Static Detection of Cross-Site Scripting Vulnerabilities*

Gary Wassermann Zhendong Su

University of California, Davis
{wassermg,su}@cs.ucdavis.edu

ABSTRACT

Web applications support many of our daily activities, but they often have security problems, and their accessibility makes them easy to exploit. In cross-site scripting (XSS), an attacker exploits the trust a web client (browser) has for a trusted server and executes injected script on the browser with the server's privileges. In 2006, XSS constituted the largest class of newly reported vulnerabilities making it the most prevalent class of attacks today. Web applications have XSS vulnerabilities because the validation they perform on untrusted input does not suffice to prevent that input from invoking a browser's JavaScript interpreter, and this validation is particularly difficult to get right if it must admit some HTML mark-up. Most existing approaches to finding XSS vulnerabilities are taint-based and assume input validation functions to be adequate, so they either miss real vulnerabilities or report many false positives.

This paper presents a static analysis for finding XSS vulnerabilities that directly addresses weak or absent input validation. Our approach combines work on tainted information flow with string analysis. Proper input validation is difficult largely because of the many ways to invoke the JavaScript interpreter; we face the same obstacle checking for vulnerabilities statically, and we address it by formalizing a policy based on the W3C recommendation, the Firefox source code, and online tutorials about closed-source browsers. We provide effective checking algorithms based on our policy. We implement our approach and provide an extensive evaluation that finds both known and unknown vulnerabilities in real-world web applications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Languages, Security, Verification

Keywords

Static Analysis, Web Applications, Cross-Site Scripting, Input Validation

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

Web application vulnerabilities have greater impact than vulnerabilities in other kinds of applications and software. Attackers only need a web browser to access them, and normal web application use involves sensitive information, such as keys for secure sessions. Additionally, such vulnerabilities can serve as launching pads for other, more severe attacks on web users' local systems. The high prevalence of web application vulnerabilities compounds the problem of their impact. Cross-site scripting (XSS) is the class of web application vulnerabilities in which an attacker causes a victim's browser to execute JavaScript from the attacker with the privileges of a trusted host. In 2006, 21.5% of all newly reported vulnerabilities were XSS, making it the most frequently reported vulnerability of the year [2, 8]. This paper proposes the first practical, static, server-side approach to detecting XSS vulnerabilities that takes into account the semantics of input validation routines.

1.1 Causes of XSS Vulnerabilities

Several factors contribute to the prevalence of XSS vulnerabilities. First, the system requirements for XSS are minimal: XSS afflicts web applications that display untrusted input, and most do. Second, most web application programming languages provide an unsafe default for passing untrusted input to the client. Typically, printing the untrusted input directly to the output page is the most straightforward way of displaying such data. Static taint analysis addresses this second factor. It marks data values assigned from untrusted sources as tainted and reports a vulnerability if the application may display that data without first sanitizing it. The analysis considers untrusted data sanitized if that data has passed through one of a designated set of sanitizing functions. This paper addresses a third factor: proper validation for untrusted input is difficult to get right, primarily because of the many, often browser-specific, ways of invoking the JavaScript interpreter.

The MySpace worm, which infected more systems quickly than previous Internet worms (see Table 1), exploited a weakness in the MySpace input filters. The MySpace input filters prohibited all occurrences of the strings "`<script>`" and "`javascript:`" however, Internet Explorer concatenates strings broken by newlines and allows JavaScript within cascading style sheet tags, so the worm in-

Worm	Infected systems after 24 hours
Code Red I	359,000
Code Red II	275,000
Slammer	55,000
Blaster	336,000
MySpace worm	1,000,000

Table 1: Infected systems after 24 hours of worm propagation.

troduced its script by means of a string like:

```
<div style="background:url('java
script:...')">.
```

As if to further illustrate the point about proper input filtering being difficult to get right, several online services sell website visitor-tracking code for sites that officially forbid JavaScript, such as Xanga and MySpace. The sites that sell this code provide their service by continually finding new XSS vulnerabilities, so that whenever their current exploit stops working, they can use another vulnerability to inject their code.

Browser-specific ways of invoking the JavaScript interpreter exist because browsers handle permissively pages that are not standards compliant. In the early days of the web, this design decision on the part of the browser implementers seemed good because it enabled browsers to make a "best effort" to display poorly written pages. However, as the JavaScript language became more standardized and its use increased, this decision has exacerbated the XSS problem.

1.2 Current Practice

Currently, XSS scanners rely on either testing or static taint analysis. Automated testing is ill-suited to finding errors in input validation code, because even flawed validation code catches most malicious uses, and exploits must be crafted specifically for a certain validation's weakness in order to work. Taint analysis takes as input a list of functions designated as sanitizers, but it does not perform any analysis on them, so it will not catch errors caused by weak input validation.

1.3 Our Approach

This paper proposes an approach to finding not only XSS vulnerabilities due to unchecked untrusted data but also XSS vulnerabilities due to insufficiently-checked untrusted data. The approach has two parts: (1) an adapted string analysis to track untrusted substring values, and (2) a check for untrusted scripts based on formal language techniques.

Standard string analysis generates a formal language representation (*e.g.*, a context free grammar) of the possible string values a program may generate at a certain program point [1]. String-taint analysis not only represents the set of string values a program may generate, it also annotates the formal language representation with labels that indicate which substrings come from untrusted sources. Our string-taint analysis uses context free grammars to represent sets of string values and models the semantics of string operations using finite state transducers [18, 25].

