CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites

Xiang Pan[1] Yinzhi Cao[2] Shuangping Liu[1] Yu Zhou [1] Yan Chen[3,1] Tingzhe Zhou[2]
[1]Northwestern University, Illinois, USA
[2]Lehigh University, Pennsylvania, USA
[3]Zhejiang University, Zhejiang, China
{xiangpan2011, shuangping-liu, yuzhou2016}@u.northwestern.edu
{yinzhi.cao, tiz214}@lehigh.edu ychen@northwestern.edu

ABSTRACT

Content security policy (CSP)—which has been standardized by W3C and adopted by all major commercial browsers—is one of the most promising approaches for defending against cross-site scripting (XSS) attacks. Although client-side adoption of CSP is successful, server-side adoption is far behind the client side: according to a large-scale survey, less than 0.002% of Alexa Top 1M websites enabled CSP.

To facilitate the adoption of CSP, we propose *CSPAutoGen* to enable CSP in real-time, without server modifications, and being compatible with real-world websites. Specifically, *CSPAutoGen* trains so-called templates for each domain, generates CSPs based on the templates, rewrites incoming webpages on the fly to apply those generated CSPs, and then serves those rewritten webpages to client browsers. *CSPAutoGen* is designed to automatically enforce the most secure and strict version of CSP without enabling "unsafe-inline" and "unsafe-eval", i.e., *CSPAutoGen* can handle all the inline and dynamic scripts.

We have implemented a prototype of *CSPAutoGen*, and our evaluation shows that *CSPAutoGen* can correctly render all the Alexa Top 50 websites. Moreover, we conduct extensive case studies on five popular websites, indicating that *CSPAutoGen* can preserve the behind-the-login functionalities, such as sending emails and posting comments. Our security analysis shows that *CSPAutoGen* is able to defend against all the tested real-world XSS attacks.

1. INTRODUCTION

Cross-site scripting (XSS) vulnerabilities—though being there for more than ten years—are still one of the most commonly found web application vulnerabilities in the wild. Towards this end, researchers have proposed numerous defense mechanisms [12, 14, 17, 21, 30, 32, 40, 41] targeting various categories of XSS vulnerabilities. Among these defenses, one widely-adopted approach is called Content Security Policy (CSP) [41], which has been standardized by W3C [1] and adopted by all major commercial browsers, such as Google Chrome, Internet Explorer, Safari, and Firefox.

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978384

Though client-side adoption has been successful, server-side adoption of CSP proves more worrisome: according to an Internet-scale survey [45] of 1M websites, at the time of the study, only 2% of top 100 Alexa websites enabled CSP, and 0.00086% of 900,000 least popular sites did so. Such low adoption rate of CSP in modern websites is because CSP[1] requires server modifications. That is, all the inline JavaScript and `eval` statements need to be removed from a website without breaking its intended functionality, which brings extensive overhead for website developers or administrators.

To facilitate server deployment, in related work, deDacota [12] and AutoCSP [14] analyze server-side code using program analysis, infer CSPs, and modify those code to enable the inferred CSPs. Another related work, autoCSP[2] [17], infers CSPs based on violation reports and enforces the inferred CSPs later on. However, deDacota and AutoCSP—due to their white-box property—require server modification. Additionally, both approaches are specific to websites written in certain web languages. Another approach, autoCSP, does not support inline scripts with runtime information and dynamic scripts, and thus websites with those scripts cannot work properly. According to our manual analysis, 88% of Alexa Top 50 websites contain such script usages.

In this paper, we propose *CSPAutoGen*, a real-time, black-box enforcement of CSP without any server modifications and being compatible with real-world websites. The key insight is that although web scripts may appear in different formats or change in runtime, they are generated from uniform templates. Therefore, *CSPAutoGen* can infer the templates behind web scripts and decouple web contents in a script from the script's inherent structure.

Specifically, *CSPAutoGen* first groups scripts in webpages under a domain, and infers script templates, defined as training phase. Next, in the so-called rewriting phase, *CSPAutoGen* generates CSPs based on the webpage and templates, and then modifies webpages on the fly—which could be at a server gateway, an enterprise gateway or a client browser—to insert the generated CSPs and apply them at client browsers. Lastly, a client-side library added in the rewriting phase will detect additional scripts generated at client side during runtime and execute these scripts that match the templates. Below we discuss two important mechanisms used in *CSPAutoGen*:

[1]In this paper, unless specified, our definition of CSP refers to the strictest CSP, i.e., the default one with no "unsafe-inline" and "unsafe-eval" options enabled. Although CSP provides options like "unsafe-inline" and "unsafe-eval" for compatibility, these options are not safe and open doors for XSS attacks.

[2]Note that AutoCSP [14] and autoCSP [17] are two pieces of related work with the difference in their first letter capitalization.

- *Template Mechanism.* The proposed template—used to match incoming scripts at the rewriting and runtime phase—is composed of two parts: *generalized Abstract Syntax Tree (gAST)* and a type system. The former captures the inherent structure of scripts, e.g., *for* loop and *if* statement; the latter abstracts runtime generated information, e.g., content-specific Uniform Resource Locator (URL), to its corresponding type, such as URL type.
- *Secure JavaScript Transformation. CSPAutoGen* securely transforms common usages of JavaScript, such as dynamic scripts and inline scripts, to comply with the strictest CSP. Particularly, we propose a novel technique called *symbolic template* to securely execute dynamic scripts in `eval` or `eval`-like functions with "unsafe-eval" disabled. *CSPAutoGen* disables "unsafe-inline" as well: it imports pre-included inline scripts, i.e., these embedded in a script tag, as external files, and monitors DOM tree changes to re-import runtime-included inline scripts, i.e., these generated by DOM operations.

To evaluate *CSPAutoGen*, we focus on the following metrics and obtain corresponding results as follows:

- *Robustness.* Our evaluation on Alexa Top 50 websites shows that the median matching rate of unknown scripts is 99.2%, and templates can sustain high matching rate for at least 60 days.
- *Correctness.* Our evaluation on an open source framework shows that the accuracy of type inference is 95.9%.
- *Security.* Our evaluation on six real-word vulnerable web applications shows that *CSPAutoGen* can protect all of them against XSS attacks. Moreover, the evaluation on Alexa Top 50 websites shows that the CSP policies automatically generated by *CSPAutoGen* are more secure than the ones generated by website themselves: none of them support both "unsafe-inline" and "unsafe-eval".
- *Compatibility.* Our evaluation on Alexa Top 50 websites shows that all the 50 websites can be correctly displayed. In addition, we extensively explore the behind-the-login functionalities—such as sending emails and web search—of five popular websites: they all work properly.
- *Performance.* The performance evaluation shows that the median overhead of *CSPAutoGen*'s is as small as 9.1%.

The rest of the paper is organized as follows. Section 2 provides an overview of Content Security Policy. Section 3 and Section 4 present *CSPAutoGen*'s deployment model, overall architecture and design. Then in Section 5, we discuss the implementation details of *CSPAutoGen*. The evaluation is discussed in Section 6, where we evaluate the system's template robustness, security, compatibility and performance. Next, in Section 7, we discuss several related topics and challenges. Related work is presented in Section 8 and Section 9 concludes the paper.

2. BACKGROUND

Content Security Policy (CSP) [1, 41] is a declarative whitelist mechanism to protect websites against XSS attacks. Specifically, CSP allows website developers to make policies for each webpage and specify which contents are allowed to load and execute on each page. These policies are delivered to the client-side browser via `Content-Security-Policy` HTTP response header or in a `meta` element. When the client-side browser receives CSP policies, the browser, if it supports CSP, will enforce the received policies to protect users. Say, for example, a webpage protected by CSP policies that only allow scripts from its own server are injected with a snippet of malicious JavaScript via an XSS vulnerability. The malicious scripts are automatically blocked because they come from an origin that is unspecified in the CSP policies.

```
1 Content-Security-Policy: defaul-src 'self';
2 image-src 'self' *.yimg.com; object-src 'none';
3 script-src 'self' apis.google.com;
```

Code 1: An example of CSP policy.

Now let us introduce the details of CSP policies. A CSP policy is consisted of a set of `directives`. Each directive is in the form of `directive-name` and `directive-value`, where the former indicates the type of resource and the latter specifies the allowed source list for that resource type. Code 1 shows an example of CSP policy. In this example, the browser is only allowed to load images (specified by `image-src`) from **.yimg.com* and the page's current origin (specified by keyword 'self'), and scripts (specified by `script-src`) from *aps.google.com* and the current origin. No plugins are allowed in this page (specified by `object-src` and keyword 'none'), and other types of resources (specified by `default-src`) are only allowed to be from the current origin. In this paper, we discuss how to use *CSPAutoGen* to infer and enforce CSP policies for script type (i.e., `script-src`). *CSPAutoGen* can be also conveniently extended to support security policies for other resource types.

By default, CSP disables inline scripts and dynamic scripts, i.e., the function calls of `eval` and `Function`, as well as `setTimeout` and `setInterval` if their first arguments are not callable [1]. For the purpose of backward compatibility, CSP allows developers to specify keywords `unsafe-inline` and `unsafe-eval` in CSP policies: the former allows inline scripts, and the latter enables dynamic scripts. However, though convenient, these two keywords seriously mitigate the protection offered by CSP. For example, inline scripts open doors for reflected XSS, while dynamic scripts lower the bar for DOM-based XSS attacks. One of *CSPAutoGen*'s contributions is to enforce policies without setting `unsafe-inline` or `unsafe-eval`, while still preserving websites' functionalities.

There are two major levels of CSP that are commonly seen in mainstream browsers: Level 1, the mostly adopted version that provides the aforementioned functionalities, and Level 2 that introduce a new `nonce` feature. Specifically, the `nonce` allows an inline script if the script's hash or token (i.e., a random token assigned to each whitelisted inline script) is specified in the CSP policy. At the time we write the paper, Internet Explorer and Microsoft Edge do not support CSP Level 2. Our *CSPAutoGen* is compatible with both levels of CSP, because we only use the basic CSP functionalities.

