Mocha:Automatically Applying Content Security Policy to HTML hybrid applicationlication on Android Device

***Abstract*—An HTML hybrid applicationlication is a type of application running on mobile devices. It is popular but may have Cross Site Scripting(XSS) vulnablitiy risks. Content Security Policy(CSP) is a security mechanism that can prevent XSS attacks. Developers can only apply CSP to applications, therefore user's safety depends on them. In this paper, we propose Mocha, automatically applying CSP to applications on Android device. Mocha uses static analysis to automatically infer CSP policies, and modifies HTML and JavaScript source code for applying CSP. Mocha can protect users and their HTML hybrid applications from XSS. We confirmed that Mocha is effective with real applications.**

***Index Terms*—cross-site scripting, CSP, security, Android, HTML hybrid application**

I. Introduction

HTML hybrid applications that consist of HTML, JavaScript and CSS files are one of applications runnning on mobile devices. They have a cross platform aspect, and operate using WebView which is common to each OS of mobile devices. However there is concerned about XSS vulnerabilities because HTML and JavaScript are used in HTML hybrid application.

XSS is an attack using vulnerabilities that crackers can execute an arbitrary JavaScript in a web application. Attackers can alter HTML pages and steal cookies from browsers. XSS attacks are just aimed at JSON formatted data and strings received from HTTP GET and POST methods in case of non-HTML hybrid applications. However, HTML hybrid application have many channels for XSS, because they can recieve various data, including meta information of images and audio files, SMS, Contact, Wi-Fi, access point names of Bluetooth, etc[1]. Therefore, countermeasures against XSS are more important in HTML hybrid applications.

There are several countermeasure methods against XSS vulnerabilities in general. HTML hybrid applications can use them as well. Representative methods are "escaping", which is the way to escape special characters included in the input/output HTML data, and an XSSfilter in browsers[2][3]. However, the methods cannot protect the applications perfectly. For instance, "escaping" can overlook vulnerabilities and mistake how escape them. Moreover, it has been reported[4] that there is an attack which can pass through the filter. Finally, existing countermeasure cannot preserve users from all XSS attacks.

Content Security Policy(CSP)[5] exists as a complete defense against XSS. CSP restricts HTML contents and offers comprehensive protection against XSS. CSP is supported by major browsers[6] and is also widely used in Web applications, Twitter and Facebook. To use CSP on applications, developers write the CSP header and declare the policy in the HTTP response header. HTML contents permitted by this policy are only contained in the applications. Although CSP is an effective defense mechanism against XSS, protection of users is dependant on developers who set CSP in applications[7]. In addition, it is difficult to protect from XSS by browser extensions applying CSP to applications because CSP restricts the action and affects the behavior and structure of applications[8]. In this paper we focus on the fact that the source code of HTML hybrid applications exists on Android devices and propose Mocha that automatically applies CSP to applications in the user side.

Mocha can run on Android devices and automatically apply CSP to HTML hybrid applications specified by users. It statically analyzes the HTML hybrid applications on Android devices, automatically declares the policy, modifies HTML and JavaScript files caused by the policy declareation, and reinstalls the application on the device. Mocha makes it easy for users to take measures against inadequate setting and implementation by developers.

The rest of paper is organized as follows: Section 2 gives a brief overview of CSP. Section 3 explains about Mocha and details of it. Section 4 talks about implementation of Mocha and Section 5 discusses evaluation of it. Related works are surveyed in Section6 and the paper is summarized in Section 7.

II. CSP

This section describes CSP. We first provide an overview of CSP. Second we discuss about policy directive, the basis of CSP, and then discribe problems of CSP.

A. Overview

CSP is a mechanism that can prevent XSS attacks and restricts the origin destination of the web contents based on the whitelist in the application. CSP takes effect when the HTTP response header as "Content-Security-Policy" are used and the policy of the contents as the policy directive are declared in the header. The WebView in HTML hybrid applications read the application's HTML files, recieve policy directives in the header, and forces scripts in the HTML files to execute under the environment declared by the policy directive.

B. Policy Directives

Policy directives are declaration for restricting the source of web page contents. The announcement of the server host-name in the policy directive limits the sender of JavaScripts, external plug-ins, and CSS to the server. The format of the policy directive takes the form "`directive-name resource-domain`". A multiple "`directive-name`" can be declared. The contents providers repeat enough declare-tions and can specify the policies in detail. Also, you need to separate policy directives with semicolons in repeating.

