AutoCSP: Automatically Retrofitting CSP to Web Applications

Mattia Fazzini*, Prateek Saxena†, Alessandro Orso*
* Georgia Institute of Technology, USA, {mfazzini, orso}@cc.gatech.edu
† National University of Singapore, Singapore, prateeks@comp.nus.edu.sg

***Abstract*—Web applications often handle sensitive user data, which makes them attractive targets for attacks such as cross-site scripting (XSS). Content security policy (CSP) is a content-restriction mechanism, now supported by all major browsers, that offers thorough protection against XSS. Unfortunately, simply enabling CSP for a web application would affect the application's behavior and likely disrupt its functionality. To address this issue, we propose AUTOCSP, an automated technique for retrofitting CSP to web applications. AUTOCSP (1) leverages dynamic taint analysis to identify which content should be allowed to load on the dynamically-generated HTML pages of a web application and (2) automatically modifies the server-side code to generate such pages with the right permissions. Our evaluation, performed on a set of real-world web applications, shows that AUTOCSP can retrofit CSP effectively and efficiently.**

I. INTRODUCTION

Web applications are extremely popular, easily accessible, and often handle personal, confidential, and even sensitive data. These characteristics, together with the widespread presence of vulnerabilities in such applications, make them an attractive target for attackers. Cross-site scripting (XSS) in particular, is one of the most commonly reported security vulnerabilities in web applications and often results in successful attacks, whose consequences range from website defacing to theft of sensitive information.

The most common defense mechanisms against XSS are based on input filtering, which can be an effective approach, but is also error prone and often results in an incomplete protection. Content security policy (CSP), conversely, is a content-restriction scheme that is currently supported by all major browsers [1] and offers comprehensive protection against XSS attacks. In fact, the popularity of CSP is increasing, and companies such as Facebook and GitHub have started to move CSP to production. In a nutshell, web developers can use a CSP header to provide, for an HTML page, a declarative whitelist policy that defines which content should be allowed to load on that page. Unfortunately, simply enabling a default CSP for a web application can dramatically affect the application's behavior and is likely to disrupt the application's functionality. On the other hand, manually defining a policy can be difficult and time consuming.

To support an effective use of CSP, while both reducing developers' effort and preserving functionality, we propose AUTOCSP, an automated technique and tool for retrofitting CSP to web applications. Given a web application, AUTOCSP operates in four main phases. *First*, it marks as trusted all "known" values in the web application's server-side code (*e.g.,* hardcoded values) and exercises the web application while performing dynamic taint tracking. The result of this phase is a set of dynamically generated HTML pages whose content is annotated with (positive) taint information. *Second*, it analyzes the annotated HTML pages to identify which elements of these pages are trusted. (Basically, the tainted elements are those that come only from trusted sources.) *Third*, AUTOCSP uses the results of the previous analysis to infer a policy that would block potentially untrusted elements while allowing trusted elements to be loaded. *Fourth*, AUTOCSP automatically modifies the server-side code of the web application so that it generates web pages with the appropriate CSP.

To assess the usefulness and practical applicability of AUTOCSP, we developed a prototype that implements our technique and used it to perform an empirical evaluation. In our evaluation, we applied AUTOCSP to seven real-world web applications and assessed whether it can protect web applications without disrupting their functionality. Overall, the results of our evaluation are encouraging and show that AUTOCSP can effectively retrofit CSP to existing web applications so that (1) the applications are actually protected against XSS attacks, (2) their functionality is either not affected or minimally affected, and (3) their performance incurs a negligible overhead. Our results also show that automating this approach is cost-effective, as the number of changes to perform to the server-side code to retrofit CSP is large enough to make a manual approach expensive and error prone.

This work makes the following contributions:

- The definition of AUTOCSP, a general approach to retrofit CSP to web applications.
- A prototype implementation of AUTOCSP that can operate on PHP-based web applications and is available at http://www.cc.gatech.edu/~orso/software/autocsp.
- An empirical evaluation of AUTOCSP that shows the effectiveness and the practicality of our approach.

II. BACKGROUND AND MOTIVATING EXAMPLE

This section provides an overview of the security-related concepts needed to understand our approach and an example that we use to motivate our work.

A. Cross-Site Scripting (XSS)

Web applications are complex multi-tier applications that include a client and a server sides. Through a browser on the client side, users can issue HTTP requests to access functionality running on the server side, or backend. The server-side code processes the inputs contained in the HTTP requests and generates web pages with the content requested by the user. These dynamically-generated web pages usually consist of basic HTML entities together with JavaScript, CSS,

and other resources. When the browser receives these pages, it renders their content and executes the code they contain.

XSS attacks are injection attacks that take advantage of the dynamically generated content in a web page to insert malicious scripts into otherwise benign and trusted web pages. In these cases, malicious code coexists and executes together with benign code, as the web browser is unable to discern between the two. XSS vulnerabilities can be divided into different types, based on the methodology used to exploit them. XSS vulnerabilities of *persistent* type are those whose attacks are performed by injecting and permanently storing malicious script content within a resource of the targeted web application. Subsequently, when a user requests that resource, the malicious code executes as if it were generated by the web application itself (*i.e.,* as if it were trusted). This type of vulnerability is widespread because the web application logic allows for the possibility of using inline scripts and inline style directives within the dynamically generated HTML (*e.g.,* using the `<script>...</script>` construct). XSS vulnerabilities can also be of *reflected* type, when the corresponding attacks are built by having the server-side code processing the content of an HTTP request and attaching it verbatim to the requested web page. This type of vulnerability is also widespread, as application logic usually permits inline scripts and inline style constructs. The *DOM-based* type identifies vulnerabilities in which the attacks are created by injecting malicious payload directly into the DOM of the victim's web browser. Because of the way this type of exploits are constructed, the server-side of the application never sees the malicious script content. This class of vulnerabilities is exploitable due to a combination of JavaScript and HTML features. One of the most important among these features is the ability of JavaScript to execute code from strings (*i.e.,* `eval`). Another feature is that JavaScript code, while executing, can create new elements in the DOM, such as inline/event scripts and inline/attribute styles. Finally, *resource-based* XSS vulnerabilities can be exploited by placing malicious scripts within resources being fetched by the web browser while rendering a requested HTML page. Examples of these attacks are malicious script content injected into SVG files andq Chameleon attacks [2]. This type of vulnerability is common in web applications because these applications generally do not restrict the sources from which dynamic web pages can fetch their content.