The second phase of our approach enforces the policy that generated web pages include no untrusted scripts. In order to generate the right low-level description of this high-level policy, we must consider how web browsers' layout engines parse web documents, and under which circumstances they invoke the JavaScript engine. In order to generate the policy description, we studied the Gecko

lib.inc.php

```
481 function stop_xss($data) {
482
524  /* Get attribute="javascript:foo()" tags.
525   * catch spaces in the regex
526   * /(=|url\()("?)[^>]*script:/
527   */
528   $preg = '/(=|(U\s*R\s*L\s*\())\s*' .
529   '("|\')?[^>]*\s*' .
530   'S\s*C\s*R\s*I\s*P\s*T\s*:/i';
531   $data = preg_replace(
532       $preg, 'HordeCleaned', $data);
533
543  /* Get all on<foo>="bar()".
544   * NEVER allow these. */
545   $data = preg_replace(
546       '/([\s"\']+on\w+)\s*=/i',
547       'HordeCleaned=', $data);
548
550  /* Remove all scripts. */
551   $data = preg_replace(
552       '|<script[^>]*>.*?</script>|is',
553       '<HordeCleaned_script />', $data);
554
555   return $data;
556 }
```

projects_stats_pop.php

```
21 $use     = (int) $_REQUEST['use'];
22 $module  = stop_xss($_REQUEST['module']);
23
62 if ($use) {
63   echo '<br>';
71 } else {
72   $hiddenfields = "<input type='hidden' "
73     . "name='module' value='$module'/>\n";
74   echo '
75     <form action="stats.php" method="get">
76       <div style="width:60%;float:right">
77         <input name="speichern" />
78         '.get_buttons().'
79       </div>
80       '.$hiddenfields.'
81     </form>';
82 }
```

Figure 1: Vulnerable PHP code.

layout engine, which Firefox and Mozilla use, looked at the W3C recommendation for scripts in HTML documents, and looked at online documentation for how other browsers handle HTML documents. We represent the policy using regular languages and check whether untrusted parts of the document can invoke the JavaScript interpreter using language inclusion.

This paper makes the following main contributions:

- It proposes an approach for finding XSS vulnerabilities due to weak input validation.
- It presents an algorithm based on the behavior of layout engines that checks (languages of) generated HTML documents for untrusted script.
- It evaluates the approach on several real-world PHP web applications and demonstrates that the tool scales to large web applications and finds known and unknown errors caused by weak input validation.

```
$data1 = $_REQUEST['module'];
$data2 = preg_replace(
      '/([\s"\']+on\w+)\s*=/i',
      'HordeCleaned=', $data1);
$module = $data2;

if ($use) {
  $output1 = '<br>';
} else {
  $hiddenfields = "<input value='$module'/>";
  $output2 = ' </div> '.$hiddenfields;
}
$output3 = φ($output1, $output2);
```

Figure 2: Code in SSA form.

2. RUNNING EXAMPLE

Section 3 presents our analysis algorithm using the PHP code in Figure 1 as input. This section explains the example code and describes its vulnerability.

The program fragment in Figure 1 is pared down and adapted for display from PHProjekt 5.2.0, a modular application for coordinating group activities and sharing information and documents via the web. It is widely used and can be configured for 38 languages and 9 DBMSes. The untrusted data comes from the `$_REQUEST` array on lines 21 and 22. Line 22 assigns untrusted data to the `$module` variable after passing it through the `stop_xss` function.

The `stop_xss` function, which PHProjekt uses to perform input validation, uses Perl-style regular expressions to remove dangerous substrings from the input. The complete function also checks for alternate character encoding schemes, likely phishing attacks, and other dangerous tags, but these checks are omitted here for brevity. The primary ways to invoke the JavaScript interpreter are through script URLs; event handlers, all of which begin with "`on`"; and "`<script>`" tags. The function `stop_xss` removes these three cases with the regular expression replacements on lines 531, 545, and 551, respectively.

The W3C recommendation for HTML attributes specifies that white space characters may separate attribute names from the following '`=`' character. The regular expression on line 546 reflects this specification: '`\w`' represents word characters (word characters include alphanumeric characters, '_', and '.'), and '`\s`' represents white space characters. However, the Gecko layout engine's HTML parser permits arbitrary non-word characters between the attribute name and the '`=`' character. Consequently, if an input string includes a substring such as "`' onload#=`" followed by arbitrary script, it will pass the filter on line 545 but will cause untrusted JavaScript to be executed when the output page is viewed in Firefox.

3. ANALYSIS ALGORITHM

This section focusses on line 546 of the program in Figure 1 as it presents the algorithm.

3.1 String-taint Analysis

The string-taint phase of our analysis comes from previous work [25] and is based on Minamide's string analysis algorithm [18]; we review the main steps here to illustrate our approach. The first phase of the string-taint analysis translates output statements (*e.g.*, `echo` statements) into assignments to an added output variable, and translates the program into static single assignment (SSA) form [3] in order to encode data dependencies. Figure 2 illustrates this translation on the example code, omitting and simplifying several parts of the program for the sake of presentation. The special statement "ϕ ($output1, $output2)" is called a ϕ (Phi) function, which selects either $output1 or $output2 depending on the control flow of the preceding **if** statement.

$$
\begin{aligned}
&\textit{REQUESTmodule}^T \rightarrow \Sigma^* \\
&\textit{data1} \rightarrow \textit{REQUESTmodule} \\
&\textit{data2} \rightarrow \texttt{preg\_replace(}\ \texttt{/([\textbackslash s"\textbackslash']+on\textbackslash w+)\textbackslash s*=/i,} \\
&\qquad\qquad\qquad\qquad \texttt{HordeCleaned=},\ \textit{data1}\texttt{);} \\
&\textit{module} \rightarrow \textit{data2} \\
&\textit{output1} \rightarrow \texttt{
} \\
&\textit{hiddenfields} \rightarrow \texttt{<input value='}\textit{module}\texttt{'/>} \\
&\textit{output2} \rightarrow \texttt{</div>}\ \textit{hiddenfields} \\
&\textit{output3} \rightarrow \textit{output1} \mid \textit{output2}
\end{aligned}
$$

Figure 3: Productions for extended CFG.

