How to Train Your Browser: Preventing XSS Attacks Using Contextual Script Fingerprints

DIMITRIS MITROPOULOS, Columbia University
KONSTANTINOS STROGGYLOS and DIOMIDIS SPINELLIS, Athens University of Economics and Business
ANGELOS D. KEROMYTIS, Columbia University

Cross-Site Scripting (XSS) is one of the most common web application vulnerabilities. It is therefore sometimes referred to as the "buffer overflow of the web." Drawing a parallel from the current state of practice in preventing unauthorized native code execution (the typical goal in a code injection), we propose a script whitelisting approach to tame JavaScript-driven XSS attacks. Our scheme involves a transparent script interception layer placed in the browser's JavaScript engine. This layer is designed to detect every script that reaches the browser, from every possible route, and compare it to a list of valid scripts for the site or page being accessed; scripts not on the list are prevented from executing. To avoid the false positives caused by minor syntactic changes (e.g., due to dynamic code generation), our layer uses the concept of contextual fingerprints when comparing scripts.

Contextual fingerprints are identifiers that represent specific elements of a script and its execution context. Fingerprints can be easily enriched with new elements, if needed, to enhance the proposed method's robustness. The list can be populated by the website's administrators or a trusted third party. To verify our approach, we have developed a prototype and tested it successfully against an extensive array of attacks that were performed on more than 50 real-world vulnerable web applications. We measured the browsing performance overhead of the proposed solution on eight websites that make heavy use of JavaScript. Our mechanism imposed an average overhead of 11.1% on the execution time of the JavaScript engine. When measured as part of a full browsing session, and for all tested websites, the overhead introduced by our layer was less than 0.05%. When script elements are altered or new scripts are added on the server side, a new fingerprint generation phase is required. To examine the temporal aspect of contextual fingerprints, we performed a short-term and a long-term experiment based on the same websites. The former, showed that in a short period of time (10 days), for seven of eight websites, the majority of valid fingerprints stay the same (more than 92% on average). The latter, though, indicated that, in the long run, the number of fingerprints that do not change is reduced. Both experiments can be seen as one of the first attempts to study the feasibility of a whitelisting approach for the web.

CCS Concepts: • **Security and privacy → Browser security; Web application security;**

Additional Key Words and Phrases: Protection mechanisms, JavaScript, XSS, JavaScript engine

---

This work was supported by NSF Grant No. CNS-13-18415. Any opinions, fundings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of the U.S. Government or the NSF.

Authors' addresses: D. Mitropoulos (corresponding author) and A. D. Keromytis, Computer Science Department, Columbia University in the City of New York, 1214 Amsterdam Avenue, New York, NY 10027; email: {dimitro, angelos}@cs.columbia.edu; K. Stroggylos and D. Spinellis, Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athens 104 34, Greece; email: {circular, dds}@aueb.gr.

DOI: http://dx.doi.org/10.1145/2939374

ACM Reference Format:
Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D. Keromytis. 2016. How to train your browser: Preventing XSS attacks using contextual script fingerprints. ACM Trans. Priv. Secur. 19, 1, Article 2 (July 2016), 31 pages.
DOI: http://dx.doi.org/10.1145/2939374

1. INTRODUCTION

For the past several years, Cross-Site Scripting (XSS) attacks have been topping the lists of the most dangerous vulnerabilities published by the MITRE Corporation,[1] Open Web Application Security Project (OWASP),[2] and other organizations. In addition, since their first appearance, they have plagued several popular websites, including Twitter[3] and Facebook,[4] and have been used as a stepping stone even for taking over full servers.[5]

An XSS vulnerability is manifested when a web application accepts and redisplays data of uncertain origin without appropriate validation and/or filtering [Stuttard and Pinto 2011]. One of the most common XSS attack vectors is JavaScript. Attackers are motivated by the fact that JavaScript is executed as a browser component and enables access to valuable objects within the browser. By exploiting an XSS vulnerability, an attacker can bypass security mechanisms that are integrated in the browser, such as the same origin policy [Saiedian and Broyle 2011] and a sandbox mechanism [Dhawan and Ganapathy 2009]. Consider an attacker who posts data that will inject a well-hidden script into a dynamically generated page of a vulnerable website. In this manner, the attacker can trick a user into downloading this script from this page in order to steal the user's cookies or manipulate the stored information of a form for malicious purposes. The injected script is confined by a sandboxing mechanism and conforms to the same origin policy, but it still violates the browser's security [De Groef et al. 2012].

Despite the numerous countermeasures that are being introduced, attackers seem to find new ways to bypass existing defense mechanisms [Reis et al. 2007; Yu et al. 2007; Jim et al. 2007; Nadji et al. 2006; Bisht and Venkatakrishnan 2008] by using a variety of techniques [Weissbacher et al. 2015; Jin et al. 2014; Stock et al. 2014; Heiderich et al. 2012; Athanasopoulos et al. 2009; Bojinov et al. 2009]. Moreover, some previous approaches are either impractical for deployment in certain environments [Van Acker et al. 2011; Oda et al. 2008; Phung et al. 2009; Nanda et al. 2007] or impose a non-negligible overhead [Agten et al. 2012; Nanda et al. 2007; Stock et al. 2014]. These are some of the main reasons why XSS prevention continues to draw the attention of researchers and practitioners alike. We further analyze such limitations in Section 2.

To protect web users from XSS attacks based on JavaScript, we propose a *training approach* where every benign script of a webpage is backed by a *fingerprint*. Training techniques are based on the ideas of Denning's original intrusion detection framework [Denning 1987]. Specifically, a training mechanism learns all valid benign code statements during a training phase. Then, only those statements will be recognized and approved for execution during production. Figure 1 includes an activity diagram illustrating the basic steps taken by our scheme. Our work can also be seen as an attempt to apply *application whitelisting* [Beechey 2010] to the web domain.

To create a fingerprint we use a blend of *elements*. Some of these elements depend on the script that is about to be executed. We chose to use specific script elements rather than the whole script because of the *dynamic nature* of JavaScript (using the

[1]http://cwe.mitre.org/top25/.
[2]https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.
[3]http://www.theguardian.com/technology/blog/2010/sep/21/twitter-hack-explained-xss-javascript.
[4]https://www.facebook.com/help/757846550903291.
[5]https://blogs.apache.org/infra/entry/apache_org_04_09_2010.

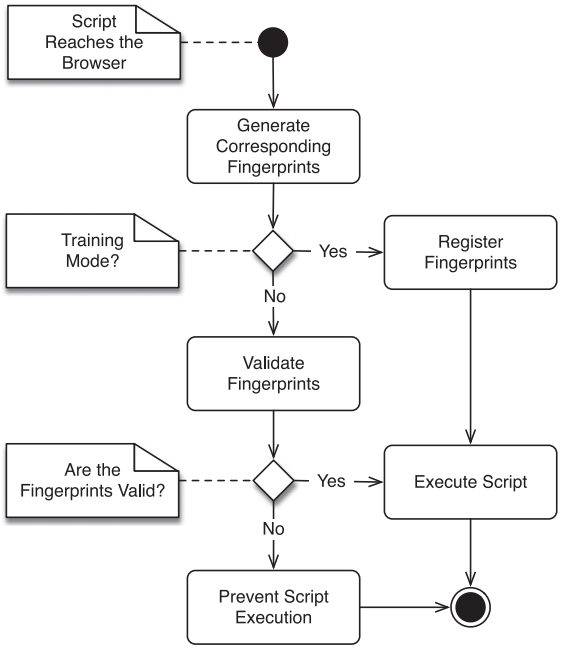

Fig. 1. An activity diagram incorporating the basic steps of our XSS-prevention scheme. The steps are typical for a training scheme [Denning 1987; Weissbacher et al. 2015; Wurzinger et al. 2009; Johns et al. 2008; Bisht and Venkatakrishnan 2008]. Two of our main contributions involve (a) the way the fingerprints are generated and (b) the diverse elements that the fingerprints contain.

whole script would be meaningful for websites that preserve only static scripts, something that rarely happens nowadays). In this way, we can apply our policies during script interpretation/execution. Other elements are extracted from the execution context and include the type of the script, the URL that triggered the execution, and more. Furthermore, *runtime checks* are applied to prevent scripts from interacting with non-legitimate domains. This is achieved by generating additional fingerprints for the domains that are dynamically referenced by the scripts during execution (e.g., URLs passed as arguments to a JavaScript function).

The combination of different elements—a process that to the best of our knowledge is unique to our approach—is a critical aspect of our "defense-in-depth" approach because it facilitates the prevention of a plethora of XSS attacks, as we demonstrate in this article. Notably, by incorporating different elements we can prevent the mounting of attacks using legitimate scripts in a malicious context, as is done in a mimicry attack [Wagner and Soto 2002]. Note, though, that there are attacks of this kind that can bypass our proposed countermeasure. We describe such cases in Subsection 5.1.3.

The fingerprint generation involves a transparent security layer that we call *nsign*. nsign wraps the browser's JavaScript engine and automatically collects all elements required for generating and verifying fingerprints. All fingerprints are created on the server side by the website's administrators or developers with the help of nsign. Then, the browser on the client's side retrieves the fingerprints in a secure way. Finally, our layer attempts to match the retrieved fingerprints with the ones generated while browsing. Our scheme was designed based on the following principles:

(1) **Efficient Script Detection.** We designed our layer to deal with all scripts that reach a browser, from every possible route. To achieve this we wrapped all entry

points of the JavaScript engine that are used by the browser to execute the scripts contained in or referenced by a webpage.

(2) **Flexibility.** Fingerprints can be enriched with new elements if needed, making the approach more robust. In the same manner, elements can be removed from the fingerprints to reduce overhead, but this makes the scheme more vulnerable to attacks. For every element that we have included in our current implementation, we provide a corresponding justification and identify what kind of attacks we prevent by making this design choice. In a similar way, we point out the choices that could lead to a false negative.

(3) **Effectiveness.** We aimed for an effective countermeasure. To evaluate this, we tested it successfully against an extensive array of attacks that were performed on more than 50 real-world vulnerable applications. During our tests we did not encounter any false alarms.

(4) **Unaffected Web User Experience.** We aimed for a transparent validation layer with low overhead. While browsing eight websites that make heavy use of JavaScript (e.g., Facebook and Twitter), we found that our layer imposed an average overhead of 11.1% on the JavaScript engine. In addition, on our testbed, and for all tested sites, the overhead introduced by our layer in the round-trip time required for downloading a script was negligible ($<0.05\%$).

Our scheme follows the trend of client-side protection mechanisms. As we will see in Section 7, 17 of 26 of the current frameworks detect XSS attacks at the client side. This is mainly because "in its core, XSS is a client-side security problem" [Stock et al. 2014]. Many client-side mechanisms, though (including ours), are not always easy to deploy.

As we mentioned earlier, during training, all fingerprints must be created on the server side by the website's administrators. This implies that the effectiveness of our approach depends on the creation of all corresponding fingerprints. However, this can be achieved to a significant extent by the increased adoption of test-driven development and the use of automated testing. To help administrators collect valid fingerprints automatically we have developed a *fingerprint generation module* based on an existing testing framework-see Subsection 3.3. Note also that this training could run without a website's cooperation. In essence, a trusted third party (e.g., Google) could create these fingerprints for a site, and users could fetch them from that third party instead of the website itself.