3. OVERVIEW

In this section, we start by describing the system architecture, and then delve into three deployment models.

3.1 System Architecture

CSPAutoGen works in three phases: training, rewriting, and runtime. In the training phase, *CSPAutoGen* takes a bunch of webpage samples as a training set and generates templates. Then, in the rewriting phase, *CSPAutoGen* parses incoming webpages, generates corresponding CSPs based on the templates and rewrites webpages to include CSPs. In addition, *CSPAutoGen* also inserts templates generated in training phase and a client-side JavaScript library to each webpage. Lastly, in the runtime phase at the client-side, the injected CSPs are enforced by the browser, guaranteeing that illegitimate scripts are not executed. The previously-injected library and templates ensure the runtime-included scripts and dynamic scripts that match templates can be imported. Now, we introduce these three phases in detail.

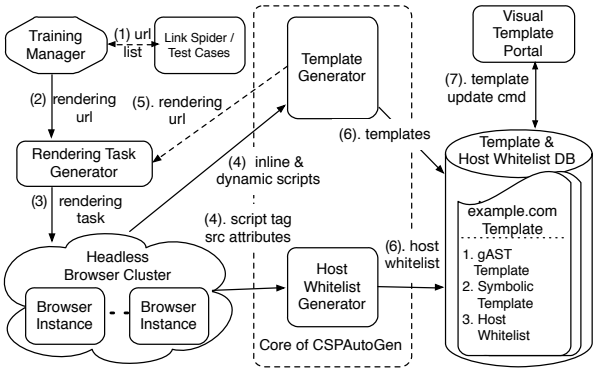

Figure 1: CSPAutoGen Architecture in the Training Phase.

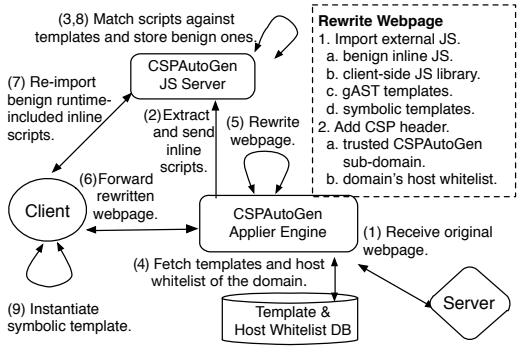

Figure 2: CSPAutoGen Architecture in the Rewriting and Runtime Phase.

Training Phase. Figure 1 illustrates the architecture of *CSPAutoGen* in the training phase. First, a training task is submitted to a *training manager* with a URL list (step 1). The URL list can be obtained by any of the following methods. The administrator specifies the URL list manually; a *link spider* crawls the target domain for the URL list; or the training manager hooks the website test cases and outputs a list of URLs. Then, these URLs are sent to a *rendering task generator* (step 2), which creates webpage rendering tasks by specifying various cookies, such as login cookies, and user-agents to maximize script coverage. These rendering tasks are performed, i.e., webpages are rendered, by a *headless browser cluster* (step 3). During rendering, all the relevant contents, including inline scripts, dynamic scripts, and src attributes of script tags, are sent to a *template generator* and a *host whitelist generator* (step 4). If the template generator needs to render more URLs, these URLs are sent to the rendering task generator (step 5) and steps 3–4 are repeated. The generated templates and host whitelist are stored at a database (step 6), where they can be conveniently updated through a *visual template portal* (step 7).

Rewriting Phase. Figure 2 illustrates the rewriting and runtime phases of *CSPAutoGen*. *CSPAutoGen*'s applier engine is placed between server and client. During the rewriting phase, when receiving a webpage (step 1), *CSPAutoGen* extracts and sends inline scripts to *CSPAutoGen JavaScript server* (step 2), where the scripts are matched against templates and those benign ones are stored in a globally unique subdomain (trusted subdomain). The stored scripts' URLs are returned to *CSPAutoGen*'s applier engine so they could be imported as external scripts (step 3). Then, the applier engine fetches templates and host whitelist of the domain for page rewriting (step 4). Next, the engine rewrites the webpage to import those trusted inline scripts as external scripts, include this domain's templates and a client-side library to be used in the runtime phase, and inject generated CSP policies (step 5). The generated CSP policies work in a strict mode–neither "unsafe-inline" nor "unsafe-eval" is set. Moreover, those CSP policies only allow scripts loaded from the host whitelist and the assigned trusted subdomain. After rewriting, the webpage will be forwarded to user's browser (step 6).

Runtime Phase. In runtime, browser's CSP mechanism guarantees only external scripts from hosts specified in CSP's `script-src` directive can be executed. *CSPAutoGen* uses its client-side library inserted during the rewriting phase to handle both runtime-included inline and dynamic scripts. For the former, once the client-side library detects DOM tree changes and inline scripts inserted, the library will match the scripts with the templates. If matched, those scripts are sent to the website's subdomain at *CSPAutoGen* JavaScript server, where the scripts will be matched again and put into the server if they pass the matching. Next, the URLs associated with these scripts will be returned so that the client-side library can load and execute them at the client side (step 7, 8). For dynamic scripts, the JavaScript library synchronously detects any function calls to `eval` or `eval`-like functions and then matches their parameters against templates. If matched, the library evaluates those scripts by instantiating corresponding symbolic templates (step 9).

3.2 Deployment Models

In this section, we describe three deployment models in detail, and compare the pros and cons of each of them.

Server Deployment. *CSPAutoGen* can be deployed on a server to help developers or site administrators automatically generate and enforce CSPs. Such deployment can protect all the users of the particular website over the Internet, and is applicable to any back-end programming languages, such as PHP, Node.js, and Ruby on Rails. In particular, at the training phase, the developer or administrator runs *CSPAutoGen* upon an internal or testing version of the website to generate templates and corresponding CSPs for the website. Later on, if she adds or changes scripts in the website, she can submit the new scripts to *CSPAutoGen* through a pre-built tool and update the stored templates. The advantage of server deployment is that we can ensure the training set being clean.

Middlebox Deployment. *CSPAutoGen* can be deployed at, for example, the gateway of enterprise networks of companies, schools, and governments. Such deployment will protect all the clients behind the middlebox, such as all the enterprise computers. The enterprise can train the templates themselves or fetch from trusted third-party template providers. Similar to all other training based approaches [11, 16, 44], *CSPAutoGen* requires that the training set is clean and without injected scripts. We understand that it is generally understood to be a hard problem for training based approaches to keep the training set clean [33, 43]. Here is how we avoid XSS attacks in the training set of *CSPAutoGen*. To mitigate known XSS attacks, all crawled webpages are tested via VirusTotal [8], where the scripts will be detected by more than 60 popular anti-virus software. In addition, we crawl webpages from a clean seed URL to avoid reflected and DOM-based XSS attacks. We also create new accounts on websites during crawling to avoid stored XSS attacks, because new accounts are usually initialized with pre-stored clean contents and have no pre-user interactions.

Client Deployment. *CSPAutoGen* can also be deployed at a client, and implemented as either a browser extension or a client proxy. Such deployment needs to fetch templates from a trusted third-party that generates *CSPAutoGen* templates. Similar to middlebox deployment, the training also needs to be performed in a clean environment with no injected scripts.

```
1 CNN.autoPlayVideoExist =
2   (CNN.autoPlayVideoExist === true) ? true:false;
3 var configObj = {
4   thumb: 'none',
5   video: 'politics/...zard-sott-origwx.cnn',
6     ...
7 },autoStartVideo = false,callbackObj,
8 currentVideoCollection = [{
9   "videoCMSUrl": "/nicola...wx.cnn/index.xml",
10  "videoId": "politics/.../ott-origwx.cnn",
11  "videoUrl": "/videos/...uk-election-2015/"
12 }, ... ];
13 configObj.autostart = autoStartVideo;
14 ...
15 if (carousel
16   && currentVideoCollectionContainsId(videoId)){...}
```

Code 2: A snippet of scripts from one of CNN's news webpages (some codes are omitted because of page limits). Values of variables configObj and currentVideoCollection depend on the topic of current webpage so this script is unique and cannot be found in other webpages. However, if we generalize this script by replacing these two variable values with "CSP_Object" and "CSP_Array", the generalized scripts exist in most of CNN's news pages.

4. DESIGN

In this section, we will discuss the design of *CSPAutoGen*. Instead of introducing *CSPAutoGen* in three phases, we will present how *CSPAutoGen* process different categories of scripts—such as inline, dynamic, and runtime-included—because these scripts are the portals of attackers to launch XSS attacks. Now, let us first define these script modifiers:

- *Inline* vs. *External.* The modifier *inline* refers to scripts that are embedded directly as part of HTML, as opposed to *external* scripts that are imported by the `src` attribute of a `script` tag.
- *Dynamic* vs. *Static.* The modifier *dynamic* refers to scripts that are generated during client-side execution through an `eval` or `eval`-like function, as opposed to *static* scripts that are provided by the server directly.
- *Runtime-included* vs. *Pre-included.* We use the modifier *runtime-included* to refer to the scripts that are included during client-side execution, such as invoking `createElement` function to create a `script` tag, as opposed to *pre-included* that are in the HTML before rendering.

Some of the modifiers can be used together to describe scripts. For example, runtime-included inline scripts mean that scripts in a `script` tag are inline, and generated through JavaScript operations. In this paper, *CSPAutoGen* needs to process four categories: pre-included inline scripts, runtime-included inline scripts, external scripts, and dynamic scripts.

In the rest of this section, we first discuss the template mechanism, gAST and its type system, in Section 4.1. Then, in Section 4.2, we present how to apply templates upon each category of scripts, generate corresponding CSPs, and enforce generated CSPs. Last, we will briefly discuss *CSPAutoGen*'s important subsidiary components in Section 4.3.

4.1 Templates Mechanism

Templates are the foundation of *CSPAutoGen* of processing all categories of scripts and generating CSPs. They are created by the *template generator* (Figure 1) in the training phase, stored in the template database, and used to match with inline scripts (by *CSPAutoGen* applier engine in Figure 2) in the rewriting phase and dynamic/runtime-included scripts in the runtime phase. In this subsection, we first discuss the principals of designing templates, and then present how our templates—gAST and its type system—work.