An example of policy directives using a meta tag is shown in the Figure 1. The policy directives declared in Figure 1 include three types as "`default-src *`", "`script-src 'self' 'http://www.test.jp'`" and "`style-src 'self' 'unsafe-inline'`". The `script-src directive-name` limits the origin host of scripts. Similarly, the `style-src directive-name` restricts the source host of the styles. The part of "`resource-domain`" defines the host name allowed to send scripts and styles. For example, if you need to load the `http://www.test.jp/test/index.js` file, you have to describe `http://www.test.jp` in the `resource-domain`. In the `resource-domain`, it is possible to use `keywords` as well as host names. There are four kinds of `keywords`: `none`, `self`, `unsafe-inline` and `unsafe-eval`. The `none` keyword refuses to read from all host. The `self` keywords only allows to recieve contents from oneself. In HTML hybrid applications, it means only to permit loading contents in the package of the application. The `unsafe-inline` keyword is available only when `directive-name` is `script-src` and `style-src`, and allows inline scripts and inline styles in HTML files. The `unsafe-eval` keyword can be accepted only when `directive-name` is `script-src`, and permits using some JavaScript methods such as "eval" and "Function()", which evaluate strings as JavaScript. Using keyword `unsafe-inline` and `unsafe-eval`, inlines and evals are freely available and are not restricted by CSP. As show in Figure 1, the `resource-domain` can use metacharacters "*". It means that CSP allows loading from any host. To summarize, the policy directive shown in Figure 1 permits reading JavaScript from its own domain and www.test.jp, reading CSS from its own domains, using inline styles, and reading other content from any domain.

DOM based XSS in HTML hybrid applications can be prevented by not including `unsafe-inline` and `unsafe-eval` in the `script-src` of the `directive-name` of the policy directive.

C. Problems of CSP

In order to apply CSP to applications, two procedures are required. First, developers declare policy directives for all HTML files. Next developers modify HTML and JavaScript files to follow the policy directive declarations. Problems on the both sides of developers and users arise from these procedures. In order for application developers to apply CSP, it takes time for setting policy directives and source code modifications. In addition, an error accompanying correction can also occur. Therefore the procedures is a burden on the developers. Also, when users apply CSP with extended function of the browser[2][3], there is a concern that the operation of the application will be affected, since the source code can not be modified. Because inline scripts are currently used in many applications[8], it is difficult to prevent XSS only by declaring policy directives.

```
<meta http-equiv="Content-Security-Policy"
content="default-src *; script-src 'self'
http://www.test.jp; style-src 'self' 'unsafe-inline';">
```

Fig. 1. example of policy directive

III. Proposal and Design

In this chapter, we talk about overview of Mocha which applies CSP to HTML hybrid applications at Android devices in III-A. Section III-B and subsequent sections describe items that need to be implemented.

A. OverView

We propose Mocha that automatically applies CSP to existing HTML hybrid applications. Using Mocha makes it possible to apply CSP regardless of application developers and users, and to prevent XSS in HTML hybrid applications.

Server side programing languages can dynamically output HTML. Hence contents of the HTML are interpreted when the web application is executed. However, because HTML hybrid applications do not use the server side programing languages such as php or perl, there are few elements to be determined dynamically. Therefore it is possible to apply CSP on Android deveices only by static analysis. Further, the time required for developing applications can be shortened because it is not necessary to perform a preliminary dynamic analysis.

The process for applying CSP to an HTML hybrid application on an Android device is shown in Figure 2. First, 1)Mocha displays a list of applications excepted pre-installed on the Android device at the setting screen. When users select an application and push the applying CSP button, Mocha starts to apply CSP to it. 2)It decompiles an apk file and extracts HTML, JavaScript and CSS files. 3)It statically analyzes HTML and JavaScript files and sets policy directives. 4)It modifies HTML files of the application to conform them to the policy directive, and 5)modifies JavaScript files as well. 6)Finally it compresses the changed source code in apk and installs the new compressed apk on the Android device.