Finally, when script elements are altered or new scripts are added on the server-side, a new training phase is required. We performed both a short-term and long-term experiment to examine the *temporal aspect* of contextual fingerprints. Our results, presented in the "Evaluation" Section (5), suggest that in a short period of time (10 days) and for seven of eight websites, the majority of fingerprints stay the same (more than 92% on average). However, after a long period of time (more than 6 months), the number of fingerprints that remained unchanged is considerably reduced for all the websites. Both experiments can be seen as one of the first attempts to study the feasibility of a whitelisting approach for the web, and, to the best of our knowledge, a study like this has not been done before.

2. LIMITATIONS IN EXISTING APPROACHES

In this section, we identify specific limitations in the existing approaches and discuss how we address them. Such limitations involve effectiveness, deployment, and overhead issues. We further describe existing approaches in Section 7.

2.1. Effectiveness Issues

For a typical XSS example that involves JavaScript, consider a webpage that prints the value of a query parameter (`query`) from the page's URL as part of the page's content without escaping the value. Attackers can take advantage of this and inject an `iframe` tag into the page to steal a user's cookie and send it via an image request to a website under their control (malicious.com). This could be achieved by including the following link to the malicious website (or sending it via phishing email) and inducing the user to click on it:

```
http://example.com/vulnerable.html?query=<iframe src="javascript:document.
     body.innerHTML=+'<img src=\"http://malicious.com/?c='+
     encodeURIComponent(document.cookie)+'\">'"></iframe>
```

Policy enforcement frameworks like BrowserShield [Reis et al. 2007] do not fully cover cases like the above. Even though they allow developers to decide if they will disable `iframe` tags or not, this is impractical because such features are quite popular and widely used. If developers choose to use them, then these frameworks cannot define policies that restrict the behavior of third-party scripts introduced by such features. Thus, they would be vulnerable to attacks like the above.

As we will see in Section 7, the majority of the existing approaches require the presence of a Document Object Model (DOM) tree to prevent an attack. This renders them vulnerable to attacks like Cross-Channel Scripting (XCS) [Bojinov et al. 2009], which is an XSS variation, or attacks that do not utilize web channels to inject JavaScript code into the user's browser [Barth et al. 2009; Athanasopoulos et al. 2010]. In an XCS attack, a malicious user may utilize the Simple Network Management Protocol (SNMP) as an attack vector. For example, there are several Network-Attached Storage (NAS) devices that allow users to upload files via the Server Message Block (SMB) protocol. An attacker could upload a file with a filename that contains a well-crafted script. When a legitimate user connects over a web channel to the device to browse its contents, the device will send through an HTTP response the list of all filenames, including the malicious one that is going to be interpreted as a valid script by the browser. An attack via a non-web channel could involve a malicious user that embeds a script within a PostScript file, uploads it as a valid document, and then uses it to trigger the attack.

Our layer detects every script that reaches the browser from every possible route, as we show in Subsection 4.1. In this manner it can prevent attacks like the ones described above.

Another way to bypass some of the existing approaches is to perform a mimicry attack [Wagner and Soto 2002]. With such an attack, a malicious user can execute legitimate scripts but not as intended by the original design of the developers. Consider a training mechanism like XSS-GUARD [Bisht and Venkatakrishnan 2008], which employs parse-tree validation. During training, the scheme maps legitimate scripts to HTTP responses. During production, it retrieves for every script included in a response its parsed tree and checks if it is one of those previously mapped to this response. Apart from the comparison of the parsed trees, XSS-GUARD checks also for an exact match of lexical entities. However, string literals are not compared literally, which can lead to false negatives. For instance, observe the JavaScript code presented in Figure 2 that implements a real-world banner rotator.[6] Every time the banner runs, it creates a value (`core`) that depends on the current date and the length of the array that contains the references of the various images to be displayed. Then, based on this value, it

[6]http://www.hotscripts.com/listing/banner-rotator/.

```
<script type="text/javascript">
var currentdate = 0;
var core = 0;

function initArray() {
 this.length = initArray.arguments.length;
 for (var i = 0; i < this.length; i++) {
    this[i] = initArray.arguments[i];
 }
}

var link = new initArray(
       "http://example.com",
       "http://foo.com"
);

var image = new initArray(
       "http://example.com/mini.gif",
       "http://foo.com/small.gif"
);

var currentdate = new Date();
var core = currentdate.getSeconds() % image.length;
var ranlink = link[core];
var ranimage = image[core];

src = '<a href="' + rlink + '"><img src="'
       + ranimage + '" border="0"></a><br>';
$("#container").html(src);
</script>
```

Fig. 2. Real-world banner rotator script.

shows a specific image to a user by using a jQuery function. In a vulnerable website that allows users to post data and contains this banner rotator, attackers could create and store the same script but with references to tiny images hosted on a web server that is maintained by them (in essence, change the values of `link` and `image` arrays). In this way, they could retrieve the IP addresses of the users that visit the vulnerable site. Mimicry attacks may also affect mechanisms that do not examine the script's location inside the web document like Browser-Enforced Embedded Policies (BEEP) [Jim et al. 2007], Document Structure Integrity (DSI) [Nadji et al. 2006], Secure Web Application Proxy (SWAP) [Wurzinger et al. 2009], and Cross-Site Scripting Detection System (XSSDS) [Johns et al. 2008], as described by Athanasopoulos et al. [2009, 2010].

By associating elements coming from a script with elements coming from the execution context and by performing specific runtime checks, nsign can defend against attacks that try to masquerade their behavior as a benign one.

Furthermore, current client-side XSS filters like Firefox's NoScript[7] plugin, Internet Explorer's XSS Filter[8] and Chrome's XSS Auditor [Bates et al. 2010] suffer from several shortcomings (e.g. large number of false positives) as explained in detail by Stock et al. [2014]. We discuss how our scheme could produce false positives in Subsection 5.2.2.

2.2. Deployment and Overhead Issues

The majority of client-side mechanisms require modifications both on the server and at the client. Hence, it would be difficult to be adopted by both browser vendors and

[7]https://noscript.net/.
[8]http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

developers, an issue that applies to our approach, too. Note, though, that there are exceptions (e.g., Blueprint [Louw and Venkatakrishnan 2009]) that enforce policies at the client side by embedding a library in the server's response to the client.

Extensive modifications in the application's source can be a reason that can render a mechanism difficult to use. Approaches that require developers to define specific security policies on the server side (e.g., the Content Security Policy (CSP) [Stamm et al. 2010], WebJail [Van Acker et al. 2011], JSand [Agten et al. 2012], and Same Origin Mutual Approval (SOMA) [Oda et al. 2008]) are such examples. For instance, consider the case of CSP, a policy enforcement approach further discussed in Section 7. In CSP, policies are inserted via response headers or meta tags. If the developers employ this approach and their JavaScript code involves code constructs like the `eval` function and inline event handlers, then they must set policies that include the keywords `unsafe-eval` and `unsafe-inline`, respectively. Contrary to such schemes, developers are not required to modify the code of a website to use our approach.

Furthermore, many existing mechanisms impose a non-negligible overhead [Agten et al. 2012; Nanda et al. 2007; Stock et al. 2014; Wurzinger et al. 2009; Louw and Venkatakrishnan 2009; Nanda et al. 2007]. Our layer exhibits lower overhead than the majority of previous work, as we show in Subsection 5.2.3.

3. APPROACH

Every script included in a webpage that is displayed within a browser will be processed by the browser's JavaScript engine. This engine is responsible for the interpretation and ultimately execution of the client-side scripts. Our approach involves a thin validation layer called nsign. nsign wraps all entry points of the JavaScript engine (typically, all the API calls used by the browser to execute the scripts contained in or referenced by a website) and can differentiate benign scripts from injected ones by associating benign scripts with their origins and all the external URLs they reference.

nsign can detect JavaScript-based XSS attacks, ranging from the standard persistent [Stuttard and Pinto 2011], non-persistent [Stuttard and Pinto 2011], or DOM-based attacks [Stock et al. 2014] to the sophisticated attacks described in Subsection 2.1 (e.g., XCS attacks and mimicry attacks) as we describe under "Evaluation" (Section 5). Nevertheless, there is a subset of mimicry attacks that can bypass our mechanism. We discuss such attacks under "Security Analysis" (Subsection 5.1.3). nsign does not prevent "scriptless" [Heiderich et al. 2012] and HTML injection attacks [Reis et al. 2007].

3.1. nSign

To prevent an attack, nsign follows the steps of Algorithm 1. First, it goes through a fingerprint generation phase. During this phase, every benign script is mapped to an identifier that we call a "*script fingerprint*" (Algorithm 1, line 2, variable `f`). This fingerprint is then associated with a set of "*URL fingerprints*" that are generated from the domain part of URLs extracted from the arguments passed to the script during execution (Algorithm 1, line 3, variable `u`). All fingerprints are stored in an auxiliary table (Algorithm 1, lines 5 and 6).

During production mode, the steps of nsign are exactly the same as during the fingerprint generation mode until nsign derives the fingerprint for the script. At that point, nsign checks if a corresponding script fingerprint exists in the aforementioned table (Algorithm 1, line 10). If there is, then the script initially passes as benign. Then, while executing it, it checks if the script arguments reference any unexpected URLs (Algorithm 1, line 11). If no corresponding fingerprint is found or an unregistered URL fingerprint is encountered, then the browser is under attack. In such a case, nsign can halt the execution or forward an alarm to the website's administrator.

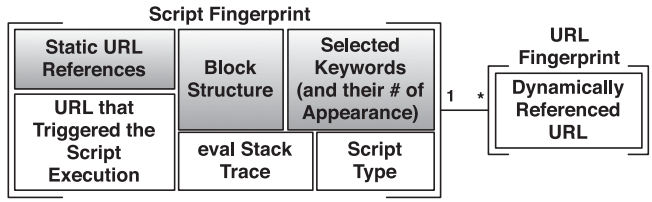

Fig. 3. Script and URL fingerprints. Elements in gray are extracted from the script (Algorithm 1, line 2, function `AnalyzeScript`), while the other elements are coming from the execution context (Algorithm 1, line 2, function `AnalyzeExecutionContext`).

ALGORITHM 1: Validation Layer Algorithm

```
1  function SecurityLayer(script)
2  f = AnalyzeScript(script) + AnalyzeExecutionContext();
3  u[ ] = GetReferencedURLFingerprints(script);
4  if fingerprintGenerationMode = true then
5  |   RegisterScriptFingerprint(f);
6  |   RegisterReferencedURLFingerprints(f,u);
7  |   res = ExecuteScript(script);
8  else
9  |   res = false;
10 |   if ValidateScriptFingerprint(f) then
11 |   |   if CheckReferencedURLFingerprints(f,u) then
12 |   |   |   res = ExecuteScript(script);
13 |   |   else
14 |   |   |   LogPossibleAttack(script);
15 |   |   end
16 |   else
17 |   |   LogPossibleAttack(script);
18 |   end
19 end
20 return res;
21 end function
```

3.2. Fingerprint Generation

In this subsection we describe how script and URL fingerprints are created. For every element that we include, we provide a corresponding justification and discuss what kind of attacks we prevent by making this design choice and how we avoid false positives. In the same manner, we point out the choices that may lead to a false negative. Figure 3 illustrates the fingerprints, their corresponding elements, and the way they are associated.