B. Content Security Policy

Content security policy (CSP) [3], [4] is a declarative mechanism that can be used to protect web applications against several classes of XSS attacks. CSP can be seen as a content restriction scheme that web developers can use to specify what content can be included and how this content operates within dynamically-generated pages of a web application. CSP is enabled by adding, on the server side, a `Content-Security-Policy` header and corresponding content to the HTTP response that contains the protected resource. When a web browser receives the policy, it enforces it on the content of the web page being rendered.

A CSP is a set of directives in the form `directive-name: source-list`, where `directive-name` is the name of the directive, and `source-list` is the set of domains to which the specific directive applies. These directives define many aspects of a dynamic web page's content: which scripts are enabled, from where plugins can be loaded, which styles are allowed, from where media content can be retrieved, which resources can be framed, to which hosts the page can connect through scripts, and from where it is possible to load fonts. There are two special keywords that can be used in a policy: `unsafe-inline` and `unsafe-eval`. The first keyword enables inline scripts and inline style constructs in the protected web page. The second keyword enables the generation of code from string elements in the client-side code. These keywords can be very useful but, if allowed, may void the protection offered by CSP.

Several instances of XSS attacks can be blocked if CSP is properly added to the HTML pages of a web application. Persistent XSS attacks, in particular, can be blocked if `unsafe-inline` is not in the policy, and the policy only whitelists scripts and style content that were created by the developer of the web application. Reflected XSS attacks can be blocked if `unsafe-inline` is not excluded from the policy. The probability of DOM-based XSS attacks can be highly reduced if both `unsafe-eval` and `unsafe-inline` are excluded from the policy. Finally, resource-based XSS attacks can be prevented by CSP in two different ways. First, the attack can be blocked by preventing the resource that contains the malicious payload from being fetched (*i.e.,* the domain of the resource must not be whitelisted in the policy). Alternatively, if the resource needs to be fetched and is located on the host where the web application is running, the developer can specify a CSP that states that no script can be executed in the context of the resource.

Given the whitelist nature of CSP, in order to take full advantage of the protection mechanism it offers, web application developers need to (1) identify a policy for each of the web pages dynamically generated by the web application and (2) rewrite parts of the web application's server-side code to make sure they generate pages with the right policy. Such a manual process is not only time consuming but also error prone. Our approach aims to remove (most of) this burden from the developers' shoulders by automating this process.

C. Motivating Example

To motivate our work, we provide an example from a real-world web application called SCHOOLMATE—a school management system that has been downloaded over 16,000 times. Figure 1 shows the server-side code for the functionality that allows students to visualize the assignments related to one of their classes (slightly modified to make it self-contained and more readable). Figure 2 shows the HTML web page generated by this server-side code.

After generating the HTML page header, the server-side code adds an inline script to the HTML page (code lines 4–9, HTML lines 7–11). This script contains the functionality

```
1  <?php
2  print("<html>");
3  ...
4  print("<script>
5   function grades(){
6    document.student.page2.value=3;
7    document.student.submit();
8   }
9   </script>");
10 ...
11 print("<a class=\"menu\"
12  href=\"javascript:grades();\">
13  Grades</a>");
14 ...
15 while($assignment =
16  mysql_fetch_row($query)){
17  ...
18  print("<tr>
19   <td style=\"text-align: left;\">
20   $assignment[5]</td>
21   </tr>");
22  ...
23 }
24 ...
25 print("</html>");
26 ?>
```

Fig. 1: SCHOOLMATE server-side code snippet.

```
1  <html>
2   <head>
3   ...
4   </head>
5   <body>
6    ...
7    <script>
8     function grades(){
9     document.student.page2.value=3;
10    document.student.submit();
11   </script>
12   ...
13   <a class="menu"
14   href="javascript:grades();">
15   Grades</a>
16   ...
17   <tr>
18    <td style="text-align: left;">
19     <script>
20      alert("XSS");
21     </script>
22    </td>
23   </tr>
24   ...
25  </body>
26 </html>
```

Fig. 2: SCHOOLMATE original web page snippet.

```
1  <html>
2   <head>
3    ...
4    <script src="uri.js"></script>
5    <link rel="stylesheet"
6    type="css" href="sty.css"/>
7    ...
8   </head>
9   <body>
10   ...
11   <script src="external.js">
12   </script>
13   ...
14   <a id="uri" class="menu"
15   href="#">
16   Grades</a>
17   ...
18   <tr>
19    <td id="sty">
20     <script>
21      alert("XSS");
22     </script>
23    </td>
24   </tr>
25   ...
26  </body>
27 </html>
```

Fig. 3: SCHOOLMATE CSP-enabled web page snippet.

necessary to navigate the menu of the web application and is encoded as a constant string in the original program. Lines 11–13 print a link for accessing the functionality offered by the previous script (HTML lines 13–15). Also in this case, the element is encoded as a constant string in the program. Lines 15–23 consist of a loop that creates a table in the HTML page containing the assignments for a given class (HTML lines 17–23). In this case, the code prints a row and a column of a table by using a constant string for its structure and a variable for its content (line 20). The value of this variable is read from a database and represents comments of the class instructor. This makes the web application vulnerable to persistent XSS attacks (see Section II-A). Assume, for instance, that the value retrieved from the database is `<script>alert("XSS");</script>`. When this value is added to the generated HTML, it is interpreted as an inline script. At line 19, an inline style is applied to the column of the table, but this value is hardcoded in the program, so it cannot be modified by an attacker. Finally, the server-side code closes the HTML page at line 25 and terminates its execution.

This example lets us show why blindly enabling CSP on the generated web page would either affect its normal functionality or add inadequate protection against XSS attacks. Simply using CSP to block all executions of script and style content (*i.e.,* using `Content-Security-Policy: default-src 'none'` as the policy header) would block the XSS attach, but would also prevent normal users from accessing the menu functionality on the page. This is because both the inline script and JavaScript URI, respectively at lines 7–11 and 13–15 of Figure 2, would be blocked. Conversely, enabling the inline script, JavaScript URI, and inline style without modifying the web application (with policy header `Content-Security-Policy: default-src 'none'; script-src 'unsafe-inline'; style-src 'unsafe-inline'`) would preserve the page's functionality, but would allow the XSS attack to succeed. Figure 3 shows the CSP-enabled web page that would preserve the functionality of the web application while blocking the XSS attack. The correct CSP would be `Content-Security-Policy: default-src 'none';script-src domain;style-src domain`, where identifier `domain` represents the host on which the external JavaScript and CSS files linked in the HTML reside. As Figures 2 and 3 show, there are significant changes between the two HTML pages. The inline script at lines 7–11 in Figure 2 is transformed into the script at lines 11–12 in Figure 3. The content of the script is moved to an external file called `external.js`, which is enabled by the `script-src` CSP directive. The JavaScript URI at lines 13–15 of Figure 2 is also moved to an external file `uri.js` (line 4 in Figure 3) and linked using the newly introduced `id="uri"` expression at line 14 in the new web page. A similar transformation occurs for the inline style attribute of line 18 in Figure 2. The style content is moved to file `sty.css`, enabled using the `style-src` CSP directive, and activated through the newly introduced `id="sty"` expression (lines 5, 6, and 19 of Figure 3, respectively). The inline script that contains the malicious payload appears unchanged in Figure 3 at lines 20–22 and would be blocked by the browser, as the CSP associated with the web page does not allow inline scripts.