B. Setting of Mocha

We designed a setting interface to select an application to apply CSP and its certificate. We described the certificate in Section IV-E

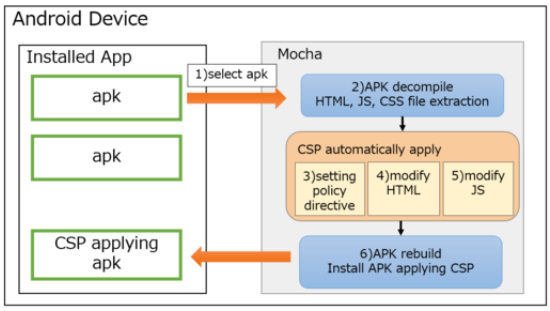

Fig. 2. Overall view of Mocha

C. Decompile of apk

It is necessary to decompile apk in order to set the policy directive and modify the source code related it. An entity of apk is a zip archive file. HTML, JavaScript and CSS files exist in the directory after extracted apk. Mocha uses these files for setting policy directives, fixing HTML and modifying JavaScript. Also if the application selected by the user isn't HTML hybrid application, Mocha notifies that CSP is not available for it.

D. Declare Policy Directive

Mocha sets a policy directive to prevent XSS. Therefore, Mocha doesn't include keywords `unsafe-inline` and `unsafe-eval` in `script-src` as described in Section II-B. Also, Mocha doesn't include the keyword `unsafe-inline` in `style-src`. If contents required for the application are on the external host, Mocha acquires the host name by analyzing HTML and JavaScript files.

E. Modifying HTML

By setting the policy directive described in Section III-D, inline scripts, inline styles, event handlers and javascript:URI styles can not be performed in HTML. Therefore, it is necessary to modify them.

F. Modifying JavaScript

As in Section III-E, it is necessary to modify the JavaScript files too. A method for dynamically generating JavaScript and a method for evaluating a character string as JavaScript can not be performed, hence they are modified by Mocha.

G. Recompile of apk

In order to install the application after applying CSP, Mocha compresses corrected files up to Section III-F into the zip archive and signs it. Then, Mocha uninstall the original application and install the application which applied CSP and signed.

IV. Implement

In this chapter, we describe implementation of Mocha. We describe methods of decompiling the apk file, setting policy directives, modifying HTML and JavaScript files and rebuilding the apk file. In modifying HTML and JavaScript files, Mocha overwrites original files.

```
1: $.ajax({
2: type:'POST',
3: dataType:'json',
4: url: 'http://www.test.jp/api.php,
5: ···});
```

Fig. 3. Example of external host with jQuery

A. Decompiling the original apk file

An apk file is a zip archive file described in Section III-C. The apk file of the installed application is stored in the /data/app/ directory of the Android device. Mocha copies the apk file of application selected by the user to Mocha's working directory, extracts the copied apk file, and stores the HTML, JavaScript and CSS files for setting policy directives.

B. Setting policy directives

To set the policy directive, Mocha prepares the basic policy directive and extended one. The basic policy directive is "`default-src *; script-src 'self'; style-src 'self';`" and means to restrict loading JavaScript and CSS files from outside the package. Also, XSS can be defended by not including `unsafe-inline` and `unsafe-eval` in `script-src` of the basic policy directive. Mocha does not include `unsafe-inline` in `style-src` hence XSS does not occur even when the application is diverted to the environment using the Internet Explorer browser[9].

The server name described in the `resource-domain` is obtained by analyzing the HTML and JavaScript files. The way to acquire the host name consists of analyses of HTML files and JavaScript files. In HTML files, Mocha can acquire it from the src attribute of the script tag or the style tag. In JavaScript files, Mocha can get it from the url for jQuery. An example of the external host using jQuery is show in Figure 3. Mocha acquires the host name described in the url of the fourth line in Figure 3.

C. Modifying HTML files

In this section, we describe modification of HTML files. Mocha modifies them using JSOUP[10]. JSOUP is an HTML parser written in Java and provides a set of API which can be used for analyzing and modifying HTML files using DOM.

Mocha needs to adjust inline scripts, inline styles, event handlers and javascript:URI according to policy directives. First, inline scripts are JavaScript written in HTML files using <script> tags. Mocha extracts JavaScript in script tags with regular expression and writes them to an external file. Then Mocha changes the original HTML file to load the external file. A modification example is shown in Figure 4. As shown in Figure 4, the JavaScript in the script tag is written to an external file and loaded by the src attribute of the script tag.