Design Principles. When designing templates for *CSPAutoGen*, we have the following three principles:

- High benign script matching rate. The templates should match benign scripts not present in the training phase. Because many scripts carry runtime information, exact string matching, such as comparing hash values, will not satisfy this principle. For example, Code 2, a snippet of scripts from CNN, shows that the values of `configObj` and `currentVideoCollection` are generated by the server based on the webpage topic. Our templates should be general enough to match these unseen scripts.
- Prevention of injected scripts. The templates should not match injected, malicious scripts. There is a tradeoff between the first and this principle: the templates should be general to match unseen website scripts, but not too general to match injected scripts.
- High performance. Because *CSPAutoGen* needs to enforce CSP in real-time, the templates should be matched with scripts as fast as possible. That is, string-similarity algorithms with high time complexity, such as Levenshtein algorithm, are not suitable here.

Template Overview. Guided by these principles, we propose a novel template based on Abstract Syntax Tree (AST) and a type system. First, as observed in real-world websites, most scripts share exactly the same logic structure but different data values (such as Code 2). Therefore, we propose a generalization of AST, called *generalized AST (gAST)*, to group together such scripts. Compared with AST, the data nodes of gAST are generalized to their type information (e.g., array and string) with no actual values so that they can cover unseen scripts in the training phase.

Second, to prevent script injection, we propose a conservative type inference system, which limits the number of possibly matched data. For example, if a data node only contains a single value in the training phase, *CSPAutoGen* will assign a CONST type to the node; similarly, if the node contains a limited number of values, *CSPAutoGen* will assign an ENUM type to the node. Such conservative inference greatly restricts the ability of an attacker to bypass the templates and inject their own scripts.

Third, the proposed template matching has O(n) time complexity against an incoming script where n is the size of the AST of the script. The overall matching has three steps: generating gAST for the script, matching the generated gAST with templates, and matching the type information in the script against the one in templates. Our evaluation results in Section 6.5 also confirm the efficiency of the matching mechanism.

In the rest of this subsection, we will first show how to build gAST together with the conservative type information, and then discuss how to match scripts against gAST.

Building gAST. The building of gAST is as follows. *CSPAutoGen* first generates the script's standard AST and then traverses the AST to generate gAST. Each gAST node has two attributes: *tag* used for tree comparing, and *value* used for type inference and comparison. Algorithm 1 illustrates how to recursively build gAST from AST.

For complex data node, the gAST node's value attribute is set to the corresponding AST node's so called *non-nested object* value (Line 18), a generalized data structure used for type inference and comparison. Code 3 illustrates an example of converting an array to a non-nested object. Each array or object is converted to a non-nested format composed of a set of key-value pairs, where each key obeys the *key assignment rule* and each value is a list of atomic data values or gAST. The conversion algorithm recursively processes data values from innermost arrays or objects: atomic value is assigned a key directly, while expression is converted to gAST and then assigned a key. Here is the *key assignment rule*:

For array elements, all boolean or number values are assigned with a key as "CSP_boolean" or "CSP_number" respectively; strings are assigned with key of "CSP_string_*level*", where "*level*" refers to the item's nested level. For object, if a value has multiple keys

Algorithm 1 The Algorithm of Building gAST

Input: AST Node: $node$
gASTNode buildGAST(node):
1: $gnode = createGASTNode()$
2: **switch** $(getNodeClassType(node))$
3: **case** $StructureNode(e.g., DoWhile, If)$**:**
4: **case** $OperatorNode(e.g., Assign, BinOp)$**:**
5: $gnode.tag = getNodeClassName(node)(e.g., Assign)$
6: $gnode.value = null$
7: break
8: **case** $IdentifierNode(i.e., Identifier)$**:**
9: $gnode.tag = getNodeValue(node)(e.g., CNN, configObj)$
10: $gnode.value = null$
11: return $gnode$
12: **case** $AtomicDataNode(e.g., Number, String)$**:**
13: $gnode.tag = getNodeClassName(node)(e.g., String)$
14: $gnode.value = getNodeValue(node)(e.g., "...gwx.cnn")$
15: return $gnode$
16: **case** $ComplexDataNode(i.e., Array, Object)$**:**
17: $gnode.tag = getNodeClassName(node)(e.g., Array)$
18: $gnode.value = getNonNestedObject(node)$
19: return $gnode$
20: **end switch**
21: **for** $child$ in $node.children$ **do**
22: $gnode.appendChild(buildGAST(child))$
23: **end for**
24: return $gnode$

```
var org_array =
['str1', [4,'str2'], {k1:'str3', k2:{k3:'str4'}},
{k1:'str5'}, 2, 3, fun()+1];
//org_array's non_nested_obj
var non_nested_obj =
{CSP_expression:[gAST], CSP_string_lev1:['str1'],
 CSP_string_lev2:['str2'], CSP_number:[2,3,4],
 k1:['str3','str5'], k3:['str4'] };
```

Code 3: An example of converting an array variable org_array to non-nested object variable non_nested_obj.

(nested array/object), keep the closet one that are not generated by *CSPAutoGen* (i.e., starting with "CSP_"); otherwise, keep its original key. All expressions in array and object will be assigned with key of *CSP_expression*.

Type System. As mentioned, *CSPAutoGen* infers the type information of each data node (i.e., array, boolean, number, object and string type) in gAST. In particular, all scripts with the same gAST are grouped together and corresponding nodes' values are put together as training samples. Based on these samples, *CSPAutoGen* infers type information for each generalized data node. For an atomic data node (i.e., boolean, number and string), each node is assigned a single type; for a complex data node (i.e., array and object), each key-value pair of the non-nested object has an independent type.

To accurately and conservatively infer type information, *CSPAutoGen* requires that the number of samples for each inferred type is larger than a threshold (10 by default in our implementation). If this condition does not meet, *CSPAutoGen* will send URLs of the webpages containing relevant scripts to the *rendering task generator* (Figure 1), which then starts rendering tasks with various client-side configurations to obtain more samples.

In our type system, we define the following six types:

- CONST. If all the samples are with the same value, *CSPAutoGen* infers a CONST type. When matched with a CONST type, the target value should be exactly the same as the const value.
- ENUM. If the sample set size is larger than a threshold (120 by default), and the number of different values is significantly smaller (5 or less than 5 by default), *CSPAutoGen* infers an ENUM type. When matched with an ENUM type, the target value should be an element of the type's value set.
- NUMBER. If all the samples can be parsed as numbers, *CSPAutoGen* infers a NUMBER type. When matched with a NUMBER type, the target value should be a number.
- URL. If all the samples can be parsed as URLs, *CSPAutoGen* infers a URL type, which contains a subtype—domain name—with a set of domains appearing in the samples. When matched with a URL type, the target value should be parsed as a URL and the domain in the URL should match the domain name as an ENUM type. Note that a URL's parameters might contain another URL for redirection and that URL also needs to be checked.
- GAST. The value of an array or object can be an expression. In the step of converting array and object to non-nested objects, expressions are extracted and put under key of *CSP_expression*. *CSPAutoGen* infers a GAST type when the key in a non-nested object is *CSP_expression*. When matched with a GAST type, the target value should be an expression and the gAST of the target value will be matched with the gASTs stored in the type.
- REGEXP. If *CSPAutoGen* cannot infer any of the aforementioned five types, *CSPAutoGen* will infer a REGEXP type based on different attributes. At the same time, *CSPAutoGen* also allows specifying a regular expression manually. Towards this end, we have identified eight possible attributes of generating a regular expression: (1) length (min and max), (2) character (is_alphabetic, is_numeric or special character set), and (3) common strings (a prefix, an appendix, or a domain name in the middle).

Note that GAST is a complex type whose restrictiveness depends on the rest five basic types. Among these five basic types, except the REGEXP type, all the rest four types are so conservative that the number of possible matched data in real time is very small and most of the matched data is seen in the training phase. Only the REGEXP type is potentially dangerous if it contains characters other than letter or number because an attacker may have more flexibility to inject suspicious contents. However, though theoretically possible, we believe that such attack venue is impractical, because it requires the collaboration of website developers to implement a functionality wanted by the attacker. Moreover, because the number of such flexible types is small, one could utilize our *visual template portal* to manually review regular expressions generated by *CSPAutoGen* to ensure security. A detailed evaluation of the amount of manual work could be found in Section 6.4.

Template Matching. There are two steps in matching an incoming script (i.e., the target script) with the templates of its domain: gAST matching and type matching. First, *CSPAutoGen* first generates target script's gAST and then compares the gAST with the ones in the templates. When comparing two gASTs, *CSPAutoGen* only compares each node's `tag` attribute because `tag` is used to describe the script's structure. To further speed up the matching, *CSPAutoGen* pre-stores a string consisted of a sequence of `tags`—which are the traversing results of each gAST in the templates—and the string's hash value (called gAST hash). When matching the target script, *CSPAutoGen* only compares the gAST hash of the target script and the ones of the templates.

Second, if the gAST matching succeeds, *CSPAutoGen* extracts the template's type set and compares all the data nodes of the target script with their corresponding types in the templates. For each data node, the matching rule depends on the data node type:

- Atomic data node. Directly match the value extracted from target script against the corresponding type associated with the node.
- Complex data node. To match a complex data node, *CSPAutoGen* first converts the value of the target script, either an array or an object, to a non-nested object, and then compares it against the corresponding object in the templates. Specifically, for each key-value pair in the target non-nested object, *CSPAutoGen* first

checks whether all the keys can be found in templates. If not, the matching fails; otherwise, for each value in the target non-nested object, *CSPAutoGen* compares it against the corresponding type specified in templates. *CSPAutoGen* will determine the complex node a match only when all the value match.

In sum, a script matches templates only when (1) the gAST matches, and (2) the types of all the data nodes match.