As this example shows, defining a suitable CSP can be a difficult, time-consuming, and error-prone task. In the next sections, we show how AUTOCSP can automate this process.

III. THE AUTOCSP APPROACH

In this section, we present our approach for retrofitting CSP to web applications. We first provide an overview of AUTOCSP, and then discuss its different phases in detail.

As we discussed in Section II-B, CSP offers a whitelist based content-restriction mechanism; that is, CSP (1) blocks

by default the loading/execution of any web-page node that is not specified in the policy but (2) allows for specifying which nodes are trusted and can thus be loaded/executed. The basic idea behind our approach is to automatically find trusted nodes in a dynamically generated web page by analyzing the execution of the server-side code that generates such page—nodes that are generated using only trusted sources are marked as trusted, whereas all other nodes are conservatively considered untrusted.

Figure 4 provides a high-level overview of our approach and shows its main phases. Given a web application and a set of test inputs, in its *dynamic tainting* phase, AUTOCSP marks as trusted all hardcoded values in the web application server-side code (plus, optionally, additional whitelisted sources specified by the developer) and performs dynamic taint analysis as the server-side code executes and generates web pages. Then, in the *web page analysis* phase, AUTOCSP analyzes a dynamically generated HTML page and the associated taint information to determine which parts of the page can be considered as trusted. In the *CSP analysis* phase of the approach, AUTOCSP processes an HTML page and its associated taint information to infer a policy that would block potentially untrusted parts while allowing trusted parts to be loaded in the browser. This phase also computes how HTML pages should be transformed in order to conform to the inferred policy. Finally, in its *source code transformation* phase, AUTOCSP modifies the source code of the web application so that it generates suitably transformed HTML pages (according to what it computed in the previous phase) in which the inferred CSPs are enabled.

In the remainder of this section, we describe AUTOCSP with the help of Algorithm 1, which represents the approach in pseudo-code. The inputs to AUTOCSP are a web application WA and a set of test inputs TS. For every test input ti in TS, AUTOCSP performs its dynamic tainting, web page analysis, and CSP analysis phases (lines 3–26). After processing every test input, the approach moves to its source code transformation phase (lines 27–33), which produces the final result: the transformed web application $\mathtt{WA}_{\mathtt{CSP}}$. We now describe the four phases of AUTOCSP in detail.

Algorithm 1: AutoCSP

```
   Input  : WA, web application; TS, set of test inputs for WA;
   Output: WA_CSP, web application using CSP;
1  begin
2      set ES := ∅
3      foreach input ti ∈ TS do
           /* Dynamic Tainting */
4          buffer HB, TB, SB := ∅
5          map TM := ∅
6          WA.set(ti)
7          while ¬WA.finish() do
8              instr i := WA.next()
9              i.execute()
10             taint(i, TM)
11             if i ≡ PRINT then
12                 HB.add(i.result)
13                 TB.add(TM[i.result])
14                 SB.add(i.line)
           /* Web Page Analysis */
15         dom DOM_A := parse(HB, TB, SB)
           /* CSP Analysis */
16         policy CSP := STRICT
17         foreach node n ∈ DOM_A do
18             if n.positive() then
19                 if n ≡ (SCR ∨ OBJ ∨ STY ∨ IMG ∨ MED ∨ FRM) then
20                     node n_new := toCSP(DOM_A, n)
21                     if n_new ≢ n then
22                         edit e_n := E_n(line_n(DOM_A, n), n, n_new)
23                         ES.add(e_n)
24                     CSP.allow(n_new)
25         edit e_CSP := E_CSP(line_CSP(DOM_A), CSP)
26         ES.add(e_CSP)
       /* Source Code Transformation */
27     webapp WP_CSP := WA
28     foreach edit e ∈ ES do
29         if e ≡ E_CSP then
30             transform_CSP(WP_CSP, e)
31         else if e ≡ E_n then
32             transform_n(WP_CSP, e)
33     return WP_CSP
```

A. Dynamic Tainting

This phase aims to determine, given a test input to the web application, which parts of the generated web page are *trusted* and which parts are *untrusted*, using a whitelist approach. We consider a DOM node of the generated HTML as trusted if it is defined by the developer or it is in full control of the developer. We consider all remaining content untrusted. Specifically, we use an approach called positive dynamic tainting, which we used successfully in previous work to counter SQL injection vulnerabilities [5]. Positive dynamic tainting marks the trusted data within an application (in this case, HTML fragments) and propagates taint marks as the application executes. Positive tainting, in contrast to more traditional negative tainting approaches, is a more conservative approach and fits naturally the whitelist approach behind CSP.

There are three main aspects that characterize a dynamic taint analysis: taint introduction, propagation policy, and taint checking. Taint introduction identifies and labels specific data within the program, called *taint sources*, using suitable taint marks. A taint *propagation policy* governs how taint marks propagate while the program is executing. Taint checking is the step in which particular actions are performed when taint marks reach special locations of the program called *taint sinks*. We present now the instance of positive dynamic taint analysis that we implemented in AUTOCSP. This part of AUTOCSP is covered by lines 4–14 in Algorithm 1.

1) Taint Sources: We introduce a taint mark for a given piece of data, hence marking it as positively tainted, whenever such data is hardcoded in the program. The rationale is that (1) server code normally uses hardcoded data, and in particular strings, to generate the different parts of an HTML page and (2) hardcoded data are defined by the developer, and thus trusted. In addition, our approach also allows developers to specify additional data that should be marked as trusted (*i.e.,* whitelisted), such as content originating from a specific column in a database or return values of specific functions. AUTOCSP keeps track of this taint information using a *taint map* (TM variable in the algorithm).