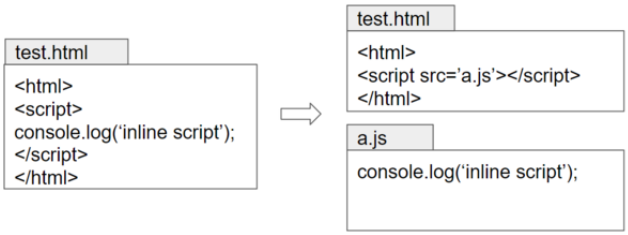

Fig. 4. Example of inline script modification

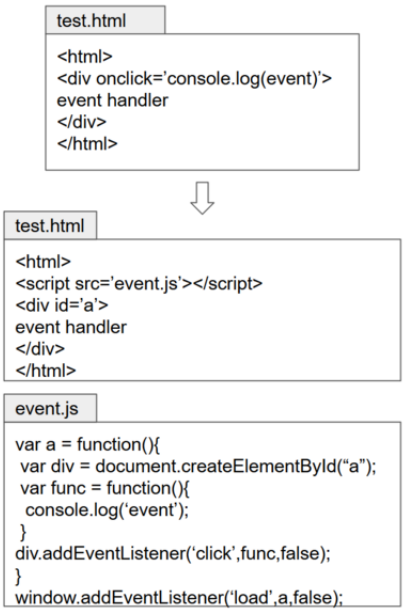

Fig. 5. Example of event handler modification

Next, inline styles are a CSS written in HTML files using <style> tags or a style attribute. Mocha detects a CSS in HTML files using regular expression, converts them to inline style, and writes them into an external file. The original HTML files are modified to get them from the external file as well as modification of inline script.

Next, We describe modification of event handling, which is a method for giving specific processing to the user's action such as click or double click. Mocha alters event handlers to the way using the addEventListener function that uses DOM as shown in Figure 5. Event names which are used with addEventListener differs from the names used in event handlers, but the correspondence between names is defined in W3C[11].

Finally, we describe modification of javascript:URI which executes JavaScript when using "javascript:..." in the href attribute of the HTML tag. Mocha alters it by the same method as the evnet handler. However, when javascript:URI and an event handler exists in the same HTML tag, Mocha considers the pre-defined priority order between them[12] and modifies the HTML files.

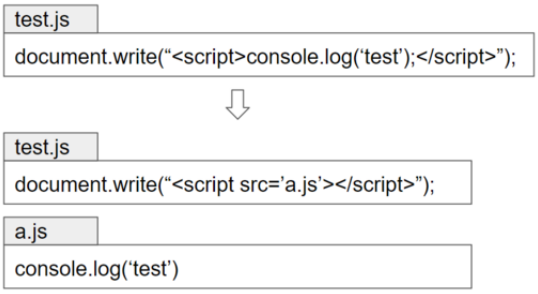

Fig. 6. example of modfying document.write

D. Modifying JavaScript file

We describe the modification of JavaScript files. First of all, we develope and use the original parser with Java to modify JavaScript files. Our parser provides a set of APIs to get the user-specified methods and variables in the JavaScript files. There are two targets to alter JavaScript files to fit the policy detectives; dynamic HTML tags and string-evaluation functions.

At first, we illustrate how to alter dynamic HTML tags. There are four methods dynamically to create HTML tags in DOMAPI: document.write, document.writeln, innerHTML and outerHTML[1]. It is necessary to modify JavaScript files when generating script tags with these methods. Mocha modifies the tags to loading external files. Figure 6 shows the example of modification when document.write is used and Figure 7 shows the example of modification when innerHTML is used.

Next, we explain the modification of string-evaluation functions. There are four methods which evaluate strings as JavaScript: setInterval, setTimeout, eval and Function(). Of these, setInterval and setTimeout can take character strings and a function as an argument. Taking strings as an arugment violates the policy directive. Hence, Mocha changes the strings to the function as an argument. Because eval and Function() have multiple usage, Mocha modifies JavaScript files for each usage. There are 10 kinds of usage according to Richard et al[13]. Since the four of 10 methods are not describe how to modify them, encoding JavaScript, which converts arguments of eval and Function() to Unicode, can prevent XSS. Therfore Mocha performs it. If Mocha fixes this way, Mocha has to use `unsafe-eval` for `script-src` of the policy directive.