4.2 Processing Scripts based on Templates

In this section, we will discuss in details how *CSPAutoGen* processes four categories of scripts, i.e., pre-included inline scripts, runtime-included inline scripts, external scripts (both pre- and runtime-included) and dynamic scripts, to ensure only these matching templates can be executed. *CSPAutoGen* stores pre- and runtime-included inline scripts at its JavaScript server (Figure 2) and specifies the server in CSP's `script-src` directive. For inline scripts from different domains, the sever provides corresponding globally unique subdomains, i.e., *hashValue.cspautogen.com*, where $hashValue$ is the hash value of the domain that inline scripts come from. We refer to this trusted subdomain as the domain's `CSP trusted host`.

Pre-included Inline Scripts. Pre-included inline scripts—which are processed during rewriting phase—can be further divided into three sub-categories:

- These embedded in script nodes (e.g., <*script*>*alert(1);*</*script*>). *CSPAutoGen* extracts scripts and sends them to *CSPAutoGen* JavaScript server, where the scripts are matched against templates. If match, the scripts are stored at the server with a unique URL from the domain's `CSP trusted host`. Then, *CSPAutoGen* rewrites the `script` tag by removing inline contents and adding a `src` attribute pointing to the URL.
- Inline event handlers embedded in tag attributes (e.g., <*div oncli ck="alert(1);"*></*div*>). The process is similar to these embedded in script nodes. The difference is that in addition to the original inline scripts, *CSPAutoGen* adds a function wrapping the handler, and further adds this function through *addEventListener* API when *onDOMContentLoaded* event is fired.
- JavaScript URL scheme (e.g., <*a href = "javascript: alert(1);"* ></*a*>). Again, other steps are similar to these embedded in script nodes. The difference is that *CSPAutoGen* adds a function wrapping the original scripts in the JavaScript URL scheme, and registers an *onClick* event for the original tag.

Dynamic Scripts (Eval/Eval-like Function Execution). To prevent users from DOM-based XSS, *CSPAutoGen* does not set the keyword "unsafe-eval" in CSP header. That is, `Function` and `eval` cannot be called to evaluate a string as JavaScript code, and functions `setTimeout` and `setInterval` can only be called when the first argument is a callable [1]. We refer to these four functions as `eval` and `eval`-like functions. Though unsafe, these functions are commonly used in modern websites, and it incurs serious compatibility issues if directly disabling them.

To execute dynamic scripts, we need to answer two questions: (1) which strings in `eval` or `eval`-like functions are allowed to execute, and (2) how to execute them without using `eval` or `eval`-like functions. The answer for the first question is simple—parsing the string into a gAST, and matching the gAST and values against templates. The details have already been discussed in the previous subsection. Then, let us answer the second question. To execute such strings, we propose a *symbolic template* mechanism to synchronously execute these allowed strings as JavaScript codes when `eval` or `eval`-like functions are disabled.

A symbolic template is a function generated from corresponding gAST by converting the data nodes in gAST to symbolic variables. Specifically, to generate a symbolic template, *CSPAutoGen* first

```
1  //Target Script
2  eval("AMPTManager.pageSlotsObj['ad_ns_btf_01']
3   = googletag.defineSlot('/866347/CNN/world/leaf',
4    [[1,2],[150,90],[300,50],[300,100]],
5     'ad_ns_btf_01').
6      addService(googletag.pubads()).
7       setTargeting('pos',['ns_btf_01']);");
8  //gAST for the Target Script
9  AMPTManager.pageSlotsObj[CSP_String]
10  = googletag.defineSlot(CSP_String,
11   CSP_Array, CSP_String).
12     addService(googletag.pubads()).
13      setTargeting(CSP_String, CSP_Array);
14 //Symbolic Template for the gAST
15 symTemplates[hash] = function(
16  CSP_S1,CSP_S2,CSP_A3,CSP_S4,CSP_S5,CSP_A6){
17   AMPTManager.pageSlotsObj[CSP_S1] =
18     googletag.defineSlot(CSP_S2,
19       resolveASTNodeVal(CSP_A3,hash),CSP_S4).
20         addService(googletag.pubads()).
21           setTargeting(CSP_S5,
22           resolveASTNodeVal(CSP_A6,hash));};
```

Code 4: Example of dynamic script. During runtime, when the target script is called (Line 2–7), our client-side JavaScript library will capture its parameter and match it against the templates. If match, the library will instantiate and call the corresponding symbolic template (Line 15–22), which is a JavaScript function corresponding to a gAST (Line 9–13). In the symbolic template, function resolveASTNodeVal(...) will resolve each complex data node (i.e., array and object) with the corresponding node's AST subtree during execution.

creates a function with all parameters corresponding to each generalized data node. Then it substitutes all the atomic data nodes in gAST with symbolic variables, and all complex data nodes with the results of runtime instantiation function ($resolveASTNodeVal$) calls. After that, it converts the gAST back to a script with symbols and sets it as the function's body. Code 4 shows an example of the symbolic template (Line 15–22) and corresponding gAST (Line 9–13) for the example at Line 2–7. During training phase, *CSPAutoGen* generates one symbolic template for each gAST, ships them with all other templates to the client. In runtime phase, these symbolic variables will be instantiated via parameters generated from the arguments of the `eval` or `eval`-like functions.

Next, we will discuss how to instantiate and execute a symbolic template. In the rewriting phase, *CSPAutoGen* rewrites an incoming webpage by inserting *CSPAutoGen* client-side library as a trusted external script in the beginning of the rewritten webpage.

Then, in the runtime phase, the client-side library overwrites the original `eval` or `eval`-like functions defined by the browser, i.e., `eval` or `eval`-like functions are redefined as a normal function in *CSPAutoGen* . Such overwritten `eval` or `eval`-like functions are allowed by CSP. These overwritten functions serve as three purposes: (1) checking whether the string to execute matches one template, (2) extracting data node values from the argument string, and (3) instantiating the symbolic template. Code 5 shows the pseudocode of overwritten `eval` function, which we will use as example to discuss. All the other `eval`-like functions can be handled in a similar way.

First, in the beginning of the overwritten `eval` function, the argument string will be matched against the domain's templates. If match, the overwritten function finds the corresponding symbolic template and continues the following two steps; otherwise, the string will not be executed (Line 3-8).

Second, the overwritten `eval` function parses the string into an AST, from which it will extract data nodes and store them into an array as arguments for symbolic template (Line 10). If the data node is atomic type (i.e., boolean, number and string), the actual value of each data node (e.g., 200 and "a string") is extracted; if the

```
1  window.eval = function(scriptStr){
2   //match target string against templates.
3   var template = findTemplate(scriptStr);
4   if (!template) return ;
5   //if match, find the symbolic template.
6   var hash = getTemplateHash(template);
7   var symTemplate = symTemplates[hash];
8   if (!symTemplate) return ;
9   //extract args & run symbolic template.
10  var args = extractArgs(genAST(scriptStr));
11  symTemplate.apply(this, args); };
```

Code 5: Pseudocode of overwritten `eval` *function. In the overwritten* `eval` *function, the function first matches the target script against templates (Line 3–4). Only when a match is found, will the overwritten* `eval` *function extract the arguments (i.e., the data nodes) from the target script's AST node (Line 10) and then instantiate the corresponding symbolic templates with the arguments (Line 11). Symbolic templates are determined based on their gAST's tree structure hash (Line 6-8).*

data node is complex type (i.e., array and object), the data node's subtree will be extracted and stored, because the value should be resolved during instantiating the symbolic template to preserve the order of expressions that might appear on complex data node. After that, the extracted arguments array will be fed into symbolic template to evaluate the string (Line 11).

Third, in symbolic template, all the array and object node arguments will be "wrapped" by function $resolveASTNodeVal$, which accepts AST data node and parses it into JavaScript object (shown in Algorithm 2). In $resolveASTNodeVal$, for a boolean, number or string node, *CSPAutoGen* creates a new Boolean, Number or String object (Line 2-4 of Algorithm 2). For an array (Line 5–10 of Algorithm 2) or object node (Line 11–17 of Algorithm 2), *CSPAutoGen* iterates each member in the array or object and recursively resolves each member. For an Expression node, *CSPAutoGen* finds the symbolic template corresponding to the expression and then invokes the symbolic template with extracted arguments. Note that such template must exist if the script passes template matching, and these expression symbolic templates are defined in their parent symbolic templates (i.e., the function body where $resolveASTNodeVal$ is called) to preserve scope chain. (Line 20–23 of Algorithm 2). Identifier node is handled similarly: for each variable that might appear in complex nodes of a template, a one-line function is defined in the corresponding symbolic template to return the variable's value, and $resolveASTNodeVal$ will call that function to resolve identifier node value (Line 18–19).

To preserve the scripts' behavior, several design details need to be further discussed.

- Scope Chain. JavaScript functions are running in the scope chain where the functions are defined [15]. In order to preserve original scripts' scope chain, we define overwritten `eval` and `eval`-like functions on the global `window` object, where their original counterparts are defined. The only exception is `eval`, because `eval` can be used as both a function and a keyword. If `eval` is used as a function (e.g., *window.eval(...)* or *var e = eval; e(...)*), our overwritten `eval` still preserves its scope chain (i.e., running on global `window` object). Otherwise, if `eval` is used as a keyword (i.e., *eval(...)*), the argument string of `eval` is running in the current scope chain. To preserve the scope chain for the keyword `eval`, during symbolic template generation, *CSPAutoGen* searches the keyword `eval`, rewrites the script by inserting the symbolic template corresponding to the `eval` argument before the keyword `eval`, and then changes the keyword `eval` invocation to the symbolic template function call. Note that if `eval` is used as a keyword, it cannot be obfuscated, which means that we can find all of such usages.