2) Taint Propagation: A taint propagation policy determines how the different instructions affect the taint marks associated with their operands. To compute accurate results, it is important to precisely model the semantics of such instructions. For each type of instruction, our taint propagation policy

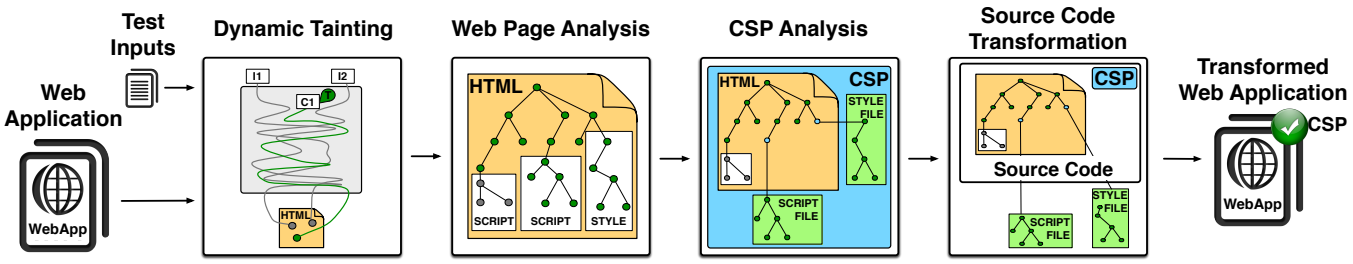

Fig. 4: High-level overview of AUTOCSP.

determines three aspects: the data defined by the instruction (*i.e.,* its output data), the data used by the instruction (*i.e.,* its input data), and a mapping function (*i.e.,* how the input data affects the output data). After executing an instruction `i` (*execute* function at line 9), AUTOCSP updates the taint marks associated with `i`'s output data based on the taint data associated with `i`'s input data and `i`'s mapping function (*taint* function at line 10).

3) Taint Sinks: Taint sinks are relevant instructions for the type of taint analysis being performed. For this reason, when executed, they trigger checking actions on the taint marks associated with their input data. In AUTOCSP, taint sinks consist of print instructions, as the data printed by the server-side code is what constitutes the web page that is then sent back to the client. By checking these sinks, AUTOCSP can check the taint marks associated to the different parts of the HTML pages dynamically generated by the application. In the server-side code, there are normally multiple places in which different parts of an HTML page are generated, and therefore multiple taint sinks. For each sink (line 11), AUTOCSP performs the following actions. First, it stores the characters of the generated HTML page (line 12) in an *HTML buffer* (`HB`). AUTOCSP also populates two other buffers: the *taint buffer* (`TB`) and the *source buffer* (`SB`). For each element in the HTML buffer, AUTOCSP creates a corresponding entry in the taint buffer (line 13) that specifies whether that element is trusted (if it has a taint mark) or untrusted (otherwise). Similarly, the approach stores into the source buffer the source code location of the statements that generated the content in the corresponding position of the HTML buffer (line 14). These three buffers are used by the subsequent phases of the approach.

B. Web Page Analysis

The second phase of our approach (represented by line 15 in Algorithm 1) analyzes the HTML, taint, and source buffers. In this phase, AUTOCSP parses (*parse* function at line 15) the HTML generated by the previous phase to build (and then operate on) its DOM representation, as a browser would do. In fact, using the same underlining model (DOM), AUTOCSP can better mimic the CSP enforcement mechanism that web browsers would apply. AUTOCSP actually produces an enhanced version of the DOM tree that we call the *annotated DOM tree* ($\texttt{DOM}_A$). AUTOCSP's parsing algorithm, which is based on the WHATWG specification [6], operates primarily on the HTML buffer, which contains the actual HTML content. The other two buffers are used to annotate the nodes of the resulting DOM tree. Basically, each of the tokens generated while parsing the HTML content contains two annotations: the first one indicates whether the token is trusted, whereas the second one indicates the locations in the server-side code of the statements that generated the token. After identifying all tokens, AUTOCSP produces a corresponding annotated DOM tree. Because each DOM node can correspond to multiple HTML tokens, to be conservative, AUTOCSP marks a node as trusted only if all of its corresponding tokens are.

C. CSP Analysis

This phase takes as input the annotated DOM tree, infers the CSP for it, and identifies how trusted DOM nodes should be transformed to comply with the inferred policy. The high-level algorithm for this phase corresponds to lines 16–26 of Algorithm 1. For each annotated DOM tree, this phase produces a set of HTML transformations and a CSP for the document. The HTML transformations and the CSP are then converted to source code edits (lines 22 and 25) and are passed to the final phase of the approach as the *edit set* (`ES`).

This phase starts by associating the strictest CSP possible to the HTML of the annotated DOM tree (line 16). This choice ensures the most effective protection offered by CSP against XSS. The initial CSP corresponds to `Content-Security-Policy: default-src 'none'`. This policy does not allow the protected HTML resource to use inline scripts, eval constructs, and inline styles. In addition, the policy does not allow the guarded resource to fetch any content from the web.

Once the initialization step is completed, AUTOCSP starts processing nodes in the annotated DOM tree (lines 17–24). The key idea of this phase is to incrementally add to the CSP trusted HTML elements. In addition, AUTOCSP transforms each such element, if necessary, so that the element's behavior is not disrupted by the enforcement mechanism of CSP. To do this, AUTOCSP identifies which elements in the annotated DOM tree relate to CSP and, if necessary, suitably transforms them.

Our approach identifies whether nodes of the annotated DOM tree relate to CSP according to what is stated in the CSP specification [4]. Specifically, there are six classes of elements that AUTOCSP identifies as related to CSP (line 19): nodes that (1) enable scripting, (2) load plugins, (3) define the style of the web page, (4) fetch images, (5) connect to media content, and (6) frame other resources.

If a DOM node that relates to CSP is trusted (*positive* function at line 18 returns `true`), it is enabled in the CSP and, if necessary, AUTOCSP identifies how to transform it to make it conform to the inferred CSP. The transformation process (*toCSP* function at line 20) is dependent on the type of node considered. If a transformation is necessary, a new *node edit* ($\mathtt{e_n}$) is added to the edit set (line 23). The node edit contains three pieces of information: the source-code location of the statement that generated the node (result of function $line_n$ at line 22), the original node (`n`), and the modified node ($\mathtt{n_{new}}$). After processing every node, AUTOCSP also creates a *CSP edit* ($\mathtt{e_{CSP}}$) and adds it to the edit set (line 26). A CSP edit contains two pieces of information: the source code location where the CSP header should be added (result of function $line_{CSP}$ at line 25) and the inferred CSP for the web page considered (`CSP`). In the reminder of this section, we provide details on the transformations performed on specific kinds of DOM nodes.