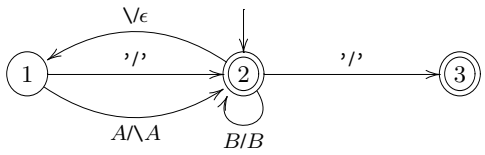

Figure 4: A finite state transducer representation of the `stripslashes` function; $A \in \Sigma \backslash \{'\}$, $B \in \Sigma \backslash \{\backslash\}$.

Because the SSA form encodes data dependencies, the next phase of the string-taint analysis drops control structures, translates assignment statements into grammar productions, and labels untrusted data sources. This phase constructs an *extended* context free grammar (CFG); it is extended in the sense the grammar productions' right hand sides may contain string functions. Figure 3 shows the extended CFG for our example, with *output3* as the start symbol. In Figure 3, *REQUESTmodule* is labeled with a taint annotation indicating that substrings derived from that non-terminal are untrusted. The only string function in our example grammar is `preg_replace`, which takes three arguments: a pattern, a replacement, and a subject. It searches for instances of the pattern in the subject and replaces them with the replacement. In general the replacement may reference and include parts of the string that the regular expression pattern matched, but in this example it does not.

In order to construct a CFG from the extended CFG, the last phase of the string-taint analysis constructs the CFGs for the arguments to the string operations and models the string operations' semantics using finite state transducers (FSTs). FSTs are finite state automata (FSAs) that produce output on transitions. Many formal language algorithms for FSAs can be adapted to FSTs. To introduce FSTs, we use `stripslashes`, which is a simpler example than `preg_replace`. Figure 4 shows an FST that models precisely the semantics of `stripslashes`, which removes from a string slashes ('\') that escape quotes. The transition labeled "$\backslash/\epsilon$" reads '\' outputs 'ϵ.' On the input word "`O\'Brian`," this FST will output "`O'Brian`," or equivalently, "`O'Brian`" is the *image* of "`O\'Brian`" over this FST.

The image relations that an FST defines between words can be lifted naturally to languages. The image of a context free language represented by a CFG over an FST can be constructed using an adaptation of the CFL-reachability algorithm [22] to construct the intersection of a CFG and an FSA [9]. Our previous work provides the details on the algorithm to construct this image as well as how

REQUESTmodule^T → ♢
data1 → *REQUESTmodule*
data2 → *data1*;
module → *data2*
output1 → `<br>`
hiddenfields → `<input value='`*module*`'/>`
output2 → `</div>` *hiddenfields*
output3 → *output1* | *output2*

Figure 5: CFG with untrusted substrings summarized; see Section 3.2.4 for an explanation of '♢.'

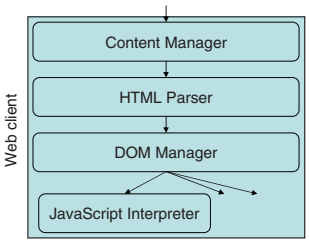

Figure 7: Client architecture.

to propagate taint annotations from the input CFG to the output CFG [25].

In the case of the extended CFG in Figure 3, the subject argument to `preg_replace` has Σ^* as its language, so the output language of the `preg_replace` function is:

$$\Sigma^* \setminus \mathcal{L}((\backslash s|"|')^+(o|O)(n|N)\backslash w^+\backslash s^*=).$$

We use regular expression notation to simplify the presentation of this example.

3.2 Preventing Untrusted Script

As Section 1 states, we seek to enforce the policy that no untrusted input may invoke the browser's JavaScript interpreter. This is the standard policy that server-side security mechanisms attempt to enforce, and the policy that untrusted input should not include any HTML mark-up subsumes it. JavaScript's highly dynamic nature as a prototype language inhibits checking from the server side whether untrusted JavaScript will stay within some safe boundaries. We leave as future work the formalization of "safe" JavaScript.

3.2.1 Challenges

Enforcing this policy to prevent XSS is challenging, and in particular, it is more difficult than preventing SQL injection. SQL injection, the second most reported vulnerability of 2006 (14%), is another input validation-based, web application vulnerability, and usually programmers attempt to enforce the policy that untrusted input can only be literals in generated SQL queries. The biggest challenge in preventing XSS is that web browsers support many ways of invoking the JavaScript interpreter, some of those according to the W3C recommendation, and some browser-specific. Web application programmers must account for even the browser-specific ways because they cannot control which browsers clients will use to view their pages. Additionally, if a web application programmer wants to allow some HTML mark-up, then every character has some legitimate use, so no single character can be escaped to prevent XSS. In contrast, the lexical definition of SQL literals and delimiters is relatively simple and standard, and web application programmers need not worry about interfacing with arbitrary DBMS's. Because of the many ways of invoking the JavaScript interpreter, statically checking sufficient input validation is more expensive and requires more careful engineering than checking for SQL injection vulnerabilities.

3.2.2 Constructing the Policy

In order to enumerate the ways an HTML document can invoke a browser's JavaScript interpreter, we examined three sources: the W3C recommendation, the Firefox source code, and online tutorials and documents. Figure 7 shows parts of the browser architecture in the context of the web document workflow that influence how the JavaScript interpreter can be invoked. After the input passes through the content manager, which handles downloads, caching, and protocols, it goes to the HTML parser. From the W3C recommendation, we gathered lexical rules for defining and separating tokens, and from an examination of Firefox's HTML parser, we modified the initially gathered lexical rules to allow non-word characters where previously only whitespace characters were allowed, as described in Section 2.