Algorithm 2 The Algorithm of Instantiating Symbolic Node

Input: AST Node: $node$
Input: String: $callerHash$
JavaScriptObject resolveASTNodeVal(node, callerHash):
1: **switch** $(getNodeClassName(node))$
2: **case** $Boolean$**:** return $Boolean(getNodeValue(node))$
3: **case** $Number$**:** return $Number(getNodeValue(node))$
4: **case** $String$**:** return $String(getNodeValue(node))$
5: **case** $Array$**:**
6: $ret = new\ Array()$
7: **for** $item$ in $extractVals(node)$ **do**
8: $ret.push(resolveASTNodeVal(item, callerHash))$
9: **end for**
10: return ret
11: **case** $Object$**:**
12: $ret = new\ Object()$
13: $keys, vals = extractKeysVals(node)$
14: **for** i in $keys$ **do**
15: $ret[keys[i]] = resolveASTNodeVal(vals[i], callerHash)$
16: **end for**
17: return ret
18: **case** $Identifier$**:**
19: return $varResolveMethods[callerHash][node.name]()$
20: **default:** $(i.e., Expression)$
21: $hash = getTemplateHash(getNodeTemplate(node))$
22: $args = extractArgs(node)$
23: return $exprSymTemp[callerHash][hash].apply(this, args)$
24: **end switch**

- Argument Instantiation Order. For array and object data nodes, their elements might contain expressions and identifiers, whose values are not definitive until being resolved. Therefore, the argument instantiation order matters. *CSPAutoGen* adopts runtime instantiation rather than pre-instantiation for array and object data nodes, ensuring the instantiation sequence not changed.

Runtime-included Inline Script. Runtime-included inline scripts are generated from other scripts at client side. Those scripts are composed of two categories: asynchronous and synchronous.

Asynchronous runtime-included inline scripts can be imported in three ways: (1) adding/changing a script tag with inline script, (2) adding/changing an inline event handler, and (3) adding/changing an attribute with JavaScript URL scheme.

To allow such inline scripts, during page rewriting, the client-side JavaScript library of *CSPAutoGen* monitors DOM tree changes by a *MutationObserver* instance and then processes runtime-included scripts. Let us use the creation of a `script` tag with inline scripts in runtime as an example. If the JavaScript library detects added or modified inline scripts, the library tries to match them against the templates of the domain. If no match, the library does nothing as CSP has already disabled the scripts; if match, the library sends these scripts to the *CSPAutoGen* JavaScript server. At the server, *CSPAutoGen* will match the scripts again. If match, the server stores the scripts, and returns corresponding URLs (part of the domain's *CSP trusted host*) to the library. We check scripts at both client and server sides for both efficiency and security: only matched scripts will be sent to server by our client-side library and the server can detect and reject those malicious scripts sent by attackers to bypass our system. After receiving a script's URL, *CSPAutoGen* changes the script tag by setting the src attribute as the received URL. The whole process is working asynchronously to minimize the overhead. For the other two categories, i.e., inline event handlers and inline JavaScript URL scheme, the entire process only differs after *CSPAutoGen* receives the script's URL from the *CSPAutoGen* JavaScript server. The details will be the same as processing pre-included inline event handlers, thus being skipped.

Synchronous, runtime-included, inline scripts are executed immediately, such as these loaded through the *document.write* API. For such scripts, *CSPAutoGen* parses the HTML code, which is

the first parameter of *document.write*, through an innerHTML attribute, extracts scripts from script tags, matches the script with gAST, and executes the script through symbolic templates. The matching and execution are exactly the same as *eval* or *eval*-like functions, and will be skipped here.

External Scripts. External scripts are imported through the `src` attribute of `script` tag (e.g., <*script src="..*"><*/script*>). *CSPAutoGen* handles these scripts by specifying the allowed hosts in CSP's `script-src` directive. These allowed hosts, generated by the *host whitelist generator* in the training phase, are part of the templates. During the rewriting phase, *CSPAutoGen* fetches the allowed script host list from the template database and adds them to the CSP. Note that as specified by CSP, external scripts can be imported during runtime, and runtime-included external scripts have to be part of the allowed host list. Therefore, *CSPAutoGen*'s client-side JavaScript library does not need to take additional actions for runtime-included external scripts.

4.3 Subsidiary Components

We briefly introduce three *CSPAutoGen*'s subsidiary components here. They are not indispensable, but can either improve template robustness or alleviate deployment burden.

Visual Template Portal. Although *CSPAutoGen* can generate templates without any human interventions, we have designed an interface, *visual template portal*, for template providers or website developers to generate templates with higher robustness and accuracy. Specifically, one—the template provider or website developer—can modify gASTs, the associated type information and template patches, discussed in the following, for any domain. To facilitate this process, *visual template portal* will show each template as well as those associated training scripts visually so user can improve templates without fully understanding each script.

Violation Script Panel. For non-server deployment, *CSPAutoGen* provides a *violation script panel* to end users so they can whitelist their trusted scripts. The panel is essentially a popup window that can be opened by an optional button inserted via *CSPAutoGen*'s client-side library. Such panel lists all the blocked scripts of current domain, specifies reason for each violation (i.e., *gAST does not exist* and *type does not match*) and related information. End user can whitelist any of those scripts or add new trusted scripts. Each client has her own whitelist, thus, *CSPAutoGen* needs to authenticate clients. The authentication is via a unique identifier cookie set in the client browser.

Violation Report Mechanism. Violation reports can help administrators understand if there are attacks happening or whether the deployed templates are outdated. CSP's native violation report mechanism is not applicable to *CSPAutoGen*, because violation cases might not trigger CSP violations (e.g., those non-matched scripts in rewriting phase) and scripts that trigger CSP native violations might get executed (e.g., the runtime-included inline scripts that match templates). Therefore, we propose a new violation report mechanism that will send accurate violation report to administrators. Those reports include violated URLs and scripts, under user agreements.

When a website's JavaScripts are updated, *CSPAutoGen* will capture the violated scripts via violation report mechanism and group them based on their gASTs. If one group's script number achieves a threshold, *CSPAutoGen* can automatically generate template patches and report the patches to the administrator. Then, the administrator can utilize visual template portal to visually review these patches as well as their corresponding templates, and securely apply the patches to existing templates. Such process is not often: our evaluation shows that people may only need to do it every other month (Section 6.2.2).

5. IMPLEMENTATION

We implement *CSPAutoGen* in 3,000 lines of Python code and 6,500 lines of JavaScript code. In the training phase, we customize PhantomJS as a headless browser to render URLs and extract scripts as well as trusted hosts. These headless browsers are driven and managed by our rendering task generator, which is written in Python to dynamically configure PhantomJS instances. The other two components, the template generator and the host whitelist generator, are written in Node.js. We use Esprima library [3], a high performance, standard-compliant JavaScript parser written in JavaScript, to parse codes and generate ASTs. The generated gASTs will be stored in our template database implemented by MongoDB.

In the rewriting phase, *CSPAutoGen* is deployed on mitmproxy, an SSL-capable man-in-the-middle proxy written in Python. It provides a scripting API to external Python scripts, defined as inline scripts (which is different from inline scripts in JavaScript). Such inline scripts can intercept HTTP/HTTPS requests and responses as well as modify them on the fly. Our *CSPAutoGen* engine is implemented as mitmproxy's inline scripts, and utilizes Python's BeautifulSoup package to parse and rewrite DOM trees. In order to be robust to the broken pages that can still be rendered by browser, we choose html5lib parser [7] as parsing engine, which is lenient and parses HTML the same way a web browser does.

In the runtime phase, the client-side JavaScript library handles runtime-included and dynamic scripts. The library is written in JavaScript, and uses Esprima library to generate script AST.

All the servers, including the *CSPAutoGen* JavaScript server, the violation report server and the whitelist server, are implemented in Node.js. At server side, the scripts need to be matched against templates. Those codes are written in Node.js with Esprima library.

6. EVALUATION

In this section, we evaluate *CSPAutoGen* to answer these six questions: (1) How do we train *CSPAutoGen*? (2) Are the generated templates robust enough to match most of the unseen but benign scripts, and stable enough to last for a relatively long period? (3) Are the generated templates correct? (4) Can *CSPAutoGen* protect vulnerable websites against real-world XSS attacks? (5) How much overhead does *CSPAutoGen* incur? (6) Is *CSPAutoGen* compatible with real-world websites?

6.1 Training Datasets

In this section, we present how to train *CSPAutoGen* based on different scenarios, i.e., whether we deploy *CSPAutoGen* at the middlebox or the server.

6.1.1 Training based on Alexa Websites

To evaluate *CSPAutoGen* over real-world websites, we obtain Alexa Top 50 websites as our dataset. For each website, we crawl 2,500 different webpages and split them into a training set with 2,000 webpages (training webpages) and a testing set with the rest 500 webpages (testing webpages). To maximize the code coverage, we use PhantomJS [34], a popular headless browser, to render each webpage ten times with five user-agents (i.e., Android, Chrome, Firefox, IE and iPhone) and two cookie settings (i.e., a clean cookie jar and a cookie jar initiated with login credential) respectively. Moreover, to make sure that the training set is clean, the crawling starts from a clean URL, and all crawled scripts are tested by VirusTotal [8] under more than 60 antivirus software preventing known XSS payload.

Table 1: Template Robustness over Time for Alexa Websites.

Domain	# of gAST (2016/01/01)	# of Scripts (2016/02/01)	Matching Rate (2016/02/01)	# of Scripts (2016/03/01)	Matching Rate (2016/03/01)	# of Scripts (2016/04/01)	Matching Rate (2016/04/01)
Amazon	596	149,592	99.62%	157,281	99.13%	373,635	98.39%
CNN	259	46,939	98.72%	45,559	98.50%	48,853	98.30%
Facebook	53	17,830	99.47%	15,419	97.90%	10,995	95.98%
Google	857	12,082	97.89%	14,501	94.13%	16,385	87.14%
Reddit	49	14,487	97.47%	12,888	97.01%	13,622	89.78%
Yahoo	295	8,834	99.30%	8,676	99.05%	7,264	94.01%

Table 2: Template Robustness over Time for Web Frameworks.