3.2.1. Script Fingerprints. nsign is designed to create fingerprints for every script that reaches the JavaScript engine of a browser. A script can be (a) embedded, which refers to code contained within HTML `<script> ...</script>` tags, code included in event handlers, and others (for more details see Subsection 4.1); (b) external, which refers to external scripts referenced by a webpage using the `src` attribute of the `script` tag, scripts included in external documents referenced by the page (e.g., jQuery and AngularJS), or scripts that reach the engine through a non-web channel; and (c) fed to `eval`, which refers to code that has been passed as an argument to the `eval` method for evaluation and execution.

```
function(qoptions){
   var e=(typeof(enc)=='function')?"n":"s";
   var r=qc.qcrnd();
   var sr='',qo='',ref='',je='u',ns='1';
   var qocount=0;qc.qad=0;
   if(typeof qc.qpixelsent=="undefined"){
       qc.qpixelsent=new Array();
   }
}
```

```
(){(())()(){()}}
  function: 1
    var: 4
  typeof: 2
    if: 1
  Array: 1
```

Fig. 4. A real-world JavaScript function and its block structure, the selected keywords, and their # of appearance. Script source: BBC.com.

When a script reaches the engine, nsign detects it and analyzes it (Algorithm 1, line 2, function `AnalyzeScript`) to extract and add to the fingerprint the following key elements:

- **Block Structure.** To obtain this element, nsign strips everything within the script except for all the braces and parentheses while retaining their order (see Figure 4).
- **Selected Keywords and Their # of Appearance.** This element involves the aggregated counts of selected JavaScript keywords, including all the reserved keywords mentioned in the following JavaScript reference [MDN 2015], for instance, words used to define control-flow statements (e.g., `while`, `for`), global functions (e.g. `eval`, `encodeURI`), and fundamental objects (e.g., `Function`, `Object`).
- **Static URL References.** nsign collects all the static references of remote URL addresses that may exist within a script. For example, if a script contains a static reference to a remote URL address, the domain name part of this reference will be included in the script fingerprint.

By including the static URL references that exist within a script, nsign can contain an attack where a malicious user includes in the injected script an element stored on another server (recall the example with the banner rotator script in Subsection 2.1). Our layer would prevent such an attack since the references to the remote images would lead to a different script fingerprint.

We chose not to include the whole script in our fingerprint for the following reason: Consider a website or service that generates scripts dynamically for its users based on some identity information, such as a service that generates detailed statistics about a website's traffic or serves advertisements. The generated scripts do not differ in structure but the values of certain class and variables names they contain are unique for each user—The Open Web Analytics[9] (OWA) software provides this type of functionality through its API.[10] This customization would generate a different fingerprint for essentially the same script and produce false alarms. However, our implementation can be easily modified to take the whole script into account when generating the fingerprint.

[9]http://www.openwebanalytics.com/.
[10]http://wiki.openwebanalytics.com/index.php?title=Tracker.

ALGORITHM 2: Traversing the Invocation Stack Trace That Led to the `Eval` Function

```
1 function GetStackTrace(frame)
2 repeat
3 |   frame = frame.caller;
4 |   s = getFrameDetails(frame);
5 |   stackTrace = stackTrace + s;
6 until frame != bottomFrame;
7 return stackTrace;
8 end function
```

To create a script fingerprint, nsign also collects elements coming from the script's execution context (Algorithm 1, line 2, function `AnalyzeExecutionContext`). Such elements include the following:

- **URL That Triggered the Script Execution.** The codebase of a script may be either the URL of the webpage containing the script (in case it is embedded) or the URL of the external file referenced by the webpage. The domain name is the only usable part of the script's codebase. This is because many websites reference dynamic pages that produce JavaScript code or generate embedded script code dynamically. Thus the path and query segment of the URL is not necessarily static.
- **Type of the Script.** The type can be classified in the three basic categories that we identified in the beginning of this subsection, namely (a) embedded, (b) external, and (c) fed to `eval`.

By including the above elements, nsign can prevent attacks like the one presented in the beginning of Subsection 2.1 because the malicious.com site will trigger a script execution that will lead to non-recorded fingerprint. Nevertheless, by choosing to leave out the path and query segment of the URL, this may lead to a false negative as we discuss in Subsection 5.1.3. In the case of `eval`, nsign also includes one more element:

- **eval Stack Trace.** This is typically the invocation stack trace that led to the `eval` function. To obtain this element we traverse the JavaScript call stack and retrieve the calling entity (location and function name) of each JavaScript stack frame (see Algorithm 2).

By incorporating the `eval` stack trace as a differentiating factor, nsign can detect attacks that utilize this method. For instance, `eval` has been extensively used to bypass the various client-side XSS filters [Lekies et al. 2014]. In addition, recall the technique employed by Sammy Kamkar in his MySpace attack [Kamkar 2005]. Such an attack would span a non-recorded fingerprint produced from (a) the script fed to `eval` and (b) the fact that `eval` was invoked from a non-expected stack trace. After retrieving and combining all the above elements, nsign applies a hash function to them and generates the final script fingerprint.

3.2.2. URL Fingerprints. As we have stated earlier, one of our objectives is to associate a script with all the URLs it references. Until now, this is partially done by extracting the domain name part of any URL statically referenced within a script. However, referenced URLs are not always static parts of the script code; they can be assembled by simple string operations like concatenations of variables or can be retrieved through the DOM. They might even be formed by processing data retrieved via Asynchronous JavaScript and XML (AJAX) [Johnson et al. 2007] calls, or they can be contained in

JavaScript Object Notation (JSON)-with-padding (JSONP)[11] responses. Eventually, these URLs may be used as string arguments to a defined function.

A key objective of nsign is to scan all the string arguments passed to JavaScript functions for external URLs (Algorithm 1, line 3, function `GetReferencedURLFingerprints`). This allows nsign to capture any *hosts dynamically referenced by a script* and associate them with the fingerprint that has been generated for the script being executed. Thus, a script fingerprint is associated to a set of URL fingerprints (Algorithm 1, line 6, function `RegisterReferencedURLFingerprints`) that represent the domains that the script may dynamically access at runtime. During runtime we can monitor the values of string arguments passed to JavaScript functions and prevent the script from interacting with non-legitimate domains. For instance, consider the following legitimate code:

```
var message = "Thank you for your visit!";
var base = "http://www.foo.com/";
var path = "thepathto/mini.gif";
var url = base + path;
```

Upon script execution, variable `url` contains a statically referencedURL that will be included in the script fingerprint. Interestingly, by performing the following mimicry attack:

```
var message = "Thank you for your visit!";
var base = "http:";
var path = "//bot.net/xss?s=http://www.foo.com";
var url = base + path;
```

the attacker's script could lead to a legitimate script fingerprint, as the script's block structure is unaffected (no braces or parentheses have been used), there are no new keywords, and the statically referenced URL is also included in the script. However, the malicious URL will eventually be assembled and passed as an argument to a JavaScript function during runtime. Therefore, such an attack would also be prevented since there will be no matching URL fingerprint for the injected URL.

Contrary to the elements described in the previous subsection, we did not include these references when generating script fingerprints because they do not depend on the script's code. They represent domains that the script may access during its execution. Therefore, their nature is purely dynamic. These references go through the same hashing algorithm used for script fingerprint generation (hereinafter, script fingerprints and URL fingerprints will be referred to as "fingerprints").

3.3. Fingerprint Propagation and Retrieval

To collect all valid fingerprints, the administrator of the protected web site should execute all possible scripts on the server side before the site goes in production mode. This is, of course, a known hard problem in web application testing, but, luckily, a number of testing frameworks have been created to overcome it [Pacheco et al. 2007; Saxena et al. 2010; Artzi et al. 2011]. We have developed a simple fingerprint generation module based on the *Crawljax* testing framework [Mesbah and van Deursen 2009; Mesbah et al. 2012]. Despite its name, Crawljax is not a web crawler but the implementation of an approach developed to test all AJAX interfaces of a web application [Mesbah and van Deursen 2009] and, consequently, execute all JavaScript code fragments of the web site. Crawljax displays a high-percentage coverage and has been used in similar occasions by researchers [Bezemer et al. 2009; Roest et al. 2010; Mesbah and Prasad 2011]. We further describe our module in Subsection 4.3.

[11] http://www.json-p.org/.

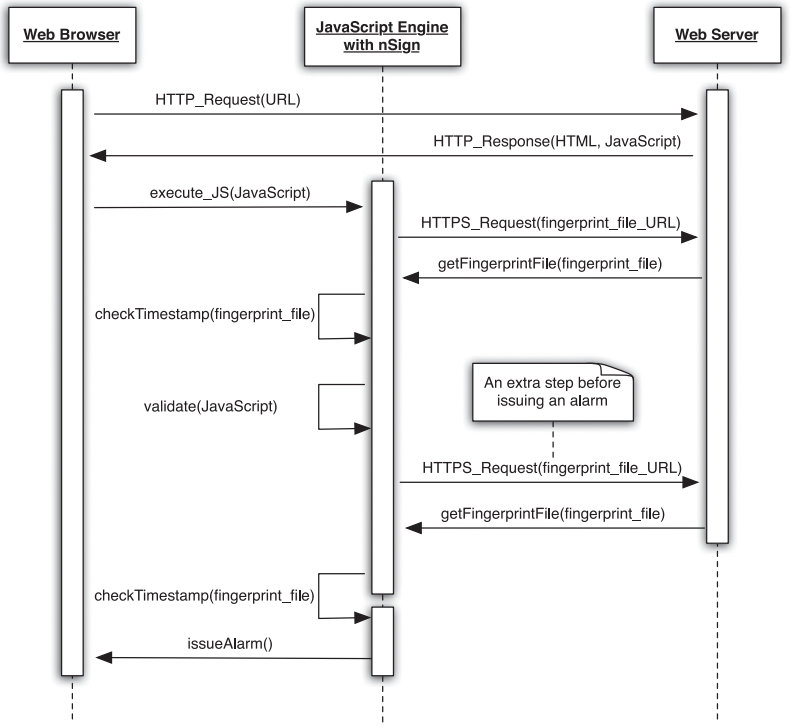

Fig. 5. Fingerprint check and retrieval during a new session.

When the website is ready for deployment in the production environment, and after all its functionality has been exercised in the fingerprint generation mode, all fingerprints are are stored locally on the server in a database. This serves the following purpose: exporting all valid fingerprints to a plain text file, along with a timestamp. This file, hereafter called the "fingerprint file," should be placed on the root folder of the website, so nsign can retrieve it during production mode over a secure channel.

In production mode, this database serves as a local cache of valid fingerprints on the client side, in order to avoid requesting the fingerprint file from a website every time the user visits it. The request for the fingerprint file occurs only once per visited website and remains cached for as long as the browser is running. The fingerprint retrieval is illustrated in Figure 5. After retrieving the fingerprint file, the timestamp that accompanies it is checked to make sure that the local database contains the latest version of valid fingerprints for the given website. If the timestamp of the local fingerprint file is older than the one on the server, then it means that a new set of fingerprints has been generated as a result of an application modification. In this case, the local cache is updated. While the browser is running, the set of valid fingerprints is cached in a hashed set in memory in order to allow for efficient fingerprint validation. Note that the location where the fingerprints are cached is an implementation detail; we could use the browser's cache.