E. rebuild of apk

We describe the recompilation of an apk file to reinstall the application applied CSP on the Android device. Mocha compresses files except files in the META-INF/ directory to the zip archive file and signs it with jarsigner. Users can choose which certificate, a Mocha provided or a user generated, at the setting screen described in Section III-B

V. Evaluation

In this section, we describe evaluation Mocha on the Android device. We consider measurement of processing-time

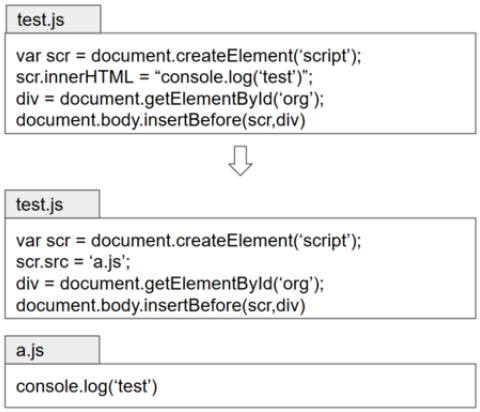

Fig. 7. example of modifying innerHTML

TABLE I
ENVIRONMENT OF EVALUATION

Android device	Zenfone 3
Android OS	AndroidOS 5.1.1

required applying CSP to the appliation, application behavior after applying CSP, the number of modified source code, and the installed policy directives.

A. Experiment setup

We apply CSP to HTML hybrid applications using Mocha. We used 26 HTML Hybrid applications published on Fossdroid[14]. First, we measure the time taken applying CSP using the `currentTimeMillis()` function of Java and confirm whether CSP works or not on the Android device. Finally, we investigate the number of modified source code required applying CSP and details of the installed policy directives. We classify modified source code into HTML files and JavaScript files and count the number of modifications independently. The Android device we used for evaluation is shown in Table I.

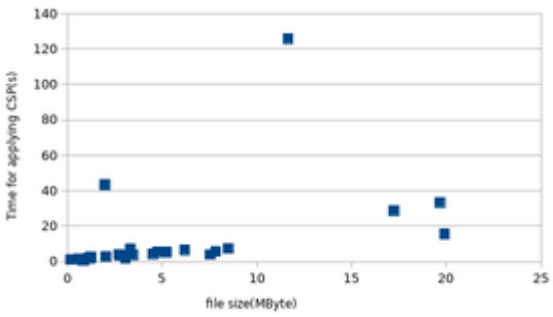

Fig. 8. Relationship between time for applying CSP and file size

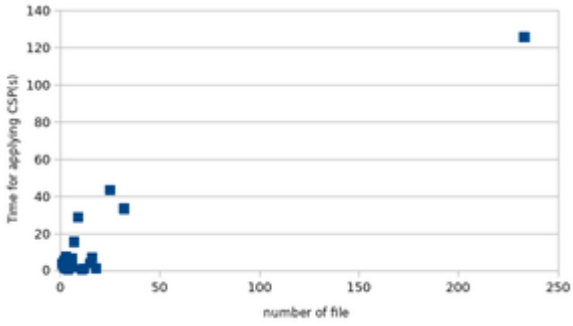

Fig. 9. Relationship between time for applying CSP and number of files

B. Time required applying CSP

We measure the time taken to apply CSP to the application after the user pushes the button applying CSP. The result time is 11.896 seconds on the average, and 125.786 seconds at the longest. Figure 8 shows the relationship between the processing time and the apk file size, and Figure 9 shows the relationship between the time and number of HTML and JavaScript files. Figure 8 demonstrates that the processing time becomes longer in proportion to the apk file size because the time includes decompiling, rebuilding, and signing the apk file and their times are all proportional to the size. Figure 9 represents that the processing time is proportion to number of HTML and JavaScript files because static analysis for HTML and JavaScript files spends the most part of the time and its analysis time is proportion to the number of files. There are a deviational application from the proportional relation in Figure 8 and an application which has over 200 files in Figure 9. These are the same application. From this, it can be said that the influence on the processing time is longer by the file size of apk than the number of HTML and JavaScript files.

C. Result of modified source code

We examined the number of alterations in the source code. The number of changes in HTML files is shown in Table II and the number of changes in the JavaScript files is shown in Table III. All applications which we used for evaluation can run after applying CSP. There are only nine applications that don't need to be modified for both HTML and JavaScript files. Therefore, it can be said that modifying HTML and JavaScript files are needed to correctly apply CSP directives.

We discuss the installed policy directives in this paragraph. Seting the policy directives are correctly done because the external files were not restricted by the CSP policy. However, because the security of the external host may not be guaranteed[15], it is necessary to download files on external hosts and include them in the apk file in the future.