1) Script Nodes: There are different types of script nodes that can appear in the DOM and that must be handled in different ways. The first type of node is represented by inline script elements, that is, script nodes in the form `<script>...</script>`. AUTOCSP transforms this type of node to a script node in the form `<script src="..."></script>`. The new node fetches an external file stored on the web application server whose content is the original script. In this case, the CSP of the protected document gets extended by allowing the web page to fetch the script resource from the application server using the `script-src: host;` directive. The second type of node consists of event handlers attributes, such as HTML elements having attributes in the form `<button onclick="..."> </button>`. In this case, AUTOCSP replaces the original element with an element that does not declare the event handler and creates a script that adds the same event handler to that element. The new script code is placed in an external script file that is linked to the corresponding HTML document and such that (1) the script is activated when the DOM is loaded, and (2) the domain in which the file is stored is allowed in the CSP. The third type of node consists of elements having attributes that invoke a script using a JavaScript URI, such as `<a href="javascript:..."> </a>`. The transformation applied to this node is similar to the one applied for event handlers. First, the new script is placed in an external script file, linked to the HTML document, and activated when the DOM is loaded. Then, the domain where the file is stored is allowed in the CSP. The final type of script node is a node that links to an external script file in the form `<script src="..."></script>`. In this case, no transformation is applied. However, the domain of the linked script file is added to the CSP.

2) Style Nodes: Also for style nodes, AUTOCSP treats different types of nodes differently. The first type of node are inline style elements, that is, style nodes in the form `<style>...</style>`. AUTOCSP transforms this type of node to a node in the form `<link rel="stylesheet" type="text/css" href="..."/>`. The new node fetches an external style file that is stored on the web server and has the same content as the original style node. In this case, the CSP gets extended by allowing the document to fetch the style resource from the server using the `style-src: host;` directive. The second type of nodes are style attributes, such as HTML elements with attributes in the form `<p style="..."> </p>`. The approach replaces this type of node with a node without the style attribute and moves the style content to an external file. It then links the file to the corresponding HTML document and allows the domain where the file is stored in the CSP. The last type of style node consists of style elements that link to an external style file, that is, elements in the form `<link rel="stylesheet" type="text/css" href="..."/>`. In this case, no transformation is applied, but the domain of the linked style file is added to the CSP.

3) Other Nodes: The remaining nodes that relate to CSP are discussed together in this section, as AUTOCSP applies a similar analysis to all such nodes. This part of the approach analyzes classes of elements that (1) load plugins, (2) fetch images, (3) connect to media content, and (4) frame other resources. AUTOCSP identifies the resource to be fetched by the nodes and adds the domain where the resource is located to the CSP directive corresponding to the node being analyzed. In addition, if the whitelisted resource is coming from the same host of the web application, the approach attaches a `Content-Security-Policy: default-src 'none'` header to the resource being fetched. This is done to avoid XSS attacks that could piggyback on the resource being loaded [2]. The policy used for this resources is as strict as possible to avoid weakening the protection offered by AUTOCSP. Section V shows that this choice did not affect the functionality of the web applications in our evaluation.

D. Source Code Transformation

AUTOCSP's source code transformation phase modifies the server-side code of the web application so that it generates HTML pages (1) with the inferred CSPs enabled and (2) conforming to such CSPs. This phase takes as input the content of the edit set that was produced in the previous phase across multiple executions of the web application and returns a transformed CSP-enabled web application. In the rest of this section, we illustrate the two main parts of this phase, which correspond to lines 27–33 in Algorithm 1.

1) Enabling CSP: This part of AUTOCSP processes CSP edits created across multiple executions of the web application (function $transform_{CSP}$ at line 30). As mentioned in Section III-C, a CSP edit contains the inferred CSP and the source-code location where the inferred CSP header needs to be added. This latter is normally the code that generates the initial HTML content, as the CSP header needs to be sent to the web browser before any other HTML content.

Because our approach collects information across different executions, it is possible to have different CSPs associated to a single statement in the source code. In this case, our

technique takes the superset of the policies and uses the newly generated policy for all the web pages whose initial HTML content is generated at the current location. It is worth noting that combining the policies corresponding to different executions may result in a more permissive policy for a specific execution. However, we believe (and our experience confirms) that whitelisted content in one execution is unlikely to harm other, related executions of the same web application.

2) HTML Generation: This part of AUTOCSP processes the DOM-level transformations identified by the third phase of the approach and changes the server-side code of the web application to reflect these transformations in the generated web pages (function $transform_n$ at line 32). It does so by analyzing the node edits in the edit set. For each edit, AUTOCSP first extracts the source code location ($\texttt{stmt}_o$). It then modifies the code as follows. First, AUTOCSP introduces a new variable `out` and adds to the code a statement that assigns to that variable the HTML content generated by $\texttt{stmt}_o$. Second, AUTOCSP adds a statement $\texttt{stmt}_r$ that replaces in `out` the original DOM content with the new one. $\texttt{stmt}_r$ takes into account the fact that `out` might not contain the value corresponding to the original node because reached by an execution different from the one for which we generated the node edit. Finally, AUTOCSP adds a statement that prints the modified HTML content contained in `out`. If the new content links to new external scripts or style files introduced by AUTOCSP, this part of the analysis also creates the files with the proper content.

IV. IMPLEMENTATION

Our implementation of AUTOCSP can analyze PHP-based web applications and supports CSP 1.0 [4]. We implemented our general approach specifically for PHP because it is a language used for over 244 million applications and installed on over 2.1 million servers [7].

A. Dynamic Tainting

The dynamic tainting module consists of two main components. The first one is an extension to the ZEND engine Version 2.4, a PHP interpreter written in C code. The engine translates PHP scripts into opcodes and calls to C implementations of PHP libraries. In the version that we used, there are 1064 opcodes and 4887 library functions. We analyzed the semantics of all of the engine's opcode handlers but, to minimize our implementation effort, only analyzed the library functions used by our experimental benchmarks. Specifically, we analyzed how values flow through them, and implemented hooks to read relevant values during the taint process. Extracted input and output values, together with the opcode and function details, are passed to the dynamic tainting component, which handles taint introduction, taint propagation and taint checking.

The dynamic tainting component is written in Java, which is also the language used by the libraries used by the remaining modules of the tool. This component introduces taint marks for hardcoded values in the server-side code and for values originating from trusted locations specified by the developer in an XML configuration file. For each opcode and library function analyzed, the component implements the function that maps taint marks of values affecting the opcode computations to the values produced as a result of the computation. When the component processes `PRINT` and `ECHO` opcodes (taint sinks), it fills the three buffers produced by AUTOCSP's dynamic tainting phase, as described in Section III-A.