However, the fact that the fingerprint file is requested once per visited website and as the browser is running could lead to a false-positive alert. Specifically, if the fingerprint file is updated at the server side, the clients that are currently visiting the site will have an outdated fingerprint file. Hence, nsign may generate a fingerprint that will not pass as legitimate. To avoid this, our method takes one more step before raising

an alarm: it checks again whether there is an updated fingerprint file on the server. If there is, nsign checks again if the fingerprint exists in the updated fingerprint file before alerting the user (see Figure 5).

4. IMPLEMENTATION

We have implemented our layer as a module integrated in the Mozilla SpiderMonkey JavaScript engine.[12] In particular, we instrumented the SpiderMonkey API methods that are used by the browser as entry points to the engine, by wrapping the implementation of each one so it goes through our validation layer. Other mechanisms employed by the engine such as Just In Time (JIT) compilation or caching of compiled scripts are neither affected by nor affect our layer, since it operates on an intermediate level between the API and core of the engine.

4.1. Intercepting JavaScript

The entry points of a JavaScript engine are the API calls used by the browser to execute the scripts contained in or referenced by a webpage. In order to improve performance, applications that use a JavaScript engine try to minimize script compilation time by holding references to compiled scripts and invoking them multiple times with different argument values. We therefore needed to instrument API methods that either take as input a script as a string or execute a script that has already been compiled by the engine. JavaScript is also used internally by Firefox in order to implement various functions mostly related to the user interface. To avoid processing such scripts, we devised a mechanism for filtering out "internal" calls to SpiderMonkey by examining the security principals and the script origin associated with each API call. By identifying all possible entry points, we managed to detect all scripts that reach the engine including scripts that may reach the engine through channels like the `document.write` method and documents loaded within `iframes`. Although we identified the entry point of every JavaScript execution, there is a case that must be handled with special care: The code fed as an argument to the `eval` function may be stripped out during code analysis. Therefore, we also wrapped the `eval` function implementation in order to perform code analysis on its input.

A potential source of false alarm could occur if we did not consider the JSON data included in scripts used by various web applications. However, SpiderMonkey distinguishes internally all JSON data coming from JavaScript code. This came in handy as a shortcut because our layer does not intend to generate fingerprints for these kinds of data. Even if an attacker manages to pass a script through JSON data (as described by Stock et al. [2014]), the script will finally reach the engine in order to be executed and a script fingerprint will be generated. As this fingerprint will not exist in the fingerprint file, the malicious script execution will be blocked.

Although the word JavaScript may bring to mind features such as DOM objects, AJAX calls (`XMLHttpRequest` objects), and event handlers like `onclick`, these features are not part of SpiderMonkey or most JavaScript engines. They are typically provided by the applications using the engine by exposing some of their objects and functions to JavaScript code.[13] For instance, the implementation of `XMLHttpRequest` in Mozilla Firefox is located in the `nsXMLHttpRequest` class. However, we were able to intercept `XMLHttpRequests` initiated by scripts, because SpiderMonkey uses an API method to call back to a `nsXMLHttpRequest` object by wrapping references to the corresponding native functions provided by Firefox in `JSObject` structures. In a similar manner, we were able to intercept the execution of scripts that are triggered by DOM event handlers.

[12] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[13] https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_User_Guide.

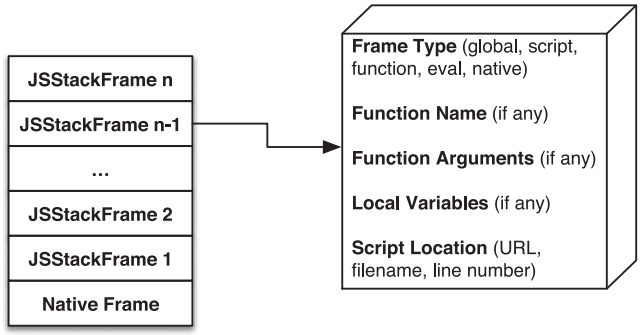

Fig. 6. JavaScript execution stack and stack frame elements.

4.2. Extracting Elements

The processing of document and script locations is performed using a custom parser. The line in which a script is located within a document is passed by the browser to the API methods as a parameter, so by examining the location of a script (codebase and line number) we can distinguish between internal and external scripts. The same parser is used to extract the static references to external resources contained in a script. In particular, by using regular expresions, the parser extracts the various parts that the URL consists of, such as the protocol (e.g., HTTP), authority (domain name and port), path, and query.

If the JavaScript code is a script fed to `eval`, then we also retrieve the JavaScript stack trace that led to the `eval` function call. To accomplish that, we utilize the data structures that SpiderMonkey uses to monitor the execution of a script, most notably the `JSContext` and `JSStackFrame`. The former contains the execution stack, while the later is a structure representing a frame of the JavaScript virtual machine stack. A frame contains information regarding the method being executed, the program counter, the formal and actual arguments, the local store associated with each call, and the script being executed (function name if available, source location, etc.). It also contains a pointer to the next frame on the stack, which allows us to traverse the execution stack by iterating through the frames and collecting the information we need for each one. This information includes the location of the script being executed (domain name part of the URL of the document containing the script) and the name of the JavaScript function if available or "anonymous function" otherwise. This information is appended to the stripped down form of the source code being passed to the `eval` method and is used to generate the final script fingerprint. The structure of the stack and the stack frames are illustrated in Figure 6. The final script fingerprint occurs by combining the above elements with the aggregated counts of the JavaScript keywords contained in the script and the script's block structure and applying an SHA1 hash function on them.

As we described in Section 3, we needed to associate the domain part of URLs that can be referenced by a script at runtime with the corresponding script fingerprint. To identify such URLs, we analyze the string type object values in the heap space of the engine. In particular, the execution of any compiled script eventually goes through the function `RunScript`. Within this function, we have access to all the argument values that are passed to any compiled JavaScript function. `RunScript` performs some sanity checks and then either invokes the JavaScript interpreter or the JIT compiler. At this point, we extract all the URLs that are contained in string arguments. If any URLs are found, then we retrieve the domain part of each one and apply the same hash function

to it, thus generating a URL fingerprint. Note that this process is only performed when a compiled script is executed.

4.3. Fingerprint Management

As we discussed in Subsection 3.3, to collect valid fingeprints automatically, we have developed a fingerprint generation module based on the Crawljax [Mesbah and van Deursen 2009; Mesbah et al. 2012] testing framework. Crawljax provides an interface that allows the tester to configure the depth of the testing (practically the depth that the web site crawler will reach during the test) and the browser to be used. By using the browser with our modified JavaScript engine, the administrator can automatically retrieve all fingerprints without having to manually browse the whole site and execute its functionality. We have evaluated the efficacy of our fingerprint generation module and we elaborate on our results in Subsection 5.2.1.

As a back-end for storing the fingerprints generated by nsign, we employed an SQLite database.[14] During training the database is used to store the fingerprints that will then be exported into the fingerprint file. In production mode, the database serves as a local cache of fingerprints for the websites that the user has visited in the current session. This helps us avoid multiple requests for the fingerprint file on each visited web site. To aid evaluation in our prototype implementation, we did not fully implement the mechanism for the retrieval of the fingerprints via a secure channel from within nsign but rather used a mock-up proxy that retrieves the fingerprint files from either local store or using an external application to fetch fingerprint files over a secure channel.

5. EVALUATION

We have evaluated our framework in terms of effectiveness, operation cost, and maintenance cost.

5.1. Effectiveness

To evaluate the effectiveness of nsign, we searched for web applications that had a record of being vulnerable to XSS attacks. Our experiment involved vulnerable applications that were installed and attacked locally and real-world XSS exploits hosted by xssed.com.[15]

5.1.1. Laboratory Tests. For our laboratory tests, we downloaded the vulnerable versions of five well-known OSS projects, including Joomla[16] and phpMyFAQ.[17] First, we applied our fingerprint generation module to the applications. Then, we switched to production mode and performed various attacks based on the vulnerability type of every application. Defects like the absence of proper input sanitization and the improper verification of HTTP requests allowed us to perform redirect and cookie stealing attacks. For example, in phpMyFAQ, URLs are not sanitized correctly, making it possible to inject JavaScript code to steal the user's cookies. Table I shows, for each web application, its vulnerable version and the attack type that we performed. All attacks were successfully prevented without encountering any false alarms.

5.1.2. Prevention of Real-World Attacks. We selected 50 real-world vulnerable applications taken directly from the vulnerability archive hosted by XSSed.com. This archive has been previously used for evaluation in other articles [Athanasopoulos et al. 2010; Nadji et al. 2006]. Initially, we searched for XSS attacks that utilized JavaScript. Then we

[14]http://www.sqlite.org/.
[15]www.xssed.com.
[16]www.joomla.org/.
[17]www.phpmyfaq.de/.

Table I. Vulnerable Web Applications Used in Our Laboratory Tests

Application	Version	Attack Type
Joomla	1.5.20	Cookie Stealing
phpMyFAQ	2.6.8	Cookie Stealing
Pluck	4.6.3	Redirect
JCart	1.1	Redirect
TikiWiki	1.9.8.1	Cookie Stealing

checked if their status was *unfixed*, which means that the site administrators have not fixed the existing defect yet, and the site is vulnerable to attacks even if it is online. The vulnerable webpages included top-ranking sites like eBay.com and nydailynews.com. The complete list of the sites can be found on Appendix A. Based on the existing examples that XSSed.com provides, we attempted a variety of attacks, including DOM-based attacks and attacks that utilized the `eval` function and `iframe` tags. nsign recognized and blocked all of the attacks, without producing any false alarms.

5.1.3. Security Analysis. All the attacks that we performed in our experiments led to fingerprints that were not generated during the training phase, and, hence, they were prevented by our mechanism. Each case had different characteristics and was prevented for different reasons. In this subsection, we show how the various attacks were stopped by nsign. We also discuss the possibility of False-Negative (FN) and False-Positive (FP) errors.

Attack Prevention. Cookie stealing and redirection attacks produced unrecorded fingerprints because of (a) the injected script and (b) the URLs that were statically referenced within the script. The attacks that utilize `iframe` tags (described in Subsection 2.1) were blocked for the same reasons. In addition, we performed a redirection attack by assembling a URL in the way we presented in Subsection 3.2.2. In this case, an unrecorded URL fingerprint was a factor, too. `eval`-based attacks were detected because they produced non-recorded fingerprints coming from (a) the script fed to the function, (b) its type, and (c) the unexpected invocation stack trace. We have also incorporated the banner-rotator mentioned in Subsection 2.1 into one of our laboratory tests. In this way, we managed to perform a mimicry attack like the one described in the same subsection. nsign successfully prevented the attack because the different URLs that were referenced within the script led to the generation of an unrecorded fingerprint.

False Negatives and Mitigation. A false negative would occur by performing a mimicry attack that would cause the execution of a script that is otherwise legitimate in a different context. For example, consider a vulnerable website that is structured in a way that contains a public facing part for visitors and a private part that is supposed to be used by authenticated users. In the private part, assume that there is a script that can be used to delete the contacts of a user based on a variable stored in the user's cookies. If an attacker manages to inject the exact same script in the public part of the site, and the website does not perform authorization checks when the corresponding request is performed, then the contacts of a casual user of this website would be deleted since the generated fingerprint will be valid. Such an attack is similar to the "Forcing Logout" attack presented in Athanasopoulos et al. [2009].