The HTML parser sends tokens to the DOM manager, which, among other tasks, constructs the DOM and calls the JavaScript interpreter. The W3C recommendation specifies two ways of including script in HTML: the "`<script>`" tag and event handlers. We found that the Firefox DOM manager also calls the JavaScript interpreter on the URL attribute of "`iframe`," "`meta`," and other tags. However, all sources we examined show that only tokens within a tag context (*i.e.*, between "`<`" and "`>`") can cause the DOM manager to call the JavaScript interpreter.

Because we are interested in whether untrusted input can invoke the JavaScript interpreter and not the string value of untrusted JavaScript code, we construct our policy in terms of the language of untrusted strings permitted or not permitted in a tag context. For example, we describe the language of tags whose names invoke the JavaScript interpreter using regular expressions such as:

```
[Ss][Cc][Rr][Ii][Pp][Tt]([^a-zA-Z0-9_.].*)?
```

The W3C recommendation includes eighteen intrinsic events (*e.g.*, `load`) and 31 events in total. Handlers for intrinsic events can be specified as attributes (*e.g.*, `onload`), but handlers for other events, such as DOM 2 events, can only be registered using "`addEventListener`" in a script. We therefore only check for handlers for intrinsic events. Firefox adds extra events (*e.g.*, `error`) and supports 36 for which handlers can be defined as attributes, all of which begin with "`on`." In order to simplify the regular expressions needed to identify these attributes, we state the policy not in terms of the whole tag, but only from the beginning of the attribute name onwards. To describe the language of attribute names that invoke the JavaScript interpreter (*i.e.*, event handlers and other attributes, such as `src`, that can introduce scripts), we therefore construct regular expressions such as

```
[Oo][Nn][Ll][Oo][Aa][Dd][^a-zA-Z0-9_.]*=.*
```

Note that this description incorporates the lexical rules we gathered in order to describe what may separate the attribute name from '`=`.'

3.2.3 Checking the Example

This section returns to our running example to show how we check the generated grammar against the policy we constructed. As Section 3.2.2 states, only tokens from within tags of an HTML document can invoke the JavaScript interpreter. Our algorithm consists of three main steps: (1) decode encoded characters within the grammar; (2) extract the string values of tags where all or part of the string is untrusted; (3) check those strings for script-inducing

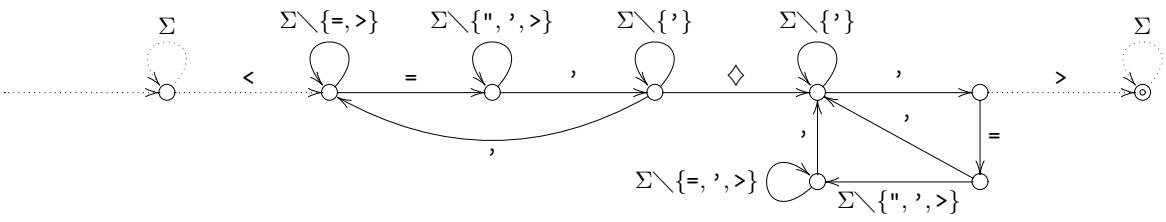

Figure 6: FST describing tags with untrusted data summarized by ◇ in an attribute value

substrings. Character encodings are not relevant to our example. In general, however, an FST can decode encoded characters (*e.g.*, translate "`&#97;`" into "`a`").

3.2.4 Extracting Untrusted Tags

For the second step, we must determine in which of the following syntactic contexts each untrusted substring can appear: character data, tag names, attribute names, and attribute values. Depending on which contexts each untrusted string can appear in, we check the string for different script-introducing values. In order to identify untrusted strings' contexts using formal language techniques, we summarize languages of untrusted strings by replacing labeled nonterminals' derivations with "fresh" terminals, *i.e.*, symbols not in Σ. Figure 5 shows the example grammar, where '◇' summarizes the untrusted substrings derivable from $REQUESTmodule^T$.

Figure 6 shows a nondeterministic FST that we use to check whether the untrusted substrings summarized by '◇' may appear in the context of single-quoted attribute values. In order to avoid having a cluttered figure, we use a different FST notation for Figure 6: solid transitions output the same character they read (*e.g.*, if the transition reads '`a`,' then it outputs '`a`'), and dotted transitions output ϵ. The image of the CFG in Figure 5 over this FST is the language of strings within a tag in the language of the original CFG that have an untrusted substring in an attribute value. We construct the CFG representation of this image, and use FSTs to remove all trusted script-introducing substrings from the language. Finally we replace '◇' with the original derivations of $REQUESTmodule^T$ so that any script-introducing substrings in the language are untrusted.

3.2.5 Identifying Script-Introducing Substrings

In order to check the CFG for untrusted script-introducing strings, we check whether it has a non-empty intersection with the regular expressions generated earlier that describe our policy. In the case of our running example, the intersection of the CFG with any of the regular expressions for event handlers is non-empty, so we discover a vulnerability.

4. EMPIRICAL EVALUATION

This section reports on our implementation and evaluation of our approach.

4.1 Implementation

We implemented our analysis by extending Minamide's PHP string analyzer written in O'Caml. In particular, we added support for tracking untrusted information flow by annotating nonterminals with a trust level and propagating them through assignments, through functions, and across FST and FSA intersections. We enhanced the tool's support for resolving dynamic includes so that we can give it a web page's top-level file and the analyzer will pull in included files as it encounters them. We added an option to make the default value for uninitialized variables be "untrusted any string" as opposed to null. We added support for several hundred PHP functions. We also implemented our algorithm to check for untrusted input that would invoke the JavaScript interpreter. The implementation of this algorithm consists of approximately 1000 lines of O'Caml.