B. Web Page Analysis

Our tool implements the parsing phase of AUTOCSP by extending JSOUP 1.7 [8], an open source Java parser able to handle real-world HTML. It provides an easy to use, flexible, and efficient API for extracting and manipulating elements of the DOM. This module extends jsoup so that the output of the parsing process is the annotated DOM tree. The module also extends the library API of JSOUP to offer easy access to the information in the annotated DOM tree.

C. CSP Analysis

This module implements the third phase of AUTOCSP. It analyzes the annotated DOM tree, computes the CSP that applies to it, finds the transformations to generate HTML that complies to the inferred CSP, computes the CSP and node edits as mentioned in Section III-C, and stores them in an XML file. The code that handles the transformations computed by this module leverages the API added to JSOUP. It also uses FREEMARKER 2.3 template technology [9] to create scripts and style code that host the original content of transformed inline script and style nodes.

D. Source Code Transformation

This module parses the XML file produced by the CSP analysis module and creates a version of the web application that uses CSP. To do so, the module adds and modifies statements in the source code of the application (see Section III-D). The module uses ECLIPSE'S PDT library to create an abstract syntax tree for the code and applies changes to the application by modifying the AST. This module also creates the external script and style files introduced by AUTOCSP.

V. EMPIRICAL EVALUATION

To determine the practicality and effectiveness of our approach, we performed an empirical evaluation of AUTOCSP on a set of real-world web applications and targeted the following research questions:

- **RQ1:** Can AUTOCSP retrofit CSP to web applications and offer an effective protection against XSS attacks without disrupting the applications' functionality?
- **RQ2:** What is the effect of AUTOCSP on the performance of the retrofitted web applications?
- **RQ3:** How dependent is AUTOCSP's performance on the input used for its taint analysis?
- **RQ4:** Is automation actually needed to retrofit web applications?

The rest of this section presents our experimental benchmarks and setup and discusses our results.

TABLE I: Experimental benchmarks used in our evaluation.

Benchmark	*Type*	*Version*	*KLOC*
GALLERY	Photo Sharing	1.5	34.4
LINPHA	Photo Sharing	1.3	59.6
MYBB	Forum	1.6	105.9
OPENEMR	Medical Management	4.1	480
PHPLIST	Newsletter Management	2.10	35.4
SCHOOLMATE	School Management	1.5	6.5
SERENDIPITY	Blogging	0.8	49.6

TABLE II: Browser's console errors that occur while running the benchmarks under different CSP schemes.

Benchmark	*TI*	*None*	*Self*	AUTOCSP
GALLERY	16	175	68	0
LINPHA	43	231	136	0
MYBB	63	598	364	2
OPENEMR	113	699	533	11
PHPLIST	77	1224	273	1
SCHOOLMATE	90	16	8	0
SERENDIPITY	65	476	385	6

A. Experimental Benchmarks and Setup

For our empirical evaluation, we used real-world PHP web applications that were also used in previous work on XSS [10]–[15]. Among the applications used in these papers, we selected those that were either used in more than one paper or had a larger code base. This resulted in nine web applications, among which we had to discard two (PHORUM and PHPBB) because they dynamically create the server-side code; that is, in these applications, HTML pages are dynamically created by code that is also dynamically created, which is something that AUTOCSP does not handle at the moment.

Table I provides a summary description of the seven applications we considered. Columns *Benchmark*, *Version*, and *Type* provide name, version, and type of the web application. The last column, *KLOC*, reports the number of (thousand) lines of PHP code in the benchmark.

We deployed our benchmarks on a server machine with 3GB of memory, two Intel Pentium D CPU 3.00GHz processors, and running Ubuntu 10.04. We used ZEND 2.4 as the PHP application server. To answer our research questions, we ran our benchmarks against a set of representative inputs. Because the benchmarks did not include a test suite, and our dynamic taint analysis needs inputs, we deployed our benchmarks and asked five graduate-level students unfamiliar with AUTOCSP to explore the functionality of the web applications. The students's sessions were recorded by a Google Chrome extension we created. These recordings are available, together with another Chrome extension for replaying them, on our tool's website, provided in the Introduction.

B. Results

a) RQ1: To answer the part of RQ1 about AUTOCSP's effectiveness, we applied our approach to the benchmarks considered and ran a set of attacks against the retrofitted applications. Specifically, we created an initial set of seven attack vectors (*i.e.,* inputs) that exploited the seven known vulnerabilities in our benchmarks (one input per vulnerability). In addition, to evaluate AUTOCSP's effectiveness in more general terms, we also considered a set of additional vulnerabilities and inputs, as follows: we first randomly selected a set of 21 (3 times the number of applications considered, for lack of a better number) XSS attack vectors from various sources [16]–[19]; we then instantiated and injected vulnerabilities to enable the attack vectors in our 7 benchmarks; finally, we added the 21 attack vectors to our set of inputs.

At this point, we assessed the effectiveness of AUTOCSP by using it to protect the vulnerable web applications and running the protected applications against the input vectors on four different browsers: Chrome (v34), Firefox (v28), Opera (v20), and Safari (v7). All exploits were successfully blocked.

For the second part of RQ1, which is about the effects of AUTOCSP on the applications' functionality. We ran the retrofitted web applications against the inputs we described in Section V-A and checked whether that resulted in errors in the browser. Unfortunately, we could not compare AUTOCSP to any existing tool, as the most related existing approach, DEDACOTA [20], only works on applications written in ASP.NET. However, in order to have a baseline, we decided to implement two simple approaches: NONE, which simply enables on all generated HTML pages the strictest possible CSP policy (`Content-Security-Policy: default-src 'none'`) and SELF, which employes a policy that allows guarded resources to fetch content only from their same domain of origin (`Content-Security-Policy: default-src 'self'`). These simple baselines can give us an idea of what would happen if developers would simply apply CSP to a web application without analyzing it.

Table II reports, for each benchmark, the number of test inputs used (*TI*) and the number of unique (based on the client-side code location) client-side errors occurring when using approaches *None*, *Self*, and AUTOCSP (and not present when executing the original web applications without CSP). As expected, all benchmarks generate the largest number of errors with approach NONE, which prevents the web browser from executing any script, applying any style, and loading any external resource in a web page. The number of errors is significant also when using SELF. This is because all the benchmarks extensively use inline scripts and style directives in their HTML pages, and SELF prevents their execution.