Application	First Version (V1) & Release Date	# of gAST from V1	Second Version (V2) & Release Date	# of gAST from V2	Third Version (V3) & Release Date	V3 LOC	V1 Matching Rate	V2 Matching Rate
Concrete5	5.7.0 (09/12/2014)	3	5.7.4 (04/17/2015)	8	5.7.5 (08/11/2015)	92,211	67.7%	85.6%
Drupal	7.2.2 (04/03/2013)	5	7.3.2 (10/15/2014)	5	7.4.3 (02/24/2016)	53,000	100%	100%
Joomla	3.4.0 (02/24/2015)	5	3.4.2 (06/30/2015)	5	3.4.3 (07/02/2015)	447,763	100%	100%
MyBB	1.8.0 09/01/2014)	25	1.8.4 (02/15/2015)	26	1.8.5 (05/27/2015)	329,633	99.6%	100%
SilverStripe	3.1.0 (10/01/2013)	0	3.1.12 (03/09/2015)	6	3.1.13 (05/27/2015)	297,787	NA	100%
WordPress	4.2.0 (04/22/2015)	3	4.2.3 (07/23/2015)	26	4.2.4 (08/04/2015)	262,348	0%	100%

6.1.2 Training based on Web Frameworks

When *CSPAutoGen* is deployed at the server, web developers can generate a customized training URL list that has better coverage than crawling. This is more like a gray-box deployment sitting in between blackbox and whitebox. The key idea is that most websites have test cases that cover the source code and explore the website functionality as much as possible. Therefore, we can utilize these test cases to generate a list of URLs, which can be used to explore the scripts on that website.

Specifically, because Alexa websites are closed source, we train *CSPAutoGen* based on open-source web frameworks. In the experiment, we use six frameworks, which are Concrete5, Drupal, Joomla, MyBB, SilverStripe and WordPress.

Here are the details about generating the URL list. Five out of the six frameworks provide extensive test cases. By modifying the testing framework to hook the HTTP request sending methods (e.g., *WP_UnitTestCase.go_to(...)* and *WebDriver.get(...)*), we get a list of testing URLs with various parameters covering all the interfaces of the website. Then we use different credential settings (i.e., admin credential, regular user credential and no credential) to crawl these URLs. For the remaining one web framework without test cases (i.e., MyBB), we first crawl a list of URLs using scrapy [5] framework. Then we use its Google SEO plugin to generate sitemap automatically and manually supplement URL list with the help of sitemap as well as source codes. After that, we crawl those URLs with the above three credential settings.

6.2 Template Robustness

In this section, we evaluate our template's robustness on unseen webpages and over time.

6.2.1 Robustness on Unseen Webpages

In this experiment, we evaluate the template robustness on unseen webpages. For Alexa websites, we use 500 unseen webpages that are crawled but not included in the training. For web frameworks, we deploy the web framework and crawl 500 new webpages. In our evaluation of Alexa websites, the average number of training scripts for each domain is 39,079 and the average number of testing scripts is 12,087 per domain. The median value of the number of gASTs for each domain is 262.

Figure 3 shows the matching rate of *CSPAutoGen*'s templates for Alexa Top 50 websites: ranging from 91.6% to 100.0%, with a median value as 99.2%. Such a high matching rate shows that *CSPAutoGen* is able to match most of the unseen scripts, and this is also confirmed in our compatibility evaluation later. In web framework evaluation, because of the training method in which we know the source code, the matching rate is 100% for all six web frameworks.

6.2.2 Robustness over Time

In this subsection, we conduct two experiments on real-world websites and web frameworks respectively to evaluate how long a template can achieve a high matching rate without updating.

Robustness on Real-world Websites over Time. In the first experiment, we randomly select six popular websites and show the template matching rate over three months. Specifically, we train templates based on the contents of each website on 01/01/2016. Then we use these templates to match the same website captured on 02/01/2016, 03/01/2016 and 04/01/2016. In this experiment, 2,000 crawled webpages are used for both training and testing.

The results are shown in Table 1. We set matching rate 90% as the threshold for template updating. From Table 2, we can see that the templates of Amazon, CNN, Facebook and Yahoo can work without updating for at least three months; the templates of Google and Reddit can maintain satisfying matching rates for two months. *CSPAutoGen* can automatically generate template patches. For security, it is highly suggested that site administrators would review the patches before applying them. We have interviewed engineers from 8 big IT companies, including Google, Facebook and Amazon: all of them considered the workload of reviewing templates every other month acceptable.

Robustness on Web Frameworks over Time. To evaluate template robustness of open-source web framework, for each of the aforementioned six in Section 6.1.2, we deploy three popular release versions, referred to as the first version (V1), the second version (V2) and the third version (V3). In this experiment, we generate two templates based on V1 and V2, and then use these two templates to match *all* the inline scripts extracted from V3.

Table 2 shows the results. Each row contains the versions and release dates of V1, V2 and V3, the number of gASTs in each template, the lines of codes for V3 and the matching rates of the two templates. The results show that using V2 template to match V3 scripts can achieve an acceptable matching rate (100% except for Concrete5, which is 85.6%). The median value of the duration between the release dates of V2 and V3 is 90 days, meaning that on average, the template can work for three months without requiring updating. As for the matching rates of V1 templates, Drupal, MyBB and Joomla can still achieve a very high rate and the duration median value is 268 days. However, SilverStripe, Concrete5 and WordPress's V1 templates need to be updated.

6.3 Correctness

In this part of the section, we evaluate the correctness of generated templates. Because Top Alexa websites are closed source, we can only evaluate web frameworks. Specifically, due to extensive

Table 3: Comparison of Types in WordPress 4.2.3 with These Inferred by CSPAutoGen

Type	# in Ground Truth	# in Templates	# of Correctly Inferred	Accuracy
CONST	95	97	95	97.9%
ENUM	9	3	3	100%
GAST	0	0	0	N/A
NUMBER	24	24	24	100%
REGEXP	10	14	10	71.4%
URL	9	9	9	100%
Total	147	147	141	95.9%

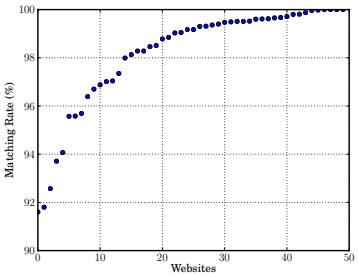

Figure 3: Template Matching Rate (Medium Rate 99.2%).

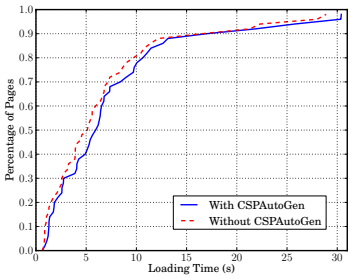

Figure 4: The Loading Time of Alexa Top 50 Websites (Median Difference 9.1%).

Table 4: Real-world Applications with All Three Types of XSS Vulnerabilities (Reflected, Stored and DOM-based)

Application	Version	Vulnerability	Language	LOC
Codiad	2.4.3	CVE-2014-9582	PHP	8,989
Ektron CMS	9.1.0	CVE-2015-4427	ASP.NET	NA
FoeCMS	0.0.1	CVE-2014-4849	PHP	17,943
JForum	2.1.9	CVE-2012-5337	JAVA	61,247
LiteCart	1.1.2.1	CVE-2014-7183	PHP	29,175
OrchardCMS	1.9.0	CVE-2015-5520	ASP.NET	109,467

human works involved in the evaluation, we use WordPress 4.2.3 as an example for the evaluation.

Our methodology is as follows. For each inline script in the WordPress source code, we find the corresponding template, and manually compare each PHP part (i.e., runtime information) in the inline script with each type in the template. For example, if the type in the template is a URL, and the corresponding PHP variable value of the inline script in the source code can only be URL, we will consider the type in the template correctly inferred.

The evaluation results are shown in Table 3. The number of each type in the WordPress source code, and the number of each type inferred by *CSPAutoGen* in the templates are listed in the second and third column respectively. We also list the number of correctly inferred for each type and the corresponding accuracy. First, note that the total numbers for all the types in both WordPress and templates are exactly the same, indicating that *CSPAutoGen* correctly finds all the possible locations. Second, the overall inference accuracy is 95.9%, a very high number. In some cases, the inferred types by *CSPAutoGen* are looser or stricter than the one in WordPress. Particularly, two CONST types should be ENUM, and four REGEXP types should be ENUM. The stricter inference (ENUM as CONST) is because of missing values not captured in the training samples; correspondingly, the looser inference (ENUM as REGEXP) is due to the fact that the number of samples with those scripts is smaller than our default threshold (i.e., 120). In this experiment, the deployed WordPress contains only a few default pages, i.e., in real-world websites, we expect the accuracy will be even higher, because the number of samples should reach the threshold, and the number of training samples will be also larger.

6.4 Security

We evaluate the security of *CSPAutoGen* in three experiments. First, we measure whether and how *CSPAutoGen* can successfully protect real-world vulnerable applications against existing XSS attacks. Second, as discussed in our type system, we manually review the flexible types, i.e., REGEXP types with sensitive characters such as '%', '.' and ':'. Third, from Alexa Top 50 websites, we find those with CSP deployed and compare their configured CSPs with the CSPs generated by *CSPAutoGen*.

Real-world Applications. To evaluate the security of our system, we apply *CSPAutoGen* on six real-world web applications (written in ASP.NET, Java and PHP) with XSS vulnerabilities listed in Table 4. Codiad is a lightweight and interactive web IDE; FoeCMS is a content management application that is largely used in the Spanish world; Litecart is a free development platform to build e-commerce websites. These three applications are all written in PHP. Orchard CMS and Ektron CMS are both content management system written in ASP.NET. The former is open source, while the latter is not. JForum is a lightweight discussion board system implemented in Java. The involved vulnerabilities of these six applications cover all the three types of XSS, that is, reflected, stored and DOM-based XSS attacks. In the table, we list the applications' names, versions, vulnerabilities, languages and lines of codes.

We deploy the six applications and initiate XSS attacks against them. The attacking payloads are created by XCampo [46], a popular XSS payload generator. We first verify that these exploits work on the applications. Then we deploy *CSPAutoGen* at the entrance of each application and initiate the same attacks again. The evaluation results indicate none of these attacks succeed, showing that *CSPAutoGen* defeats against all the three types of XSS attacks.