Subject	Files	Lines Per File			Total lines
		mean	std dev	max	
Claroline	1144	148	248	5,207	169,232
FishCart	218	230	196	1,182	50,047
GecBBLite	11	29	30	95	323
PhPetition	17	159	75	281	2,701
PhPoll	40	144	112	512	5,757
Warp	44	554	520	2,276	24,365
Yapig	50	170	191	946	8,500

Table 2: Statistics on subjects' files.

Our goal in tracking untrusted information flow is to track direct information flow, not implicit information flows. In particular, we do not seek to perform an analysis that is sound with respect to covert channels. We choose this goal both because we assume that web users and not web programmers may be malicious and because of precedent, *i.e.*, static taint analysis for input validation vulnerabilities also has this goal.

4.2 Test Subjects

In evaluating our implementation, we sought to answer the following questions: How well does it scale on large, real-world web applications? How well does it check manually written input validation code, and how common are manual input validation errors?

We selected various web applications as test subjects in order to address each of these questions. In order to address the first question, we selected several PHP web applications of varying sizes whose names and sizes are listed in Table 2; Table 4 includes version numbers as well. One of the web applications, Claroline 1.5.3, is one of the largest open source, PHP web applications we have found (169 Kloc), and this particular version has several known vulnerabilities (CVE-2005-1374).

In order to evaluate the second question, we searched for PHP functions with "xss" in their name on the assumption that these functions likely represent manually written input validation specifically designed to prevent XSS. The functions we analyzed come from the following projects: VLBook, a light-weight guest book; Sendcard, an e-card system; Drupal, a content management system; LinPHA, a photo archive; Sugar Suite, a customer relationship management system; BASE, an engine to search and process a database of security events; FishCart, an online shopping cart and

Per File Resource Usage														
Additional Files Included			Time (h:mm:ss)								Memory (MB)		
			String Analysis				Policy Check						
mean	std dev	max	mean	std dev	max	sum	mean	std dev	max	sum	mean	std dev	max
10.0	16.5	148	0:00:11	0:01:52	0:21:58	3:20:29	0:00:08	0:00:45	0:18:48	2:08:30	71	117	1030
3.0	2.8	8	0:00:01	0:00:01	0:00:10	0:01:14	0:00:01	0:00:03	0:00:17	0:02:18	41	13	161
2.3	2.0	5	0:00:01	0:00:01	0:00:01	0:00:02	0:00:01	0:00:01	0:00:02	0:00:03	30	6	41
5.5	4.1	15	0:00:03	0:00:05	0:00:15	0:00:39	0:00:12	0:00:17	0:00:50	0:02:26	54	23	101
1.0	1.2	5	0:00:01	0:00:01	0:00:01	0:00:02	0:00:01	0:00:03	0:00:12	0:00:25	34	19	135
2.0	1.4	3	0:04:40	0:21:14	1:55:28	2:19:54	0:01:50	0:08:04	0:44:04	0:55:06	64	106	740
6.1	9.9	41	0:00:14	0:00:43	0:04:26	0:09:06	0:07:08	0:13:48	0:46:53	4:45:27	124	188	645

Table 3: Analysis results for test subjects. The subjects are listed in the same order as in Tables 2 and 4.

Subject	Direct				Indirect
	GPC		Uninit		
	t	f	t	f	
Claroline 1.5.3	32	43	38	25	42
FishCart 3.1	2	2	30	12	2
GecBBLite 0.1	1	1	0	0	7
PhPetition 0.3.1b	0	0	7	8	7
PhPoll 0.96 beta	5	6	0	0	0
Warp CMS 1.2.1	1	1	22	19	18
Yapig 0.95b	15	13	9	1	14

Table 4: Reported bugs.

catalog management system; PHPlist, a simple mailing list system; and PHProjekt, a groupware suite.

4.3 Evaluation Results

This section presents the results of our empirical evaluation to address the questions listed above.

4.3.1 Results on Programs

As stated in Section 4.1, we can analyze all PHP pages by giving the analyzer the top-level files. However, we found it easier to run the analyzer on each file (with dynamic includes still being resolved as before) even though this would involve some duplication of work. Running the analyzer on each file skews the average time and memory usage down (because many of the files only define values and are intended for inclusion in other files) and skews the total up. Table 3 shows the resource usage per file. In most cases the time for performing the string-taint analysis dominated the total time. The cases that took the longest for the string-taint analysis had string operations with cyclic dependencies. The cases that took the longest for the policy checking had many labeled nonterminals in the output grammar; each had to be checked individually. The included files column shows the average number of included files the analyzer parsed and analyzed on a given input file.

Table 4 shows the breakdown of bug reports from our experiments. Vulnerabilities are "direct" if an untrusted user can provide the data directly, whereas vulnerabilities are "indirect" if the data comes from a source such as a file or a database where untrusted data may have entered, but users cannot provide the value directly. "GPC" vulnerabilities come from GET, POST, or COOKIE variables, which the user can set. "Uninit" vulnerabilities come from uninitialized variables being used for output. If "export globals" is set, then each key in the associative GET, POST, and COOKIE arrays becomes the name of a variable, and its initial value is the value it maps to in the array. Therefore, if "export globals" is set and an uninitialized variable's values is displayed, a user can provide a GET parameter with that variable's name and the server will include the untrusted data into the generated page. For both GPC and uninit vulnerabilities, 't' and 'f' designate true and false vulnerabilities, respectively. We do not attempt to distinguish true and false vulnerabilities from indirect sources because we cannot determine whether or not it is possible for untrusted data to enter a given source.

Figure 8 shows one of the previously unreported vulnerabilities that our analysis discovered in Claroline; it is not due to weak input validation, but because the untrusted data passes function and file boundaries and is passed through an array, it would be easy to miss in a manual inspection. To illustrate the benefit of automated analysis, our analysis found 32 true GPC vulnerabilities, whereas CVE-2005-1374 lists only 10, although it does indicate that its list is not exhaustive. Most of the false positives our tool produced come from either spurious paths, or untrusted input being used in a conditional expression and the "taintedness" being propagated into the condition's branches. We could reduce the number of false positives by modifying the tool to output reports from untrusted conditionals as a different class of warnings.