However, this attack can be prevented by taking into account the entire URL path from which a script originated and is referenced (recall our design choice at the end of Subsection 3.2.1). Note that this is based on the assumption that the public and private parts of the website share the same domain name but have different path and segment locations. If the domain name differed, then the attack would not work in the

first place, because the different domain name would lead to a different fingerprint. Nevertheless, including the entire URL path as a fingerprint element would lead to the increase of the number of fingerprints as we show in Subsection 5.3.

Another false negative could occur in the context of a social media website, such as Facebook. Consider an attacker who could craft a script that includes the same keywords and has the same block structure with the one used by the website when users share a post on their wall. If the attacker manages to share a post that incorporates this script on a user's wall, then this script would run every time another user sees the post without the intervention of nsign. This mimicry attack, though, would work only if the benign script is an embedded script. If it is an external script, then the fingerprint would differ (recall that we include the type of each script as an element of the fingerprint). In the same context, consider an embedded script that invokes a function defined in an external one. Suppose that this function has the following signature: `sendMessage(<user>, <message>)`, and is used to send a message to another user on behalf of the currently logged in user. By leveraging a benign call to a function with the same signature in an embedded script, an attacker could invoke the external script function with different arguments and potentially spam other users.

False Positives. There are two main reasons behind the false positives that may be produced by our scheme. The first involves an incomplete training phase. Nevertheless, by using our fingerprint generation module (see Subsections 3.3 and 4.3) we have achieved a high percentage of coverage in all the above cases. This was exemplified by the fact that we did not encounter any false positives during our tests. A false positive may also occur if scripts are modified or new scripts are added on the server side. In this case, a training phase should take place again. We further discuss this issue in Subsection 5.2.2.

5.2. Fingerprint Maintenance and Performance

To evaluate the runtime performance of our prototype and provide details on the automation of the fingerprint collection, we performed two experiments. In the first one, we evaluated the effectiveness of the proposed fingerprint collection scheme on the server side and examined the temporal aspect of fingerprints. In the second one, we evaluated the performance of our solution on the client side. In both cases, we used a build of Firefox (v32.0.3). We ran our experiments against eight high-profile websites. Both experiments were run on a client with a quad core Intel i7 processor with 8GB of RAM running Linux (64-bit Ubuntu 15.02).

5.2.1. Fingerprint Generation with Crawljax. We configured Crawljax to automatically follow the links from the site's homepage up to three levels deep in the site's page hierarchy. The number of fingerprints captured for each site, along with the total time required by Crawljax to crawl the site (including network latency), are reported in Table II. The results show that the total number of fingerprints is small even for very large websites containing thousands of pages. Even if a full website scan is required, our measurements indicate that the fingerprint collection is feasible, and it can be automated via a testing suite like Crawljax. After generating the fingerprints, we extensively browsed the websites. While browsing, we did not encounter any false positives, and nsign did not interfere with the browser's operation in any way.

5.2.2. Fingerprint Change over Time and Potential False Positives. When script elements are altered or new scripts are added on the server side, a fingerprint generation phase should take place again, otherwise false positives will occur. To examine the temporal aspect of contextual fingerprints we performed two experiments.

Table II. Crawljax Execution Statistics and the Temporal Aspect of Fingerprints (Long-Term Experiment). Time Is Measured in Minutes and Seconds

Domain	T_1		T_1 + 6 Months		# of unchanged
	Fingerprints	Crawljax Time	Fingerprints	Crawljax Time	fingerprints
gmail.com	511	48:23	534	49:15	206 (40.31%)
ebay.com	253	37:10	312	40:41	133 (52.56%)
amazon.com	1404	62:04	1467	66:45	575 (40.59%)
twitter.com	1465	34:06	1504	37:12	611 (41.7%)
facebook.com	3108	45:36	3511	47:29	456 (14.67%)
wikipedia.org	343	62:34	378	65:06	211 (61.51%)
cnn.com	537	54:53	511	53:33	282 (52.51%)
bbc.co.uk	853	41:20	696	36:53	147 (17.23%)

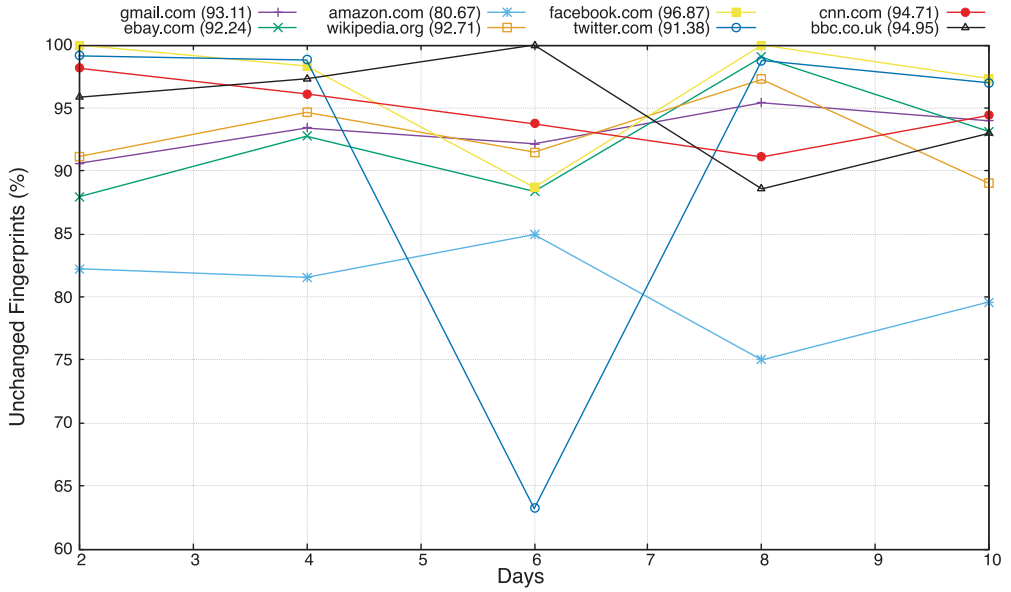

Fig. 7. Short-term experiment on the temporal aspect of fingerprints. Examining how many fingerprints (%) remain the same for every 2 days. The arithmetic mean of each website can be seen on top of the graph, right next to the website's name.

Short-Term Experiment. Every 2 days, we generated the fingerprints for a specified set of pages coming from each website, using our fingerprint generation module. This was done over a period of 10 days. Each time we collected the fingerprints, we checked them against those generated two days earlier and identified the ones that were the same. Our results can be seen in Figure 7. Observe that in most cases the majority of the fingerprints remained unchanged. For instance, if we take a closer look on BBC's transition from day 0 to day 2, we see that the 95.86% of the fingerprints remained unchanged. Specifically, we found nine new fingerprints. There are three possible reasons behind this: (a) the introduction of new scripts, (b) the modification of existing ones, and (c) interactions with new, dynamically referenced URLs (which led to unrecorded URL fingerprints; recall that URL fingerprints correspond to different URLs that are passed as arguments to the various functions). During our experiments, though, we were not aware of all the benign domains that could interact with each website in this way. If we did, then the number of unchanged fingerprints could be higher because we would be able to check if we have all the corresponding URL fingerprints. Furthermore,

URL fingerprints seem to be the main reason behind the different behavior of Amazon. Note that the developers of a web application may choose not to use the URLs that are passed as arguments to the various functions as elements. In this case, the number of unchanged fingerprints would increase and the need for retraining would decrease.

Another interesting observation involves the massive change in the fingerprints of Twitter in the sixth day. A change like this could suggest scheduled updates in many scripts and/or the introduction of new ones. This behavior is consistent with our expectations and further promotes the idea of regenerating the fingerprint set when changes are made on the server side. Overall, our results indicate that nsign could be used by dynamic websites.

Long-Term Experiment. To check how fingerprints change over a long period of time, we did the same experiment presented in Subsection 5.2.1, 6 months later. In Table II, we illustrate how many fingerprints stayed the same after 6 months. Observe that in all cases there are unchanged fingerprints, while there are many new ones that have been introduced. This could involve software updates, changes in advertising, and interactions with new URLs. Notably, Wikipedia holds a high percentage of unchanged fingerprints—61.51%. In the case of Facebook and BBC, though, the number of unchanged fingerprints is quite low: 14.67% and 17.23%, respectively. This is reasonable because, during this period, the user interfaces of both websites went through numerous changes. Besides, such popular websites are expected to go through significant changes more often than others.

5.2.3. Computational Overhead. To measure the overhead of fingerprint generation, we instrumented both SpiderMonkey's JavaScript parser and interpreter and our fingerprint generation code with high precision time counters. For SpiderMonkey, we measured the time required to parse the input code, as well as the execution time, whether the script was interpreted or compiled and executed. The total time required to execute our code was broken down into four operations as follows: (a) code formatting, stripping and basic extraction of elements, (b) lookup of a fingerprint from the persistent store, (c) traversal of the execution stack (only applicable in the case of `eval` calls), and (d) scanning of function arguments for URLs.

Using the instrumented version of SpiderMonkey, we used Firefox to visit the aforementioned list of websites in order to trigger fingerprint generation. For websites whose home page requires authentication (e.g., Gmail), we logged in with the benchmarker's account before benchmarking; the login information persisted across benchmark runs. Each run consisted of starting a saved Firefox session with all websites opened as tabs. Measuring performance at the microsecond level in complex systems in non-isolated environments can introduce a high degree of variance in the obtained results [Georges et al. 2007]. In our case, sources that added to variance are background operating system services and system interrupts required for processing user and network events. Consequently, and in order to obtain statistically significant results, we run the experiments 31 times. Table III indicates that the bulk time is spent on the generation of script fingerprints; upon further investigation, we concluded that the majority of time is spent on script analysis. Note that we browsed selected pages for each website. Hence, the fact that we did not encounter the `eval` facility during our tests does not imply that the method is not used by the webpage. However, even when we did encounter it, the time required to generate stack traces was negligible. Furthermore, most compiled scripts invoked either had no string arguments or these arguments did not contain URLs, making the argument parsing overhead minimal in most cases.

Depending on the website design, the number of produced fingerprints can be quite large; even so, since our system only stores an SHA1 hash for each fingerprint, the space requirements are trivial.

Table III. Performance of System Prototype. Time Is Measured in μs

Domain	nsign Time				Total	SpiderMonkey[5]	Overhead (%)
	Analysis[1]	Retrieval[2]	Stack[3]	Args[4]			
gmail.com	154	13	0	6	172	1182	14.5
ebay.com	381	150	0	0	531	9269	5.7
amazon.com	287	47	0	27	361	3860	9.3
twitter.com	257	451	0	0	708	3120	22.6
facebook.com	1782	453	15	0	2235	9933	22.5
wikipedia.org	218	49	0	0	268	28,030	0.9
cnn.com	265	181	22	0	446	93,148	0.4
bbc.co.uk	64	21	0	3	88	659	13.3

[1]Code stripping and element extraction.
[2]Fingerprint retrieval from the fingerprint file.
[3]Stack traversal.
[4]URL extraction from string arguments.
[5]SpiderMonkey's execution time without our functionality.