Manual Reviewing Flexible Types. We count the number of gASTs, nodes, atom data nodes, complex data nodes, types and flexible types from the templates of Alexa Top 50 websites. The results are shown in Table 5: flexible types only account for 1.4% of all the types. Further analysis shows that on average, four gASTs have one data node or field assigned with flexible type; the number of flexible type for each domain template ranges from 1 (e.g., *ask.com*) to 243 (i.e., *amazon.com*) with the median of 81 (i.e., *baidu.com*).

We then evaluate the workload of manual reviewing flexible types. We randomly pick out five websites (*aliexpress.com*, *reddit.com*, *taobao.com*, *weibo.com* and *youtube.com*). The numbers of their flexible types are 121, 6, 59, 27 and 90 respectively. With the help of *visual template portal* (Section 4.3), one student reviews and modifies these flexible types in 2 days (16 hours). Our reviewing shows that no flexible type in these templates needs to be changed to more restrictive ones. Also, fully understanding those reviewed scripts are not required because the *visual template portal* lists all the training samples and highlights the corresponding values.

Comparison with existing websites' CSPs. Among Alexa Top 50 websites, six websites (*facebook.com*, *twitter.com*, *yandex.ru*, *mail.ru*, *pinterest.com* and *alibaba.com*) have configured CSPs. However, five out of the six set both keywords "unsafe-inline" and "unsafe-eval" in their CSPs, and the remaining one (*twitter.com*) sets "unsafe-eval". "unsafe-inline" still allows an attacker to inject scripts via both stored and reflected XSS, and "unsafe-eval" allows DOM-based XSS attacks. As a comparison, the CSPs generated by *CSPAutoGen* set neither "unsafe-inline" nor "unsafe-eval", being more secure than CSPs used in any of the six websites. That is, *CSPAutoGen* can even help existing websites that partially adopt CSP and enhance their security.

Table 5: Templates Statistics of the Alexa Top 50 Websites.

gAST Number	Total Node	Atom Data Node	Complex Data Node	Type	Flexible Type
20,888	1,922,147	225,807	57,391	322,194	4,580

Table 6: The Breakdown of CSPAutoGen Latency.

Latency Source	Min (ms)	Median (ms)	Max (ms)
DOM Tree Parsing	1.97	112.62	1349.00
Script Transmission (Rewriting Phase)	23	51	332
gAST Building	∼0	0.2	33
Template Matching	∼0	0.1	0.5
Runtime-included Script	9	107	1419
Symbolic Template	∼0	9	64

6.5 Performance Overhead

We first evaluate *CSPAutoGen*'s overhead by calculating the time difference of browsing Alexa Top 50 Websites with and without *CSPAutoGen*. To have a better understanding of the performance, we break down the overhead and measure the latency of DOM Tree parse, JavaScript AST generation, template matching, and handling runtime-included scripts and dynamic scripts.

Overall Latency Overhead. In this experiment, we deploy *CSPAutoGen*'s applier engine in a middlebox proxy, which runs on a 2.20 GHz Intel Xeon E5 server with 64GB RAM running Ubuntu 14.04. Other *CSPAutoGen* components, including JavaScript server, violation report server and whitelist server, all run on the same server but listen to different port. We refer to this server as *CSPAutoGen* server. The client-side machine is a 1.4GHz Intel Core i5 Mac Air machine with 8GB RAM running OSX 10.9 and Chrome 43. The latency between the client and the *CSPAutoGen* server is 0.5 ms. Then, we evaluate our Alexa dataset with and without deploying *CSPAutoGen* on mitmproxy. Each experiment is repeated five times without caching contents, and we use the median value.

Figure 4 shows the evaluation results with the blue solid line as the loading time with *CSPAutoGen* and the red dotted line as the one without *CSPAutoGen*. The median loading time are 5.94s and 5.11s respectively. When sorted by the difference of the two loading times, the median value is 470ms, i.e., 9.1%. This means that *CSPAutoGen*'s median overhead for the server and middlebox deployment is 9.1%. If deployed at client side, the client-perceived overhead should be *CSPAutoGen*'s overhead plus a proxy or extension's overhead depending on how it is implemented.

Latency Overhead Breakdown. To have a better understanding of *CSPAutoGen*'s latency, we break down its latency into the following categories (1) DOM Tree parsing, (2) scripts transmission (sending all scripts in a webpage to JavaScript server for matching from applier engine), (3). gAST building, (4) template matching, (5) handling runtime-included script, and (6) handing dynamic script. We evaluate the min, max and median value for each of them on our Alexa dataset. DOM Tree parsing and scripts transmission happen at rewriting phase; gAST building and template matching happen at rewriting and runtime phase; handling runtime-included scripts and dynamic scripts both happen at runtime phase.

Table 6 shows the evaluation results. Note that the time precision of JavaScript is 1ms, so when evaluating the latency of JavaScript-related codes, we repeat each snippet of codes to be measured for ten times and then calculate the average number. For the ones whose ten-time latency is still zero, we record them as ∼0, indicating those latency negligible. From Table 6, the biggest overhead comes from handling runtime-included script and DOM Tree parsing. For the former, it works asynchronously and thus incurs little overhead on the system; the later contributes most to the *CSPAutoGen*'s overhead. One future work is to improve the performance by replacing *CSPAutoGen*'s HTML parser (we now use html5lib [7], a Python implementation parser) with a more efficient one, such as Mozilla's DOMParser [2]. Because the *CSPAutoGen* JavaScript server and applier engine reside in the same physical server, the script transmission delay in rewriting phase is acceptable: the median value is 51 ms per webpage.

The overhead of gAST building and template matching is small: during loading time, *CSPAutoGen* processes ∼20 scripts on average for each webpage, so the estimated overall overhead of gAST building would be 4 ms and overall overhead for template matching would be 2 ms respectively.

Table 7: Script Matching Rates in Behind-the-login Functionality Experiment.

Domain	Eval	Function	setTimeout & setInterval	Runtime-included Inline Script
Amazon	100% (48/48)	100% (168/168)	99.6% (69,221/69,499)	99.8% (4,998/5,008)
Gmail	100% (134/134)	75% (3/4)	97.8% (4,313/4,411)	98.8% (2,038/2,061)
Google	100% (182/182)	N/A (0/0)	98.0% (539/550)	97.0% (875/902)
Linkedin	100% (27/27)	100% (81/81)	92.1% (70/76)	89.6% (831/927)
Yahoo	N/A (0/0)	100% (38/38)	99.8% (1,602/1,606)	95.9% (421/439)

6.6 Compatibility

In this section, we evaluate *CSPAutoGen*'s compatibility with real-world websites from two aspects: the appearance and the deep, behind-the-login functionalities.

Appearance. We evaluate whether the front pages of Alexa Top 50 websites can be correctly displayed based on the similarity scores of screenshots taken with and without *CSPAutoGen* deployed. Here is our methodology. We browse each front pages twice: at first, we deploy *CSPAutoGen* and take a screenshot of the webpage after it is loaded, which we refer to as *CSP_image*; next, we repeat the process without deploying *CSPAutoGen* and obtain another screenshot referred to as *std_image*. We then calculate the image similarity score, called CSP_score, between CSP_image and std_image, based on image histogram [26]. If CSP_score is larger than 0.9, indicating that users can hardly notice any difference [26], we will not examine the website. Note that, for about half of the 50 websites, due to the existence of advertisement and real-time contents (such as news), even consecutive screenshots of the same front page without deploying *CSPAutoGen* render a similarity score, which we refer to as std_score, less than 0.9. Therefore, if CSP_score is less than 0.9, we will manually compare these two screenshots.

The results show that the front pages of 28 websites pass our initial filtering stage, i.e., CSP_score is larger than 0.9. Then, we manually examine the rest 22 websites' front pages, and find that the differences that a human eye can notice are all caused by advertisements and news update. We further compare CSP_score with std_score for these 22 websites' front pages, and find out that the differences between two scores are all less than 0.1, which is another evidence that the low similarity scores are caused by the website itself. In addition, we find that the DOM tree structures of these webpages with and without *CSPAutoGen* are exactly the same.

Behind-the-login Functionalities. To explore the deep, behind-the-login functionalities of websites with *CSPAutoGen* deployed, we choose five major website categories based on their functionalities: email, online searching, online shopping, online social network and web portal. For each category, we manually conduct one extensive case study on the most popular website (based on Alexa ranking) that has not deployed CSP at the time of our study.

The results show that all the tested functionalities of these websites work properly with *CSPAutoGen* deployed. Table 7 shows the matching rates of unique dynamic and runtime-included inline

scripts that *CSPAutoGen* has encountered and processed in the experiment. Now we introduce the experiment details.

Email—Gmail. We register and log into a new Google account. Then, we send ten emails to different recipients with various attachments, links and photos from this account. All the ten recipients can receive the emails successfully. Next, these recipients reply to the emails with different attachments and contents. We can receive all these emails and the client-side Gmail with *CSPAutoGen* deployed can correctly render the contents. For the ones with attachments, we successfully download all of them from graphic user interface.

Online Searching—Google. We use the Google account in the Gmail experiment, and search ten different keywords on different Google products. These products include Google Search, Google Images, Google Books, Google Maps, Google Shopping, Google Videos, Google News, Google Flights and Google Apps. All the searching results can be displayed correctly. Then, we use Google's advanced search functionality, searching ten times with random conditions.

Online Shopping—Amazon. We register and log into an Amazon account. Then, we search ten different products, such as jewelry and books. After reviewing product descriptions and customer reviews, we successfully purchase a book and a coffee maker with a newly-added credit card.

Online Social Network—Linkedin. Because both *Facebook* and *Twitter* have deployed CSP, we use *Linkedin* in the experiment. We register and log into a new Linkedin account. Then, we upload a photo, publish a post, search and connect to five people. Next, we like, comment and share two posts from these connections. Lastly, we send messages to two people, one connected with the account and one not.

Web Portal—Yahoo. We register a Yahoo account via Audio code and log into the account. Then we open ten news/posts belonging to different categories. We like, comment on two news and then share them to our Facebook and Twitter accounts.