In addition to the vulnerabilities listed in Table 4, Claroline has 77 vulnerabilities that do not fit naturally into any of the categories in the table. Claroline has a debugging mode that can be turned on and off by the administrator. When it is on, it displays all SQL queries before they are sent to the database, and that is the source of these 77. They are neither clearly true vulnerabilities, since Claroline would not normally run in debugging mode, and when it does, it would be under close control, nor are they false positives, because under specific circumstances XSS is possible with them. Note that these do not necessarily represent SQL injection vulnerabilities because an escaping function that properly sanitizes input for inclusion in SQL queries may not be adequate for preventing XSS.

Our tool failed to analyze some of the web applications we tried it on. It failed to analyze e107 0.75 (132,863 lines) because it failed to resolve certain alias relationships between variables whose values are used for dynamic features, including dynamic file inclusions. In the future we could address this by using a more conservative alias analysis. Our tool exceeded its memory limit of 4.5GB when attempting to analyze Phorum 5.1.16a (30,871 lines), because Phorum uses `preg_replace` tens of times consecutively in several cycles and with variables in all arguments. We expect that by redesigning the string analysis to retain only the precision it needs to check our policy, we could substantially reduce its memory requirement in such cases.

Project name	Time (h:mm:ss)		Memory (MB)	Vulnerability	
	String Analysis	Policy Check		Reported	Present
PHPlist 2.10.2	0:00:01	0:00:01	36	yes	yes
PHProjekt 5.2.0	0:00:36	0:00:39	167	yes	yes
Sendcard 3.2.2	0:00:15	1:01:11	2822	yes	yes
VLBook 1.21	0:00:01	0:00:27	232	yes	yes
Drupal 4.2.0	—	—	—	failed	yes
BASE 1.2.5	0:00:01	0:00:01	33	no	no
FishCart 3.1	0:00:01	0:00:01	39	no	no
SugarSuite 4.2.1	0:00:01	0:00:01	36	no	no
LinPHA 1.3.0	—	—	—	failed	no

Table 5: Analysis results for manual input validation functions. "failed" = "failed to analyze."

user_access_details.php

```
43  switch ($_GET['cmd'])
44  {
45    ...
57    case 'doc':
58      $toolTitle['subTitle'] =
59          $langDocument.$_GET['data'];
60    ...
69  }
70  claro_disp_tool_title($toolTitle);
```

claro_main.lib.php

```
435 function claro_disp_tool_title(
436     $titleElement, $helpUrl = false)
437 {
438   ...
474   if ($titleElement['subTitle'])
475   {
476     echo '<br><small>'.
477       $titleElement['subTitle'].'</small>';
478   }
479   echo '</h3>';
480 }
```

Figure 8: A vulnerability in Claroline 1.5.3.

4.3.2 Manual Validation

We had two goals in checking manually written input validation code: we wanted to see how our tool would perform in terms of time and memory usage and precision (can the tool do the job it is supposed to do?), and we wanted to get a sense of how prevalent insufficient input validation errors are (is the tool's job necessary?). Each of the nine test subjects we selected for this section had one function that performed all, or nearly all, of the application's input validation. We identified these functions, wrote small test files that call them, and sent those files to the analyzer.

Table 5 reports on how the tool performed on each of the nine subjects. SendCard stands out as being much more expensive to analyze than the rest. It uses several parameterized regular expression replacements (*i.e.*, the replacement includes a reference to a substring that the parameter matched) that cause the string analysis to generate a large and complex CFG to represent the possible strings it may generate. Some of the other subjects use regular expression replacements, but they are not parameterized. Our tool failed to analyze two of the files because they use PHP features that the tool does not support, and no simple modification to the files would retain their semantics and be analyzable by the tool. Except for the case of SendCard, the analyzer runs relatively efficiently on these subjects and produced precise results, so it appears to be practical.

Table 6 describes the weakness in the vulnerable filters and explains the effects of those that are not vulnerable. With the exception of the vulnerability in PHProjekt, these vulnerabilities were unknown to us, and, we believe, previously unreported. We are in the process of confirming them with the authors of the respective projects. Notably, all five of the nine subjects that allow any HTML mark-up (*e.g.*, the `<b>` tag) from untrusted input have vulnerabilities. The only ones without vulnerabilities prevent all untrusted mark-up. This suggests that writing web applications correctly is a difficult software engineering problem and that principled checking is necessary in real-world web applications.

4.4 Current Limitations

We discuss here some of the current limitations of our analysis.

Three main kinds of XSS exist: stored, reflected, and DOM-based; our analysis currently does not detect DOM-based XSS. Stored XSS occurs when the server stores untrusted data and later displays it; this kind of XSS commonly afflicts forums and online bulletin boards. Reflected XSS occurs when a server echos back untrusted input; this kind of XSS usually shows up in error messages. Unlike stored and reflected XSS, DOM-based XSS reads malicious data from the DOM, and the malicious data need not ever appear on the server. Detecting DOM-based XSS requires an analysis of the generated web page's semantics, not just its syntax. We expect that a reasonably precise approximation could be added on top of our framework, but currently our analysis does not include that check.