Compared to the time required by SpiderMonkey to parse each page's scripts, our layer appears to impose an average overhead of 11.1% (note that this overhead is produced by a proof-of-concept with no optimizations). In practical terms, however, this overhead is smaller or comparable to the speed-ups offered by modern JavaScript engines in recent years and, in most cases, is not perceivable by end users, as their experience is largely dominated by network latency and page rendering. On our test machine, and for all tested sites, the round-trip time required for downloading a script was in the range of seconds, which makes our layer's overhead negligible (less than 0.05%). The results indicate that the described solution can be of practical value as the overhead it imposes is minimal both in time and in space (a few kilobytes) terms, and as a result it does not affect the user's experience.

5.3. Tradeoffs

Our approach's design aims to provide flexibility regarding the inclusion and exclusion of elements from script fingerprints. For instance, the semicolons contained in the script could be easily added as a characteristic to the script fingerprint. This renders mimicry attacks that add statements more difficult to perform, because the number of script commands must remain unaltered, thus imposing a strict limit on the attacker and providing extra robustness (mimicry attacks that remove statements can simply pad the script with additional semicolons). This though, comes with a tradeoff between effectiveness and computational overhead (the time required to analyze the code increases). Similarly, by checking the URLs that are dynamically referenced by a script, the computational overhead grows. However, without this element, our approach would become susceptible to an attack like the one presented in Subsection 3.2.2.

As we mentioned in Subsection 5.1.3, we can avoid a class of potential false negatives by including the URL path as an element (instead of having the domain name only). Nevertheless, this may lead to fingerprint coverage and maintenance issues. The flexibility of our scheme allowed us to perform an experiment to highlight the aforementioned tradeoff. The experiment involves the generation of fingerprints for the Imgur online image sharing community.[18] We have selected this website as the subject of our experiment because it follows modern web development practices, like many other social networking websites (e.g., Facebook). Specifically, Imgur uses URL patterns

[18]http://imgur.com/.

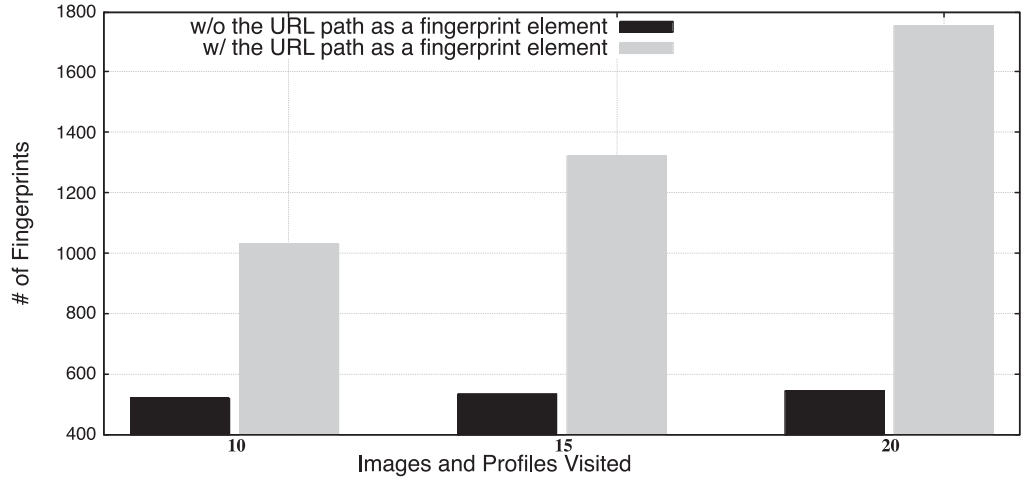

Fig. 8. Adding the entire URL path as a fingerprint element. For a given website (imgur.com), we examined how the number of the fingerprints changes during the training phase with and without the path. Black bars show the number of fingerprints that nsign generates when the domain name of the URL is incorporated as an element. Grey bars illustrate the number of fingerprints generated when the whole URL path is included.

(routes) for its pages that adhere to the REST style conventions [Richardson and Ruby 2007]. For instance, the route for an image gallery is as follows: `/gallery/{galleryID}`.

The experiment was performed in three steps. For each step, we changed our scheme in order to measure the fingerprints with and without the URL path that triggered the execution of each script. During the first step we visited the website's main page and consecutively the first 10 images that appear in the first page, as well as the profile page for the user that posted each one. After generating the fingerprints, we performed the second step, in which we visited the next five images and the corresponding profiles. For the third and last step we did the same thing for the next five images/profiles. Figure 8, illustrates the results of the experiment. In black, we see the number of fingerprints when only the domain name is included as a fingerprint element (imgur.com). In grey, we observe the number of fingerprints when the whole URL path is included (e.g., imgur.com/gallery/L2SPQwQ).

As our results indicate, including the entire URL path as a fingerprint element leads at best to a linear increase of the number of fingerprints for sites built with such a structure (because similar paths that differ only by one or more segments, must be associated with the same scripts). This behavior would make the training phase more difficult and lead to potential false positives. Hence, apart from the tradeoff between efficiency and overhead, we observe that there can be a tradeoff between false positives and negatives.

6. DISCUSSION

Our scheme can be used together with other training mechanisms. ZigZag [Weissbacher et al. 2015] is one of those cases. In particular, ZigZag prevents attacks that exploit client-side validation vulnerabilities. To do so, it generates models that depict how client-side components interact. To work as intented, ZigZag should be installed on the browser side, together with "complementary defenses against XSS-based code injection attacks." Also, when a script is modified on the server side, the script-instrumentation that is needed to create the models must take place again. Hence, both mechanisms could share the same training phase.

In order to simplify the fingerprint management and propagation, the client-side processing could be simplified by removing the fingerprint caching mechanism. The fingerprints that are valid for a given page could be sent over to the client along the response using standard HTTP headers. The communication overhead should be minimal, since a page is associated to a limited number of fingerprints. As a result, the size of the headers would not increase significantly. However, such an approach would require modifications on the server side, so the valid fingerprints will be sent to the client embedded in the response headers.

In recent years the web application development practice has displayed an increasing trend towards the use of common JavaScript libraries like jQuery. This provides an opportunity to minimize the number of fingerprints that need to be generated for a website by whitelisting the fingerprints of popular libraries. In essence, this technique not only leads to a smaller fingerprint set but also minimizes the need for re-generating fingerprints for a web application if a library it depends on is updated. Web applications are typically built and tested against a specific version of a third-party library, thus no re-syncing is required for every new release of a library. If a developer chooses to use a new version of a library, then the full set of fingerprints needs to be generated anyway, but most popular JavaScript libraries have a release cycle of 6 months or more.

This type of whitelisting is becoming increasingly meaningful given that web developers nowadays link directly to the latest version of libraries hosted on Content Distribution Networks (CDNs) [Nikiforakis et al. 2012], instead of bundling local copies with their application. Direct linking provides the advantage that no manual updating of libraries is required and may possibly protect against vulnerabilities contained in outdated library versions. In addition, CDNs typically offer higher performance and availability for delivering libraries to the end user. Such a whitelisting approach could allow for false positives in the following scenario: Consider a web application that links to the latest version of a library that contains calls to the `eval` method. The web application itself does not use `eval` directly but relies on some API method of the library that does. If the library is updated and the aforementioned method is changed so `eval` is called through a different path, then a different stack trace will be generated, causing a false positive. However, the effect of this shortcoming can be minified by employing the whitelisting technique only for libraries that are not directly or indirectly dependent on `eval`.

We have also experimented with our mechanism to examine the impact of denying the execution of the scripts without a corresponding fingerprint. The results of such an event may vary, due to possible cascading effects. For example, consider a scenario where a web page is using the jQuery library, and the developers update the version of jQuery without performing the training phase. In this case, the functionality of the page—and possibly its rendering—can be severely affected. Apart from this case, though, we did not observe any other significant effect during our testing. For instance, if the banner rotator script mentioned in Subsection 2.1 has no corresponding fingerprint, the webpage is still functional but with the banner missing.

Finally, note that ECMAScript 5 provides a mode ("strict mode") where, when enabled, `eval` cannot introduce new variables into the surrounding scope. This could contain `eval`-driven attacks, but it cannot detect mimicry and XCS attacks.

7. RELATED WORK

There are four basic approaches to dynamically detect XSS attacks, namely policy enforcement, Instruction Set Randomization (ISR), taint tracking, and training. When using a framework that implements policy enforcement, developers must define specific security policies on the server side. Policies can be expressed through JavaScript

extensions, pattern matching, or syntax-specific settings. The policies are then enforced either in the user's browser at runtime or on a server-side proxy that intercepts server responses. Some frameworks define policies based on information and features provided by the DOM of a web page. Specifically, developers must place all benign scripts inside HTML elements like `div`. The web browser parses the DOM tree and executes scripts only when they are contained in such elements. All other scripts are treated according to the policies defined on the server. Frameworks that support this functionality include BEEP [Jim et al. 2007] and DSI [Nadji et al. 2006]. Both mechanisms suffer from effectiveness issues, as indicated in Subsection 2.1.

Another policy enforcement approach introduces policies directly either in HTML or JavaScript code to confine their behavior. BrowserShield [Reis et al. 2007] acts as a proxy on the server side to parse the HTML of server responses and identify scripts. Then, it rewrites them into safe equivalents and protects the web user from exploits that are based on reported browser vulnerabilities. ConScript [Meyerovich and Livshits 2010] and CoreScript [Yu et al. 2007] extend JavaScript with new primitive functions that provide safe methods to protect potentially vulnerable JavaScript functions. In most cases, policy enforcement takes place at client side in the JavaScript engine of the browser. As we mentioned in Subsection 2.1, the above frameworks are vulnerable to attacks that employ either script inclusion or `iframe` tags.

The framework by Phung et al. [2009], WebJail [Van Acker et al. 2011], JSand [Agten et al. 2012], TreeHouse [Ingram and Walfish 2012], and SOMA [Oda et al. 2008] are five schemes that, contrary to the previous ones, can actually detect attacks that employ script inclusion or `iframe` tags. To achieve this, SOMA requires site administrators to specify legitimate, external domains for sending or receiving information in order to approve interactions between them and the protected website. WebJail contains the functionality of third-party scripts by introducing a web component integrator that restricts the access that these scripts may have to either the data or the functionality of other components. JSand and the framework by Phung et al. operates in a similar way. TreeHouse provides to the developers frameworks that contain JavaScript in ways similar to sandboxing and virtualization. Blueprint [Louw and Venkatakrishnan 2009] is a policy enforcement framework that uses parsed trees to detect XSS attacks. To guarantee that untrusted content is not executed, Blueprint generates at the server side a parsed tree from untrusted HTML to ensure that it does not contain any dynamic content. Then, the parsed tree is transfered to the document generator of the browser, where untrusted browser parsing behavior is ruled out.