AUTOCSP significantly reduces the number of errors compared to the other two approaches, but it does not completely eliminate them. More precisely, it transforms three of the web applications (GALLERY, LINPHA, and SCHOOLMATE) without introducing any error, and causes a limited number of errors in the other four applications. We investigated the reasons for these errors and found that they are of three types: (E1) executions of `eval` (*e.g.,* `var x = eval('...')`), (E2) client-side creation of inline script nodes in the DOM (*e.g.,* `document.write('<input onclick="...">')`), and (E3) client-side creation of style nodes in the DOM (*e.g.,* `document.write('<td style="...">')`). In the applications we considered, there are 4 instances of E1, 5 of E2, and 11 of E3.

TABLE III: Performance and modification data for the retrofitted applications in our empirical study.

Benchmark	T_o(*ms*)	T_t(*ms*)	E_{csp}(#)	E_n(#)	*F*(#)
GALLERY	339	391	2	76	12
LINPHA	128	125	2	67	11
MYBB	142	142	5	97	6
OPENEMR	288	288	31	319	52
PHPLIST	193	208	1	33	8
SCHOOLMATE	24	31	1	328	26
SERENDIPITY	473	532	5	103	16

These errors could be easily removed by adding policies `'unsafe-eval'` and `'unsafe-inline'` to CSP. Doing so, however, would reduce the protection offered by our approach against XSS. To avoid that, a more sophisticated analysis of the semantics of the JavaScript code in the HTML pages would be needed, which is something that AUTOCSP does not do at the moment. In future work, we plan to extend AUTOCSP with (1) an approach similar to the one proposed by Jensen and colleagues [21], to remove errors of type E1, and (2) automated script analysis, to remove errors of types E2 and E3. Currently, AUTOCSP simply reports such issues to the web application developers.

Based on these results we can answer RQ1 as follows: AUTOCSP is able, for the cases considered, to retrofit CSP to the web applications and effectively protect them against XSS attacks. It may generate false positives that require manual intervention in some cases, but the number of such false positives is significantly lower than if CSP were applied using a straw man approach.

b) RQ2: To answer RQ2, we compared the execution time of the retrofitted web applications with that of the original applications when run against our set of the test inputs. For each input, we measured the time from when the web browser issued an HTTP request to the moment in which the requested page was fully loaded in the web browser. The first part of Table III reports, for each application, the average execution time in milliseconds over one run of all inputs for the given application. Column T_o shows the average time for the original applications, while T_t shows the average time for retrofitted applications. As the results show, the overhead is mostly negligible from a practical standpoint. Moreover, it appears that the larger the subject, the lower the relative overhead, which is in fact not measurable for OPENEMR, our largest application.

We can therefore say, for RQ2, that the transformations introduced by AUTOCSP do not seem to significantly affect the performance of the retrofitted web applications.

c) RQ3: To answer RQ3, we compared the number of source code edits performed by AUTOCSP as more inputs are considered for its taint analysis. For each application considered, we computed the total number of edits when considering 0%, 20%, 40%, 60%, 80%, and 100% of the inputs of a server-side PHP file, where 100% means running that file against all of its inputs in the input set. Because we randomly selected the subset of inputs, we repeated the experiment 30 times and reported the average of these results in Figure 5. In the figure, AUTOCSP shows a similar trend for all of the web

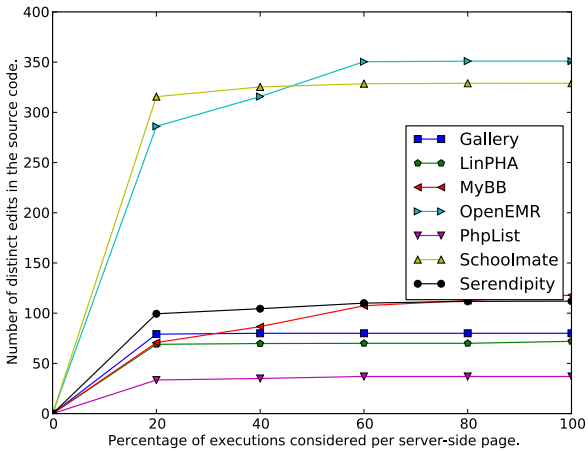

Fig. 5: Number of source code edits performed by AUTOCSP as more inputs are considered.

applications considered. In most cases, the number of edits converges after only 20% of the inputs have been considered, and in two cases after 60% are considered (for OPENEMR and MYBB, which are the two largest applications in our pool).

Overall, these results provide initial evidence that the approach is not strongly dependent on the specific inputs used.

d) RQ4: The second part of Table III lets us investigate RQ4. It shows the number of modifications performed by AUTOCSP on the benchmark applications: the number of CSPs added to the web applications (E_{csp}), the distinct number of DOM node edits (E_n), and the overall number of server-side source code files modified by AUTOCSP (*F*). For five out of the seven applications considered, our approach finds more than one CSP to apply in the server-side code, and it finds 31 in the worst case. The number of DOM node edits per web application is significant and could be in the order of a few hundreds even for a small web application (*e.g.*, SCHOOLMATE). The number of source code files affected by the changes is significant as well.

These results indicate that automation is necessary to retrofit CSP to existing web applications.

VI. LIMITATIONS OF CURRENT TECHNIQUE AND IMPLEMENTATION

Our current prototype tool does not fully support, in its source code transformation phase, web applications that can dynamically generate server-side code. More precisely, the tool cannot apply changes to source code statements that are dynamically generated by server-side code. (In this case, however, the prototype could still be used as a reporting tool; the generated report would help developers understand how to retrofit the computed CSP to the analyzed web application.)

In addition, AUTOCSP currently performs only a superficial analysis of the JavaScript code in the generated HTML pages. As a consequence, it may generate a CSP that is too strict and disrupts the application's functionality.

Despite these limitations, our empirical evaluation shows initial evidence that our approach can be practical, effective, and produce a low rate of false positives.

VII. Related Work

DEDACOTA [20] is the work most closely related to ours. It differs, however, in the nature of the approach, as it statically rewrites a web application to separate data and code in the generated web pages and applies CSP on the transformed version of the application. Specifically, it performs static data-flow analysis to approximate the HTML output of a web page and then rewrites the HTML such that inline JavaScript is stored in a separate JavaScript file. DEDACOTA does not deal with the problem of rewriting CSS code and inline event handlers (although it could probably be extended to do so). In our evaluation, we found that inline event handlers are a conspicuous part of the content that needs to be rewritten. We also found that rewriting inline event handlers strictly depends on the DOM node in which they appear, and that this information can be better determined with a dynamic approach. PIXY [13], [22] is a technique for detecting XSS vulnerabilities based on static data-flow analysis of PHP scripts. PIXY and AUTOCSP have similar goals, but operate differently: PIXY aims to find vulnerabilities in web applications, whereas AUTOCSP aims to protect them using CSP.