In modifying HTML files, no abnormality occurs in operations of applications after modification, because there is no server side languages in HTML hybrid applications. Regarding modification of JavaScript files, since contents of JavaScript may be determined dynamically, in some cases,

TABLE II
NUMBER OF MODIFYING HTML FILES

number of violations	number of applications
0	17
1-5	7
6-10	2
11-30	1

TABLE III
NUMBER OF MODIFYING JS FILES

number of violations	number of applications
0	15
1-10	8
11-30	3
31-	1

abnormality may occur in the operations of the modified applications. For this reason, it is necessary to implement a mechanism to change their behavior depending on contents of dynamic elements at the runtime. Moreover, Mocha can not handle JavaScript compression and obfuscation. Therefore, these points are future tasks.

VI. Related Work

In this Chapter, we describe related works. We describe techniques on CSP's approach to applications and compare with Mocha.

A. AutoCSP

AutoCSP[16] is a method of applying CSP automatically to Web applications using PHP. It uses dynamic taint analysis and realizes advanced defense against stored XSS by running with judging whether JavaScript is reliable or not. It also supports developers when writing HTML using PHP. However, AutoCSP does not provide modification of JavaScript files and its users can not use it independently. These are different from Mocha.

B. UserCSP

UserCSP[8] is implemented as an extension of FireFox, and it is a mechanism that dynamically analizes and applies CSP when a Web browser loads HTML. UserCSP has proposed to help developers to apply CSP and users to protect from XSS attacks. However, since UserCSP can not modify the source code, it can not be said that protection of the Web application using inline scripts and evals from XSS is not sufficient. Mocha can provide higher protection than UserCSP because source code can be modified.

C. CSPAider

CSPAider[17] is a mechanism for presenting better policy direcitives for developers of web applications. CSPAider sets the appropriate policy directive with analyzing the Web application. Because the CSPAider was proposed at the CSP plannning stage, the form of the policy directive is different from what is currently used. Also, CSPAider suggests only policy directives. It differs from Mocha in no modification of HTML and JavaScript files.

VII. Conclusion

In this paper, we proposed Mocha, automatically applying CSP with only static analysis for HTML hybrid applications on Android devices. Mocha became possible for users to prevent XSS attacks by automatically applying CSP. As a result of evaluating HTML hybrid applications using Mocha, we confirmed that setting policy directives and modifying HTML files could be correct by only static analysis. Regarding JavaScript modification, there are four problems to be solved; receiving data from outer servers, dynamic arguments of functions, compression, and obfuscation but we confirmed that other modifications are possible. In the future, we will consider a method to investigate and resolve dynamically determined arguments just before executing the function.

References

[1] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 66–77(2014).

[2] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 19th international conference on World wide web*, pp. 91–100(2010).

[3] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 601–614(2011).

[4] G.Heyes, "Bypassing xss auditor," http://www.thespanner.co.uk/2013/02/19/bypassing-xss-auditor/.

[5] W3C, ""Content Security Policy 2.0"," http://www.w3.org/TR/CSP.

[6] caniuse, "Can i use content security policy?" http://caniuse.com/contentsecuritypolicy.

[7] H. Kour and L. S. Sharma, "Browser compatibility issues in implementing content security policy to prevent cross site scripting attacks," *International Journal of Modern Computer Science, pp.108-112(2012)*.

[8] K. Patil and B. Frederik, "A measurement study of the content security policy on real-world applications." *IJ Network Security*, vol. 18, no. 2, pp. 383–392(2016).

[9] Ruby, "Ruby on rails security guide," http://guides.rubyonrails.org/security.html#css-injection.

[10] jsoup, "jsoup: Java html parser," http://jsoup.org.

[11] W3C, "Ui events specification," https://www.w3.org/TR/DOM-Level-3-Events/.

[12] CodeDay, "Detailed html a label href and onclick usage, distinction, priority level," https://www.codeday.top/2017/06/17/25323.html.

[13] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *ECOOP 2011–Object-Oriented Programming*. Springer, pp. 52–78(2011).

[14] D. Simonin, "Fossdroid," https://fossdroid.com/.

[15] G. S. Blog, "Csp evaluator," https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html.

[16] M. Fazzini, P. Saxena, and A. Orso, "Autocsp: Automatically retrofitting csp to web applications," in *the Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[17] A. Javed, "Csp aider: An automated recommendation of content security policy for web applications," in *IEEE Symposium on Security and Privacy*, 2011.