same origin policy. http://en.wikipedia.org/wiki/Same origin policy.

Wikixss. 2011. Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site scripting.

Willison, S. 2005. 24 ways: Don't be eval(). http://24ways.org/2005/dont-be-eval.

Wot. 2012. Safe browsing tool—WOT (web of trust). http://www.mywot.com/.

Xhr. 2011. XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest/.

Yang, W. 1991. Identifying syntactic differences between two programs. *Softw. Pract. Exper. 21*, 7(1999), 739–755.

Yu, D., Chander, A., Islam, N., and Serikov, I. 2007. Javascript instrumentation for browser security. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 237–249.

Yue, C. 2012. Preventing the revealing of online passwords to inappropriate websites with login inspector. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*. 67–81.

Yue, C. and Wang, H. 2009. Characterizing insecure javascript practices on the web. In *Proceedings of the International Conference on World Wide Web (WWW)*. 961–970.

Yue, C. and Wang, H. 2010. BogusBiter: A transparent protection against phishing attacks. *ACM Trans. Internet Technol. 10*, 2, 1–31.

YUE, C., XIE, M., AND WANG, H. 2010. An automatic http cookie management system. *J. Comput. Netw. 54*, 13, 2182–2198.

ZALEWSKI, M. 2012. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main.

ZHAI, Y. AND LIU, B. 2005. Web data extraction based on partial tree alignment. In *Proceedings of the International Conference on World Wide Web (WWW)*. 76–85.

ZHAO, R. AND YUE, C. 2013. All your browser-saved passwords could belong to us: A security analysis and a cloud-based new design. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*.

Received February 2011; revised November 2012; accepted February 2013