A policy enforcement technique developed by Mozilla, called CSP [Stamm et al. 2010; Fazzini et al. 2015] is currently supported by the majority of browsers to prevent various types of attacks, including XSS attacks. To eliminate such attacks, website developers can specify which domains the browser should treat as valid sources of a script and which it should not. Then, the browser will only execute scripts that exist in source files from whitelisted domains. Note that if a CSP-based application involves embedded scripts, developers must utilize the *nonce* concept. This is an unpredictable, random value indicated in the `script-src` directive that, in turn, is applied as a nonce attribute to `<script>` elements. As a result, only those elements that have the correct nonce will execute. Even if an attacker is able to inject markup into the page, the attack will be prevented by the attacker's inability to guess the nonce value. However, attackers may still bypass this feature and invoke a script from a non-whitelisted source. To do so, their injected code must be crafted in a way that the nonce is handled by the browser as an attribute of the payload.[19] Notably, this attack would be prevented by

[19]http://blog.innerht.ml/csp-2015/.

our mechanism because the injected script and the non-whitelisted URL that triggered its execution would lead to non-recorded fingerprints. CSP could be used along with our mechanism. For instance, developers could set policies only for the domains that should be treated as valid sources of scripts. Then, they could utilize nsign to guard embedded scripts against XSS attacks. Note that, in this case, URLs should not be included as fingerprint elements.

AutoCSP [Fazzini et al. 2015] and deDacota [Doupé et al. 2013] are two schemes that are based on CSP. In particular, AutoCSP is a mechanism that can automatically retrofit CSP to web applications and deDacota [Doupé et al. 2013] enforces code and data separation at runtime through the CSP mechanism. Google Caja[20] is another policy enforcement approach provided by Google. It is based on the object-capability security model [Miller 2006], and it aims to control what the embedded third-party code can do with user data.

ISR is a method that has been applied to counter different kinds of application attacks [Keromytis 2009]. In the XSS prevention context, this approach works as follows: Initially, the trusted code of a webpage can be transformed to a random representation using a simple function such as XOR. Before being sent to the client, or being processed by the browser, the legitimate code is transformed back to its original form, while any additional injected code will be transformed into junk code. Variations of this approach include Noncespaces [Gundy and Chen 2009] and xJS [Athanasopoulos et al. 2010], which randomize the instruction set of HTML and JavaScript, respectively. Contrary to xJS, in Noncespaces, administrators must set specific policies in a manner similar to a firewall configuration language. SMask [Johns and Beyerlein 2007] is another framework that was inspired by ISR. To detect XSS attacks, it searches for HTML and JavaScript keywords within the application's legitimate code. This is done before the processing of any HTTP request. When a keyword is found, it adds a token to it, resulting in a "code mask." Then, before sending the resulting HTML data to the user, the framework searches the data. Since all legitimate code has been "masked," the injected code can be identified.

Notably, Sovarel et al. [2005] have examined thoroughly the effectiveness of ISR and showed that an attacker may be able to circumvent it by determining the randomization key. Their results indicate that applying ISR in a way that provides a certain degree of security against a motivated attacker is more difficult than previously thought.

A taint tracking scheme, marks untrusted ("tainted") data and traces its propagation through the program. Context-Sensitive String Evaluation (CSSE) [Pietraszek and Berghe 2006] associates tainted data with specific metadata that includes the origins of the tainted data and its propagation within the application. When tainted data reach a potentially risky operation, CSSE performs syntactic checks based on its metadata. PHP Aspis [Papagiannis et al. 2011] works in a similar way. To obtain metadata, it takes advantage of the PHP array data structure. Finally, WASC [Nanda et al. 2007] analyzes HTML responses to check if there is any tainted data that contain scripts. A recent study [Naderi et al. 2014] showed that there are ways to circumvent schemes like the above. Vogt et al. [2007] have developed a tainting scheme that follows a different approach. In contrast to the above methods, their technique tracks sensitive information at the client side. The scheme counters XSS attacks by ensuring that a script can send sensitive user data only to the site from which it came from. Stock et al. [2014] have developed a mechanism that also operates in the browser and focuses on the prevention of DOM-based XSS attacks. To do so, it employs a taint-enhanced JavaScript engine that tracks the flow of attacker-controlled data. To detect potential

[20]https://code.google.com/p/google-caja/.

attacks, it uses HTML and JavaScript parsers that can identify the generation of code coming from tainted data. Finally, SCRIPTGARD [Saxena et al. 2011] employs positive taint tracking on the server side to detect sanitization errors that may lead to XSS attacks. Note that, in the context of positive data flow tracking, the tagged data are considered to be legitimate.

An issue that involves taint-tracking schemes is the difficulty of maintaining accurate taint information [Kang et al. 2011] (e.g., handling implicit flows). In such cases, certain, tainted inputs can escape the tracking mechanism. Keeping track of such input may be impractical not only because of the various technical difficulties but also because it would raise false alarms.

Apart from nsign and XSS-GUARD [Bisht and Venkatakrishnan 2008] (described in Subsection 2.1), training approaches also include mechanisms like SWAP [Wurzinger et al. 2009] and XSSDS [Johns et al. 2008]. In Subsection 2.1 we presented in detail an attack to bypass XSS-GUARD. Recall that the attack involved a script that references external URLs. nsign can prevent such an attack because the referenced URLs would lead to an unrecorded fingerprint. Note that referenced URLs are not always static parts of the script code: They can be assembled in numerous ways by attackers as we showed in Subsection 3.2.2.

SWAP creates a unique identifier for every benign script on the server. Then, a JavaScript detection component placed in a web proxy searches for injected scripts with no corresponding identifier in the server's responses. If no injected scripts are found, then the proxy forwards the request to the client. SWAP uses the whole benign script to create an identifier during the training phase. However, this is meaningful for websites that preserve only static scripts, something that rarely happens nowadays. nsign uses specific script elements rather than the whole script. In this way, it can apply its policies during script interpretation/execution, thus supporting dynamic scripts.

The authors of XSSDS have implemented a similar mechanism that also supports dynamic and external scripts. Specifically, during the training phase, they build a list of all benign scripts. For external scripts, they keep a whitelist of all the valid domain names that contain scripts used by the application. A limitation of the XSSDS mechanism (as the authors themselves point out in their article) is that stored XSS attacks are not always detectable. This is because the mechanism relies on a direct comparison of incoming HTTP parameters and outgoing HTML, something that does not apply to our scheme. Specifically, nsign generates fingerprints for all scripts because it wraps the JavaScript engine of the browser. Thus, it would generate an unrecorded fingerprint for the script involved in the corresponding stored XSS attack, and the attack would fail.

Table IV compares the various XSS countermeasures in terms of false positives and negatives and computational overhead. If the publication mentions that the mechanism actually produces FPs or FNs but it does not explicitly states any rates, then we use an X mark (✘). If it is effective, then we use a tick mark (✔). If the mechanism was not tested in terms of effectiveness at all, then we add a question mark (**?**). Note that there are mechanisms where, even if they seem effective, their testing might be poor contrary to other schemes that may have false alarms but have been tested thoroughly. For example, SMask [Johns and Beyerlein 2007] appears to be an effective solution, but the corresponding publication presents only a few test cases. On the other hand, DSI [Nadji et al. 2006] appears to have FPs and FNs, but it was evaluated on a dataset of hundreds of vulnerable websites.

In a similar manner, we list the overhead for every mechanism as stated in the original publication. If the publication mentions that the mechanism suffers from a runtime overheard but does not state the occurring overhead, then we use the X mark (✘). If the authors did not measure the overhead, then we use a question mark (**?**).

Table IV. Comparison Summary of Mechanisms Developed to Prevent XSS Attacks

Approach	Mechanism	Criteria		
		FP	**FN**	**Computational Overhead**
Policy Enforcement	DSI [Nadji et al. 2006]	✘	✘	1.85%
	BEEP [Jim et al. 2007]	?	✔	14.4%
	BrowserShield [Reis et al. 2007]	✔	✔	8%
	CoreScript [Yu et al. 2007]	?	✘	?
	SOMA [Oda et al. 2008]	?	✔	5.58%
	Blueprint [Louw and Venkatakrishnan 2009]	?	✔	13.6%
	Phung et al. [2009]	?	✘	5.37%
	ConScript [Meyerovich and Livshits 2010]	?	?	7%
	CSP [Stamm et al. 2010; Fazzini et al. 2015]	?	?	?
	WebJail [Van Acker et al. 2011]	?	✘	6.89ms
	JSand [Agten et al. 2012]	?	?	up to 365%
	TreeHouse [Ingram and Walfish 2012]	?	?	757–1218ms
	Google Caja	?	?	?
	deDacota [Doupé et al. 2013]	?	✔	?
ISR	SMask [Johns and Beyerlein 2007]	✘	✘	?
	Noncespaces [Gundy and Chen 2009]	?	✔	2%
	xJS [Athanasopoulos et al. 2010]	✔	✘	1.6–40ms
Runtime Tainting	CSSE [Pietraszek and Berghe 2006]	✘	✘	2–10%
	Vogt et al. [2007]	✘	?	✘
	WASC [Nanda et al. 2007]	✘	✘	up to 30%
	PHP Aspis [Papagiannis et al. 2011]	✘	✘	2×
	SCRIPTGARD [Saxena et al. 2011]	?	?	up to 3.82%
	Stock et al. [2014]	?	✔	7–17%
Training	SWAP [Wurzinger et al. 2009]	✘	✘	up to 261ms
	XSSDS [Johns et al. 2008]	✘	✔	?
	XSS-GUARD [Bisht and Venkatakrishnan 2008]	✘	✘	5–24%
	nsign	✔	✘	11.1%

8. CONCLUSION AND FUTURE WORK

We have proposed a training approach to counter the ongoing threat of JavaScript-driven XSS attacks based on fine-grained, contextual script fingerprints. Our "defense-in-depth" approach is flexible because it can include diverse elements to create a fingerprint, depending on accuracy and convenience tradeoffs. Another advantage of our approach is that it deals with all scripts, coming from every web location and through any possible route. In this way, it can prevent a variety of attacks. Also, the browsing experience of a user is not noticeably affected by the overhead that the layer imposes. Finally, contrary to other schemes, developers are not required to modify the code of a web application to support our approach.

A disadvantage regarding the effectiveness of our approach is that when a fingerprint element is altered, a new fingerprint generation phase is necessary on the server side. However, as we showed earlier in the article, with the increased adoption of test-driven development and the use of automated testing and continuous integration frameworks, the fingerprint generation phase can be easily repeated. Also, our experiments regarding the temporal aspect of fingerprints, indicate that for a reasonable period of time, and for most popular websites, the majority of fingerprints remain unchanged. This suggests that even dynamic websites like Facebook and Twitter could take advantage of a countermeasure like nsign.

Another limitation is that nsign depends on the creation of all corresponding fingerprints via dynamic analysis, which is a known hard problem. Finally, the fact that our layer is placed within the JavaScript engine of the browser implies that for users

to benefit from our scheme's security they must install an nsign-modified browser. Nevertheless, as we observed in Section 7, many effective solutions require similar modifications.

Although we have implemented our layer to wrap SpiderMonkey, the same layer could be retrofitted in other JavaScript engines, such as Chrome's v8.[21] Future work on our system involves the implementation of HTTP-based fingerprint data retrieval and further evaluation in terms of effectiveness. In addition, we plan to investigate how a third party can run the training phase so end-users can fetch the fingerprints from that third party without the involvement of the website itself.

Availability

The source code of nsign is available as open-source software at https://github.com/istlab/nSign/.