7. DISCUSSION

Inline CSS. In this version of *CSPAutoGen*, we do not disable inline CSS. However, *CSPAutoGen* is easily extendable to support disabling inline CSS without incurring compatibility issues, as what we did for JavaScript, and we plan to support disabling inline CSS in next version of *CSPAutoGen*. Note that even if *CSPAutoGen* does not support disabling inline CSS, an attacker cannot inject JavaScript embedded in CSS rules because injected scripts are blocked by *CSPAutoGen*.

Obfuscated Code. Code obfuscation does not influence the generation of gAST and symbolic templates, because obfuscated code is still parsed and executed in the browser. In addition, the number of flexible types in obfuscated code is similar to the one in normal code, because most of the strings in obfuscated code are inferred as const or enum type.

Event Triggering. In the headless browser cluster of the training phase, we triggered all the registered events. However, the triggering does not have any effects on the template generation. The reason is that we obtain all the scripts in the DOM, no matter they are triggered or not. The only exception is that when dynamic scripts or runtime-included scripts are embedded inside an event handler, but in practice we did not find any web developers did so.

Script Execution Sequence. *CSPAutoGen* does not change the original script execution sequence, i.e., both synchronous and asynchronous scripts are still executed the way as they are. For example, synchronous scripts inside *eval* or *eval*-like functions are executed synchronously through symbolic templates, an essentially built-in function call; asynchronous, runtime-included, inline scripts are executed inside DOM event handlers, an asynchronous script execution method.

8. RELATED WORK

In this section, we first introduce several works in automatic enforcement of CSP. Then we present past works preventing XSS attacks. Lastly, we discuss related works in academia and industry using rewriting and AST techniques.

Automatic Enforcement of CSP. Researchers have proposed several interesting works on automatic enforcement of CSP [12, 14, 17, 23]. The first work of facilitating CSP adoption is deDacota [12]. Primarily, it statically rewrites ASP.NET applications to separate data and codes; AutoCSP [14] uses dynamic taint analysis in PHP to find trusted elements of dynamically generated HTML pages and then infers a policy to block untrusted elements, while allowing trusted ones. Both AutoCSP and deDacota are white-box approaches, i.e., they need to access target application's codes and require server modifications. Moreover, neither AutoCSP nor deDacota can securely transform runtime-included inline scripts or dynamic scripts to comply with the strictest CSP. As a comparison, *CSPAutoGen* requires no server-side modifications, and allows trusted runtime-included inline scripts and dynamic scripts.

A black-box approach, autoCSP [17], first generates strictest policies and then gradually relaxes them by adding the scripts received from users' violation reports to its whitelist. Kerschbaumer *et al.* [23] adopt crowdsourcing approach to collect JavaScripts' hashes and then generate CSP rules based on these collected hashes, i.e., they adopt strict string matching for inline scripts. Neither of the aforementioned approaches can process inline scripts with runtime information, runtime-included inline scripts and dynamic scripts. According to our manual analysis, 46 of Top 50 Alexa websites contain such script usages. That is, they cannot be deployable with real-world websites. As a comparison, *CSPAutoGen* is compatible with all Alexa Top 50 websites.

To defend against XSS attacks, other than CSP, researchers have also proposed many approaches before, which can be further classified as sever-side methods [9, 10, 19, 21, 22, 29, 30, 36–38, 40, 42] and client-side methods [20, 32]. We focus on the deployment of CSP, because it has been adopted by all major browsers and standardized by W3C [1].

Server-side Defenses of XSS Attacks. XSS-GUARD [10] dynamically determines legitimate scripts and removes illegitimate ones from responses. However, since it only works at server side, XSS-GUARD cannot determine whether dynamic scripts are legitimate or not. As a comparison, *CSPAutoGen* can address both static scripts and dynamic scripts. BLUEPRINT [30] proposes an approach to ensure the safe construction of the intended HTML parse tree on the client without changing browser. Template-based approaches [36, 37] propose novel web frameworks that incorporate correct sanitization based on contexts. There are also many works based on server-side input sanitization [9, 19, 40]. All the aforementioned approaches require server-side modifications. As a comparison, *CSPAutoGen* has a much more flexible deployment model, which can be at a server, client, or middlebox.

Many other XSS defenses are based on flow analysis or taint tracking [21, 22, 29, 29, 31, 38, 42]. Compared with *CSPAutoGen*, such programming analysis requires server modification and is bound to a certain programming language. For example, Taj [42] works specifically for Java, while Pixy [21] only operates on PHP.

Client-side Defenses of XSS Attacks. Lekies *et al.* [27] focus on detecting DOM-based XSS vulnerabilities using taint analysis approach. Saxena et al. [39] highlight a new class of vulnerabilities, referred to as client-side validation vulnerabilities, and propose a

dynamic analysis system to discover it. Noxes [25] is a client-side firewall-based defense that protects users with permit/deny rules to restrict HTTP requests. Similar to CSP, BEEP [20] and ConScript [32] are policy-based approaches: websites specify security policies and the client-side browser enforces these policies. Unlike CSP used in *CSPAutoGen*, they are not supported by existing browsers so client-side modifications are required.

Rewriting Technique. *CSPAutoGen* rewrites webpages to automatically generate CSPs without compromising compatibility. Rewriting techniques have been widely used in academia [13, 24, 28] and industry [4, 6]. In academia, WebShield [28] rewrites webpages to enable web defense techniques. Erlingsson *et al.* [13] enforce security policies on binaries by taking advantage of rewriting techniques. In industry, ShapeSecurity [6], a commercial product, rewrites websites to prevent bot and malware. Google's PageSpeed Module [4] improves websites' performance by rewriting webpages.

AST Technique. Abstract Syntax Tree (AST) has been widely used by web security researchers to extract JavaScript structure information [11, 18, 35]. However, these works do not aim to generate script templates and thus cannot address the challenges of inline scripts with runtime information or dynamic scripts.

9. CONCLUSION

In conclusion, we propose *CSPAutoGen*, which generates CSPs from training dataset, rewrites webpages in real-time to insert CSPs, and applies CSPs at client browsers. *CSPAutoGen* is able to deal with all the real-world yet unsafe script usages, such as inline scripts with runtime information and dynamic scripts, and securely convert them to be compatible with CSP.

Our evaluation shows that *CSPAutoGen* can correctly render all Top 50 Alexa websites. The correctness evaluation that compares generated gAST templates with web framework source code shows that *CSPAutoGen* can successfully infer type information with 95.9% accuracy. In addition, *CSPAutoGen* only incurs 9.1% overhead in median when rendering the front pages of Alexa Top 50 websites.

10. ACKNOWLEDGEMENTS

This work is collectively supported by U.S. National Science Foundation (NSF) under Grants CNS-1646662, CNS-1563843, by National Natural Science Foundation of China (NSFC) under Grant No. 61472359, and by Defense Advanced Research Projects Agency (DARPA) under Agreement No. FA8650-15-C-7561. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, NSFC, DARPA, or the Government.

11. REFERENCES

[1] Content Security Policy. http://www.w3.org/TR/2012/CR-CSP-20121115/.
[2] Domparser. https://developer.mozilla.org/en-US/docs/Web/API/DOMParser.
[3] Esprima. http://esprima.org/.
[4] Pagespeed module: open-source server modules that optimize your site automatically. https://developers.google.com/speed/pagespeed/module/.
[5] Scrapy | a fast and powerful scraping and web crawling framework. http://scrapy.org/.
[6] Shape security. https://www.shapesecurity.com/.
[7] Standards-compliant library for parsing and serializing html documents and fragments in python. https://github.com/html5lib/html5lib-python.
[8] VirusTotal. https://www.virustotal.com/.
[9] 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 *IEEE S&P*, 2008.
[10] P. Bisht and V. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *DIMVA*. 2008.
[11] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security*, 2011.
[12] 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 *SIGSAC*, 2013.
[13] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *IEEE S&P*, 2000.
[14] M. Fazzini, P. Saxena, and A. Orso. AutoCSP: Automatically Retrofitting CSP to Web Applications. In *ICSE*, 2015.
[15] D. Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.
[16] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary. Towards online spam filtering in social networks. In *Proceedings of Network and Distributed Systems Security Symposium*, NDSS, 2012.
[17] N. Golubovic. *autoCSP: CSP-injecting reverse HTTP proxy*. B.S. Thesis, Ruhr University Bochum, 2013.
[18] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security*, 2009.
[19] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security*, 2011.
[20] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.
[21] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE S&P*, 2006.
[22] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS*, 2006.
[23] C. Kerschbaumer, S. Stamm, and S. Brunthaler. Injecting csp for fun and security. In *ICISSP*, 2016.
[24] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SIGOPS*, 2007.
[25] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC*, 2006.
[26] S. Lee, J. Xin, and S. Westland. Evaluation of image similarity by histogram intersection. *Color Research & Application*, 2005.
[27] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.
[28] Z. Li, Y. Tang, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS*, 2011.
[29] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security*, 2005.
[30] M. T. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE S&P*, 2009.
[31] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX Security*, 2008.
[32] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE S&P*, 2010.
[33] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif. Misleadingworm signature generators using deliberate noise injection. In *IEEE S&P*, 2006.
[34] PhantomJS. PhantomJS . http://phantomjs.org/.
[35] E. B. Pratik Soni and P. Saxena. The SICILIAN Defense: Signature-based Whitelisting of Web JavaScript. In *CCS*, 2015.
[36] W. K. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *USENIX Security*, 2009.
[37] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS*, 2011.
[38] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE S&P*, 2010.
[39] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.
[40] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, 2011.
[41] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, 2010.
[42] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. *SIGPLAN*, 2009.
[43] G. Wang, T. Wang, H. Zheng, and B. Y. Zhao. Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In *USENIX Security*, 2014.
[44] R. Wang, W. Enck, D. S. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *USENIX Security*, 2015.
[45] M. Weissbacher, T. Lauinger, and W. Robertson. Why is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*. 2014.
[46] XCampo. A XSS payload generator. https://code.google.com/p/xcampo/.