Our analysis checks web applications against the policy that no untrusted data should invoke the JavaScript interpreter, and we represent this policy as a black-list rather than a white-list. Omissions in black-list policies usually manifest themselves as difficult-to-detect security vulnerabilities, whereas omissions in white-list policies usually appear as disruptions of functionality, which show up rather quickly. We believe that our policy representation is correct for Gecko-based browsers, but we do not have a formal proof of its correctness with respect to the Gecko source code. Although a white-list policy could prove effective when designed for specific web applications that expect an easy-to-represent language of inputs, one main factor inhibits the use of a white-list policy in the general case. A regular language representation of all input that is valid HTML and does not invoke the JavaScript interpreter would be huge and likely impractical for language inclusion/intersection checks. Additionally, a white-list policy would always report errors

Project name	Allows some HTML mark-up	Has XSS Vulnerability	Vulnerability description
PHPlist 2.10.2	yes	yes	Filter only removes "script" tag
PHProjekt 5.2.0			Event handler filter only matches handlers with arbitrary white space between the handler name and the '='
Sendcard 3.2.2			Event handler filter only matches handlers with no characters between the handler name and the '='
VLBook 1.21			Event handler filter only matches handlers preceded with a space (' ')
Drupal 4.2.0			Event handler filter only matches handlers with arbitrary white space between the handler name and the '='
BASE 1.2.5	no	no	Filter wraps htmlspecialchars, prevents untrusted HTML mark-up
FishCart 3.1			Filter removes special characters, prevents untrusted HTML mark-up
SugarSuite 4.2.1			Filter encodes special characters, prevents untrusted HTML mark-up
LinPHA 1.3.0			Filter permits only alphabetic characters

Table 6: Explanation of (absense of) vulnerabilities for manual input validation functions.

in manually written input validation routines that enforce black-list policies, as the manually written code that we have seen does. However, even a weak black-list policy based solely on the W3C recommendation will help to uncover more vulnerabilities than a standard taint analysis will.

Our string analysis-based tool cannot handle arbitrarily complex and dynamic code. For example, because it does not track information flow across web page visits, it loses precision when the web application performs operations and calls functions based on the values of session variables. The tool also cannot verify input validation routines based on manually written HTML parsers and manipulators. Finally, the tool does not support some of PHP's features, such as array arguments in string replacement functions.

5. RELATED WORK

We classify related work into server-side and client-side techniques.

5.1 Server-Side Validation

Previous work server-side techniques to address web application vulnerabilities generally emphasizes static taint analysis or string analysis.

5.1.1 Static Taint Analysis

Most XSS vulnerabilities come from absence of input validation rather than insufficient input validation. Static taint tracking is designed to detect this kind of vulnerability. Because static taint analysis does not address string values, it can usually be applied to finding XSS vulnerabilities as well as finding SQL injection vulnerabilities, even if an implementation was designed for only one kind of vulnerability. Huang et al. presented one of the first taint analyses for web applications and applied it to SQL injection [11]. They used a CQual-like [4, 5] type system to propagate taint information through PHP programs. Livshits and Lam [17] used a precise points-to analysis for Java [26] and queries specified in PQL [16] to find paths in Java programs that allow "raw" input to flow into HTML output, file paths, and SQL queries. Both of these tools are sound with respect to the policy they enforce and the language features they support, and both find many vulnerabilities. However, both consider all values returned from designated filtering functions to be safe. Because the policy they use specifies nothing about the context of the user input or the input's value in that context, both techniques may miss real vulnerabilities. Additionally, Huang's type system does not support some of PHP's more dynamic features, in part because it does not track string values and supporting these features would likely result in excessively many false positives.

Jovanovic, Kruegel, and Kirda designed Pixy to propagate limited string value information in order to handle some of PHP's more dynamic features [13, 14]. They also address some of the characteristics of scripting languages with their precise and finely-tuned alias analysis. Xie and Aiken designed an SQL injection vulnerability analysis that gains scalability and efficiency in exchange for soundness by using block- and function-summaries [27]. Their analysis requires some interaction with the user—the user must provide the filenames when the analysis encounters a dynamic include statement, and the user must tell the analysis whether each regular expression encountered in a filtering function is "safe." Asking the user about regular expression filters may be acceptable for SQL injection vulnerabilities where the regular expressions enforce relatively simple lexical rules, but this will not be acceptable for the sequences of complex regular expressions that programmers use to prevent XSS. By analyzing the possible string values according to a formal specification using formal language techniques, we are able to make a stronger and more reliable guarantee that a given program is free of XSS vulnerabilities.

5.1.2 String Analysis

The other major category of server-side techniques related to our work is static string analysis. However, existing work on string analysis fails to consider the source of the substrings in the generated output. The study of static string analysis grew out of the study of text processing programs. An early work to use formal languages (*viz.* regular languages) to represent string values is XDuce [10], a language designed for XML transformations. Tabuchi et al. designed regular expression types for strings in a functional language with a type system that could handle certain programming constructs with greater precision than had been done before [23].

Christensen et al. introduced the study of static string analysis for imperative (and real-world) languages by showing the usefulness of string analysis for analyzing reflective code in Java programs and checking for errors in dynamically generated SQL queries [1]. They designed an analysis for Java that uses finite state automata (FSA) as its target language representation; they chose FSA because efficient algorithms exist to manipulate FSA. They also applied techniques from computational linguistics to generate good FSA approximations of CFGs [19]. Their analysis, however, does not track the source of data, and because it must determinize the

FSA between each operation, it is less efficient than other string analyzes and not practical for finding XSS vulnerabilities. Gould et al. used this analysis to type check dynamically generated database queries, but made approximations that would cause them to miss SQL injection vulnerabilities [6].

Minamide borrowed techniques from Christensen et al. to design a string analysis for PHP that does not approximate CFGs to FSA, so it can be more efficient and more accurate [18]. He also utilized techniques from computational linguistics (*viz.* language transducers) [20] to improve the precision of his analysis and model the effects of string operations, which are used frequently in scripting languages. He suggested using this analysis to check for XSS vulnerabilities, but his proposed technique checks the whole document for the presence of the "`<script>`" tag. Because web applications often include their own scripts, and because many other ways of invoking the JavaScript interpreter exist, this approach is not practical for finding XSS vulnerabilities.