A. APPENDIX

Table V contains the complete list of the vulnerable sites that we tested. In particular, they contain, for every vulnerable domain name, the URL pointing at the corresponding entry at the XSSed.com archive and the current status of the vulnerability.

Table V. Real-World, Vulnerable Websites where nSign Was Tested

Vulnerable Domain	URL entry at XSSed.com	Status
search.nate.com	http://www.xssed.com/mirror/70707/	*unfixed*
mg269.imageshack.us	http://www.xssed.com/mirror/70704/	*unfixed*
www.aolsvc.merriam-webster.aol.com	http://www.xssed.com/mirror/70719/	*unfixed*
cookbooks.adobe.com	http://www.xssed.com/mirror/70721/	*unfixed*
wap.ebay.ie	http://www.xssed.com/mirror/70726/	*unfixed*
hotels.qantas.com.au	http://www.xssed.com/mirror/70735/	*unfixed*
www.attijaribank.com.tn	http://www.xssed.com/mirror/70696/	*unfixed*
www.topgear.com	http://www.xssed.com/mirror/70346/	*fixed*
webforms.ey.com	http://www.xssed.com/mirror/70345/	*unfixed*
cib.ibank.ge	http://www.xssed.com/mirror/69548/	*unfixed*
ie.jrc.ec.europa.eu	http://www.xssed.com/mirror/69254/	*unfixed*
esupport.trendmicro.com	http://www.xssed.com/mirror/68756/	*fixed*
mail.kmr.gov.ua	http://www.xssed.com/mirror/68689/	*unfixed*
sge.corumba.ms.gov.br	http://www.xssed.com/mirror/68688/	*unfixed*
zones.computerworld.com	http://www.xssed.com/mirror/70652/	*unfixed*
www.nydailynews.com	http://www.xssed.com/mirror/67620/	*unfixed*
software.gsfc.nasa.gov	http://www.xssed.com/mirror/67530/	*unfixed*
seguridad.terra.es	http://www.xssed.com/mirror/67296/	*unfixed*
www.jinx.com	http://www.xssed.com/mirror/70589/	*unfixed*
www.nsbank.com	http://www.xssed.com/mirror/70528/	*unfixed*
blackhat2010.sched.org	http://www.xssed.com/mirror/70534/	*unfixed*
www.ccbill.com	http://www.xssed.com/mirror/70537/	*unfixed*
w2.eff.org	http://www.xssed.com/mirror/70353/	*unfixed*
www.adobe.com	http://www.xssed.com/mirror/70463/	*unfixed*
www.antenna.gr	http://www.xssed.com/mirror/70495/	*fixed*
pub.kathimerini.gr	http://www.xssed.com/mirror/70510/	*fixed*
vod.grnet.gr	http://www.xssed.com/mirror/70523/	*fixed*

(Continued)

[21] https://chromium.googlesource.com/v8/v8/.

Table V. Continued

Vulnerable Domain	URL entry at XSSed.com	Status
panorama.ert.gr	http://www.xssed.com/mirror/70506/	*fixed*
www.cia.gov	http://www.xssed.com/mirror/70465/	*fixed*
www.eff.org	http://www.xssed.com/mirror/70354/	*unfixed*
roma.corriere.it	http://www.xssed.com/mirror/71241/	*unfixed*
www.slideshare.net	http://www.xssed.com/mirror/71127/	*fixed*
www.truste.com	http://www.xssed.com/mirror/71015/	*unfixed*
www.millenniumbank.ro	http://www.xssed.com/mirror/71460/	*unfixed*
quicktrip.olympicair.com	http://www.xssed.com/mirror/71369/	*unfixed*
blogs.oreilly.com	http://www.xssed.com/mirror/71319/	*unfixed*
search.indiatimes.co	http://www.xssed.com/mirror/71201/	*unfixed*
www.mysql.com	http://www.xssed.com/mirror/71496/	*fixed*
www.accionpyme.mecon.gov.ar	http://www.xssed.com/mirror/68499/	*unfixed*
reg.sun.com	http://www.xssed.com/mirror/71687/	*unfixed*
www.clickbank.com	http://www.xssed.com/mirror/71689/	*unfixed*
support.ts.fujitsu.com	http://www.xssed.com/mirror/69286/	*unfixed*
jobs.orange.com	http://www.xssed.com/mirror/70165/	*unfixed*
www.wi-fi.org	http://www.xssed.com/mirror/64296/	*unfixed*
www.ema.europa.eu	http://www.xssed.com/mirror/70197/	*unfixed*
sandiego.bbb.org	http://www.xssed.com/mirror/70320/	*unfixed*
www.energyrating.gov.au	http://www.xssed.com/mirror/67540/	*unfixed*
help.comodo.com	http://www.xssed.com/mirror/71276/	*unfixed*
leibniz.stanford.edu	http://www.xssed.com/mirror/71178/	*unfixed*
west.stanford.edu	http://www.xssed.com/mirror/71175/	*unfixed*

ACKNOWLEDGMENTS

We thank Georgios Gousios for helping us port the Crawljax framework. We also thank George Argyros, Vasileios P. Kemerlis, Michalis Polychronakis, and the anonymous reviewers for their insightful comments.

REFERENCES

Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, New York, NY, 1–10.

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 571–580.

Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. 2010. xJS: Practical XSS prevention for web application development. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, 13–13.

Elias Athanasopoulos, Vasilis Pappas, and Evangelos Markatos. 2009. Code-injection attacks in browsers supporting policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*.

Adam Barth, Juan Caballero, and Dawn Song. 2009. Secure content sniffing for web browsers, or how to stop articles from reviewing themselves. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, 360–371.

Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, 91–100.

Jim Beechey. 2010. Application Whitelisting: Panacea or Propaganda. http://www.sans.org/reading-room/whitepapers/application/application-whitelisting-panacea-propaganda-33599. (2010).

Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. 2009. Automated security testing of web widget interactions. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM Symposium on The Foundations of Software Engineering (ESEC/FSE'09)*. ACM, New York, NY, 81–90.

Prithvi Bisht and V. N. Venkatakrishnan. 2008. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA'08: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, Berlin, 23–43.

Hristo Bojinov, Elie Bursztein, and Dan Boneh. 2009. XCS: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, New York, NY, 420–431.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 748–759.

Dorothy Elizabeth Robling Denning. 1987. An intrusion detection model. 13, 2 (Feb. 1987), 222–232.

Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC'09)*. IEEE Computer Society, Washington, DC, 382–391.

Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. 2013. deDacota: Toward preventing server-side XSS via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*. ACM, New York, NY, 1205–1216.

Mattia Fazzini, Prateek Saxena, and Alessandro Orso. 2015. AutoCSP: Automatically retrofitting CSP to web applications. In *Proceedings of the 37th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE'15)*. ACM, New York, NY.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Adding rigorous statistics to the java benchmarker's toolbox. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA'07)*. ACM, New York, NY, 793–794.

Matthew Van Gundy and Hao Chen. 2009. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless attacks: Stealing the pie without touching the sill. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 760–771.

Lon Ingram and Michael Walfish. 2012. TreeHouse: JavaScript sandboxes to helpweb developers help themselves. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, 13–13.

Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. ACM, New York, NY, 601–610.

Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, New York, NY, 66–77.

Martin Johns and Christian Beyerlein. 2007. SMask: Preventing injection attacks in web applications by approximating automatic data/code separation. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*. ACM, New York, NY, 284–291.

Martin Johns, Björn Engelmann, and Joachim Posegga. 2008. XSSDS: Server-side detection of cross-site scripting attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*. IEEE Computer Society, Washington, DC, 335–344.

David Johnson, Alexei White, and Andre Charland. 2007. *Enterprise AJAX: Strategies for Building High Performance Web Applications*. Prentice Hall PTR, Upper Saddle River, NJ.

Sammy Kamkar. 2005. Technical Explanation of The MySpace Worm. (2005). http://namb.la/popular/tech.html.

Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

Angelos D. Keromytis. 2009. Randomized instruction sets and runtime environments: Past research and future directions. *IEEE Security and Privacy* 7, 1 (Jan. 2009), 18–25.

Sebastian Lekies, Ben Stock, and Martin Johns. 2014. A tale of the weaknesses of current client-side XSS filtering. Presented at Black Hat Europe 2014.

Mike Ter Louw and V. N. Venkatakrishnan. 2009. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP'09)*. IEEE Computer Society, Washington, DC, 331–346.

Mozilla Developer Network (MDN). 2015. JavaScript Reference and Global Objects. (2015). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

Ali Mesbah and Mukul R. Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. New York, NY, 561–570.

Ali Mesbah and Arie van Deursen. 2009. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Washington, DC, 210–220.

Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)* 6, 1 (2012), 3:1–3:30.

Leo A. Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, Washington, DC, 481–496.

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

Abbas Naderi, Mandana Bagheri, and Shahin Ramezany. 2014. Taintless: Defeating taint-powered protection tachniques. Presented at Black Hat USA 2014.

Yacin Nadji, Prateek Saxena, and Dawn Song. 2006. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE Computer Society, Washington, DC, 463–472.

Susanta Nanda, Lap-Chung Lam, and Tzi-cker Chiueh. 2007. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware Companion (MC'07)*. ACM, New York, NY, Article 19, 20 pages.

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*. 736–747.

Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. 2008. SOMA: Mutual approval for included content in web pages. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM, New York, NY, 89–98.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, Washington, DC, 75–84.

Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP aspis: Using partial taint tracking to protect against injection attacks. In *Proceedings of the 2nd USENIX Conference on Web Application Development (WebApps'11)*. USENIX Association, Berkeley, CA, 2–2.

Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS'09)*. ACM, New York, NY, 47–60.

Tadeusz Pietraszek and Chris Vanden Berghe. 2006. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection (RAID'05)*. Springer-Verlag, Berlin, 124–145.

Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. 2007. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web* 1, Article 11, issue 3 (September 2007).

Leonard Richardson and Sam Ruby. 2007. *Restful Web Services* (first ed.). O'Reilly.

Danny Roest, Ali Mesbah, and Arie van Deursen. 2010. Regression testing ajax applications: Coping with dynamism. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE Computer Society, Washington, DC, 127–136.

Hossein Saiedian and Dan Broyle. 2011. Security vulnerabilities in the same-origin policy: Implications and alternatives. *Computer* 44, 9 (Sept. 2011), 29–36.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, Washington, DC, 513–528.

Prateek Saxena, David Molnar, and Benjamin Livshits. 2011. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 601–614.

Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium (SSYM'05)*. USENIX Association, Berkeley, CA, 10–10.

Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, 921–930.

Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against DOM-based cross-site scripting. In *23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, 655–670.

D. Stuttard and M. Pinto. 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, New York, NY.

Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2011. WebJail: Least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, New York, NY, 307–316.

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

David Wagner and Paolo Soto. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS'02)*. ACM, New York, NY, 255–264.

Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2015. ZigZag: Automatically hardening web applications against client-side validation vulnerabilities. In *24th USENIX Security Symposium*. USENIX Association, Washington, DC, 737–752.

P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. 2009. SWAP: Mitigating XSS attacks using a reverse proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems (IWSESS'09)*. IEEE Computer Society, Washington, DC, 33–39.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. ACM, New York, NY, 237–249.

Received September 2015; revised May 2016; accepted May 2016