Server-side protection mechanisms (*e.g.,* [23]–[26]) range from techniques that provide defenses based on input sanitizer to methods that can differentiate between legal and illegal scripts. Di Lucca and colleagues [23] propose a mixed static and dynamic approach to find XSS vulnerabilities. Static analysis identifies web pages that might be vulnerable to XSS attacks, whereas dynamic analysis verifies whether such web pages are actually vulnerable. SANER [24] statically tracks unsafe information from sources to sinks and applies input sanitization. Subsequently, the technique tests for proper sanitization along the analyzed paths. This technique performs the opposite type of information-flow tracking than AUTOCSP (*i.e.,* untrusted vs trusted). SCRIPTGARD [25] performs context-sensitive sanitization to match the browser's parsing behavior. In a similar fashion, our approach tries to emulate browser parsing behavior when building the DOM tree for an HTML web page. XSS-GUARD [26] learns legal scripts generated by HTML requests and removes illegal content from the output of dynamically-generated HTML pages. Similarly, our approach tries to identify legal scripts but whitelists them instead of removing the untrusted one from generated web pages. In general, many of these techniques rely on some form of negative tainting and sanitization, which is error prone and can lead to false negatives.

Researchers also explored XSS protection mechanisms based on data-flow analysis (*e.g.,* [13], [22], [27]–[30]). Among those, dynamic tainting techniques (*e.g.,* [29], [30]) are most closely relate to our work. Nguyen-Tuong and colleagues [29] presented a technique that replaces the standard PHP interpreter with a modified one to track taint marks of string values. Based on the computed taint information, they check whether elements of the dynamically generated HTML pages were created from untrusted sources; if found, these elements are suitably removed or sanitized. The kind of dynamic tainting implemented by this technique differs from ours. We propagate taint marks associated to every type of data in the program, independently from its type. In addition, we create an extension of the PHP engine and do not modify the interpreter, which enables portability across multiple versions of PHP. CSSE is a method to detect and prevent injection attacks [30]. Their technique assigns metadata to user inputs, propagates and checks metadata based on the concepts of metadata-string operations and context-sensitive string evaluation. The technique also modifies the core of the PHP engine, which hinders portability. In addition, the technique might suffer from imprecision because it deals only with metadata of string values. Finally, the authors mention that their technique can protect against XSS attacks and provide one example for one class of such attacks. AUTOCSP, conversely, by relying on CSP, can protect against different classes of XSS attacks.

Client-side protection mechanisms (*e.g.,* [3], [31]) aim to enhance the client-side code. BEEP [31], in particular, whitelists scripts on the server side and specifies a security policy for every web page. The security policy enables or disables execution of scripts and is similar to CSP [3]. However, CSP handles more classes of HTML elements and provides better guarantees of protection on the client side, as it is a W3C standard implemented by all major web browsers.

VIII. Conclusion

We presented AUTOCSP, a novel approach for retrofitting CSP to existing web applications. Given a web application, AUTOCSP first performs positive dynamic tainting on the serve-side code of the application. It then uses the computed taint information to find trusted elements of dynamically generated HTML pages and infer a policy that would block potentially untrusted elements while allowing the trusted ones. Finally, it automatically modifies the server-side code of the web application so that it generates web pages with the appropriate CSP. To assess precision, practicality, and effectiveness of AUTOCSP, we implemented it in a tool that targets PHP web applications and performed an empirical evaluation on a set of real-world web applications. The results of our evaluation show that, for the cases considered, AUTOCSP was effective in retrofitting CSP to the existing applications while either preserving their functionality or minimally affecting it.

In future work, we will first address the limitations of our current implementation. We then plan to extend AUTOCSP so that it automatically updates the computed CSPs for a web application as developers modify the application during evolution.

Acknowledgements

We thank Abhik Roychoudhury for his input and several fruitful discussions. This work is partly supported by NSF awards CCF-1320783 and CCF-1161821, by funding from Google, IBM Research, and Microsoft Research to Georgia Tech, by the Ministry of Education, Singapore under Grant No. R-252-000-495-133, by a University Research Grant from Intel, and by a research grant from Symantec.

REFERENCES

[1] "Can I use... Support tables for HTML5, CSS3, etc," http://www.caniuse.com/contentsecuritypolicy, 2014.

[2] A. Barth, J. Caballero, and D. Song, "Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2009.

[3] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *Proceedings of the International World Wide Web Conference (WWW)*, 2009.

[4] "Content Security Policy 1.0," http://www.w3.org/TR/CSP/, 2014.

[5] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 1, pp. 65–81, 2008.

[6] "HTML Standard," http://www.whatwg.org/specs/web-apps/current-work/multipage/, 2014.

[7] "PHP: PHP Usage Stats," http://php.net/usage.php, 2014.

[8] "jsoup Java HTML Parser, with best of DOM, CSS, and jquery," http://jsoup.org/, 2014.

[9] "FreeMarker Java Template Engine - Overview," http://freemarker.org/, 2014.

[10] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[11] G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[12] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[13] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2006.

[14] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2011.

[15] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[16] "XSS Filter Evasion Cheat Sheet - OWASP," https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2014.

[17] "XSS (Cross Site Scripting) Prevention Cheat Sheet - OWASP," https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2014.

[18] "DOM based XSS Prevention Cheat Sheet - OWASP," https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet, 2014.

[19] "HTML5 Security Cheatsheet," https://html5sec.org/, 2014.

[20] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation," in *Proceedings of the ACM SIGSAC Conference on Computer amd Communications Security (CCS)*, 2013.

[21] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the Eval That Men Do," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[22] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," in *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[23] G. D. Lucca, A. Fasolino, M. Mastroianni, and P. Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications," in *6th IEEE International Workshop on Web Site Evolution (WSE)*, 2004.

[24] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2008.

[25] P. Saxena, D. Molnar, and B. Livshits, "ScriptGard: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[26] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," in *Proceedings of the International Conference on Detection of Intrusions and Malware (DIMVA)*, 2008.

[27] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective Taint Analysis of Web Applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[28] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Proceedings of the USENIX Security Symposium*, 2005.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting," in *Proceedings of the International Information Security Conference*, 2005.

[30] T. Pietraszek and C. V. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[31] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," in *Proceedings of the International World Wide Web Conference (WWW)*, 2007.