5.2 Client-Side Mitigation

We organize our discussion of client-side approaches into those that enforce policies on the local behavior of JavaScript code and those that regulate outbound traffic based on information gained during the JavaScript's execution.

5.2.1 Local Behavior Enforcement

Hallaraker and Vigna use logging and auditing integrated into the JavaScript interpreter to enforce any policy specified with JavaScript code [7]. Yu et al. describe formally how to enforce arbitrary policies using interposition in a JavaScript-like language that is capable of code generation [28]. Both of these approaches impose low overhead because they integrate the enforcement mechanism into the JavaScript engine. However, they have no way to distinguish legitimate JavaScript from malicious JavaScript, and they leave open the question of which policy to enforce. BROWSERSHIELD prevents known browser vulnerabilities from being exploited by receiving a vulnerability description from a central server and interposing JavaScript wrappers to enforce the given policy [21]. BROWSERSHIELD has the advantage that it prevents real exploits, but it does not address the more general XSS problem, and because its enforcement mechanism consists of JavaScript wrappers to the JavaScript code, it may impose significant overhead.

BEEP (Browser-Enforced Embedded Policies) overcomes the problem of distinguishing trusted from untrusted JavaScript by providing a mechanism for the client to enforce either a black-list or white-list policy that the server sends specifying which scripts are trusted [12]. This coarse-grained policy language is similar to script signing, where servers can sign the scripts that they intend to be executed and request the clients execute only those. Deployment poses a practical limitation for BEEP, because both the client and the server must use it in order for it to work.

5.2.2 Outbound Traffic Regulation

NOXES regulates activity that occurs over the network, but it does not address local behavior of JavaScript code [15]. Its default rule prohibits dynamically constructed links from being followed, because these are the primary mechanism attackers use to communicate sensitive information. It enforces this policy by adding JavaScript code underneath the received web document.

Vogt et al. propose a client-side, information flow-based policy to mitigate the effects of XSS [24]. Their approach involves marking the client's confidential data as tainted, tracking tainted data through the client's browser, and only allowing tainted data to be sent to sites that have permission to access that data. This approach augments the same-origin policy that browsers already enforce. The same origin policy only permits servers to access information on a user's system that the server "owns" (*e.g.*, the cookie for that site); common examples of XSS exploits use injected code to send the data that a server owns elsewhere, and this policy prevents that.

This client-side mitigation complements server-side analysis, in the sense that server-side analysis protects many clients of one server, whereas Vogt et al.'s approach protects one client from many servers. However, even applying their approach universally does not suffice to solve the XSS problem completely. First, their approach addresses only one class of XSS attack; it does not mitigate the damage of other XSS-based attacks, such as port-scanning (where the sensitive information does not appear in the form of data), browser vulnerability exploitation, web page defacement, and browser resource consumption. Second, as web applications move closer to the desktop, identifying confidential data becomes a bigger and more error-prone task. For example, the Google Desktop indexes a user's local system and runs a web server on it in order to provide efficient search capabilities. It had an XSS vulnerability that, when coupled with an XSS vulnerability in Google that exposes the Google Desktop key, exposes the user's private data to remote attackers. Preventing this attack by tainting confidential data at the client-side would require designating confidential data at as fine a granularity as the DOM element level and as broad a scope as the user's whole system.

6. CONCLUSION

This paper presents a static analysis for finding XSS vulnerabilities that analyzes the root cause of XSS: weak input validation. Our analysis checks whether untrusted input to the server can invoke a client's JavaScript interpreter. We made a careful examination of the W3C recommendation, the Firefox source code, and other online source to express this policy in formal language terms. We have demonstrated that our approach can scale to large code bases and can detect known and unknown XSS vulnerabilities in real-world web applications with manually written input validation routines.

In the future, we would like to develop an analysis for web browsers to discover the set of strings that can cause their JavaScript interpreter to be invoked. Our own policy does not cover some exploits specific to browsers other than Firefox. Because of the complexity of layout engine code, we expect that such an analysis will require some interaction with the user.

Acknowledgements

We thank Earl Barr, Christian Bird, Mark Gabel, Lingxiao Jiang, and the anonymous reviewers for useful comments on earlier drafts of this paper.

7. REFERENCES

[1] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[2] S. Christey. Vulnerability type distributions in CVE, Oct. 2006. `http://cwe.mitre.org/documents/vuln-trends.html`.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment

form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[4] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, Atlanta, Georgia, May 1–4, 1999.

[5] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002. ACM Press.

[6] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 645–654, May 2004.

[7] O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, 2005.

[8] K. J. Higgins. Cross-site scripting: Attackers' new favorite flaw, September 2006. `http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1`.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley, Boston, MA, 2000.

[10] H. Hosoya and B. C. Pierce. Xduce: A typed xml processing language (preliminary report). In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 226–244, London, UK, 2001. Springer-Verlag.

[11] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[12] T. Jim, N. Swamy, and M. Hicks. Defeating scripting attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, 2007. ACM.

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for syntactic detection of web application vulnerabilities. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Ottowa, Canada, June 2006.

[15] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, 2006. ACM.

[16] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, June 2005.

[17] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[18] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW'05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[19] M. Mohri and M. Nederhof. Regular approximation of context-free grammars through transformation. *Robustness in Language and Speech Technology*, pages 153–163, 2001.

[20] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Meeting of the Association for Computational Linguistics*, pages 231–238, 1996.

[21] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[22] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[23] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language (extended abstract). In *Proceedings of TIP'02 Workshop on Types in Programming*, pages 1–18, July 2002.

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

[25] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007. ACM Press New York, NY, USA.

[26] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[27] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, pages 179–192, July 2006.

[28] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.