CDLint: A Cloud Service for Dynamic Analysis of JavaScript Code

류샤오*, 우균*,**
*부산대학교 전기전자컴퓨터공학과
**LG전자스마트제어센터
e-mail:{liuxiao, woogyun}@pusan.ac.kr

CDLint: A Cloud Service for Dynamic Analysis of JavaScript Code

Xiao Liu*, Gyun Woo*,**
*Dept. of Electrical and Computer Engineering, Pusan National University
**Smart Control Center of LG Electronics

Abstract

This paper presents a cloud service called CDLint for checking and analyzing the Javascript code dynamically. The correctness of Javascript code is becoming more important since it can also run on the server side as well as the client side. There are several analysis systems for checking the bad code in JavaScript but they seem like have one or more weaknesses. CDLint is developed based on an existing work named DLint which is a powerful bad JavaScript checker. Compared with similar systems, CDLint shows the best performances with respect to the system extensibility, the freedom to use without installation, the automatic parsing of JavaScript code from website, and the environment configuration for JavaScript code checking.

1. Introduction

In this paper, we provide a new cloud service called CDLint that is a cloud service for dynamically checking vulnerabilities in JavaScript code. The uses of Javascript are becoming more widespread in a few past years [1]. Javascript was used only on the front-end of Web application for handling value transmission in DOM elements until a few years ago [2]. Since the Node.js was published, Javascript and its plenty of server-side frameworks have become to be used on the back-end [3].

Meanwhile, since JavaScript is not a 'well-formed' language, it has many pitfalls that developers should avoid [4]. Developers usually refer the guidelines of corresponding programming languages to avoid common pitfalls. There are books such as Herman's [5] introducing several rules on code quality for developers to follow and to improve the overall software quality by solving security vulnerabilities and bugs, improving the performance and the maintainability.

On the other hand, there are several automatic code checking techniques which are available for developers to rely on them checking the code quality rules [6]. The lint [7] based checking rules such as JS-Lint, JSHint, ESLint, and Closure Linter are wildly used by developers for their JavaScript code.

Those existing static analysis tools have limitations on analyzing complicated syntax such as for-in loops and eval functions. There is a study named DLint that gathers 28 famous checkers in their system to improve the ability of checking bad code to detect most common pitfalls related to the inheritance, types, language and API usage issues, and uncommon values [8].

CDLint is an improved version of DLint which is deployed on the cloud letting JavaScript developers use this service without installing any tools. CDLint is deployed on a Node.js server and takes advantages of features in DLint. Further, it provides a Web GUI for developers to analyze their JavaScript code. There are two approaches to access CDLint for the JavaScript code analysis; one is to upload the JavaScript code through web interface and the back-end of CDLint will analyze it; the other way is to input the web address which directed to the corresponding web page containing the JavaScript code that the developers want to analyze, then CDLint will parse the web code and find corresponding JavaScript code then analyze it.

The rest of this paper is organized as follows. We

discuss on the related work in Section 2 including DLint and other similar methods. The architecture and features of CDLint is presented in Section 3. Section 4 compares CDLint to other systems for demonstrating the usability of CDLint. Section 5 concludes.

2. Related work

ESLint is a static analysis tool for JavaScript code written using Node.js [9]. Since ESLint is an open source project and based on Node.js, it can be installed through npm, a Node.js package manager program. ESLint provides useful configurations such as JavaScript code executing environment settings and several checking rules. However, since JavaScript is a dynamic and loosely-typed language, the static analysis may be not able to find all bugs in JavaScript.

JSHint is another open source project for detecting errors in JavaScript code [10]. There are different ways to use JSHint such as installing it as a program through npm, or as a plugin which is available on popular code editors and IDEs like Vim, Emacs, Sublime Text, and Visual Studio. However, similar with ESLint, the error detection in JSHint still is based on the static analysis mechanism.

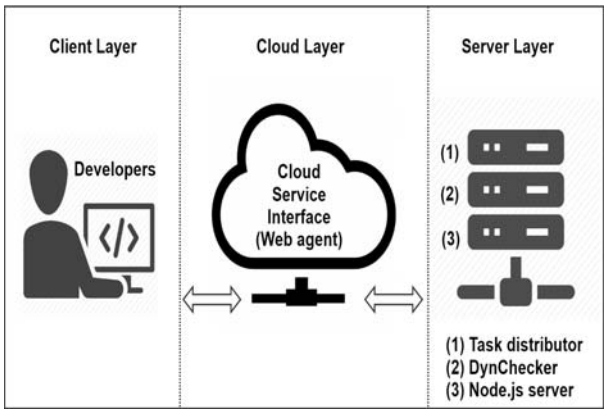
DLint is a dynamic analysis tool for detecting violations of JavaScript codes. DLint consists of a extensible framework and several JavaScript code checkers based on it. Each of the checkers handles a specific rule and detects violations of it. Contrasting with other similar tools, DLint focuses on detecting rules that other systems are not able to cover. Examples include inheritance, type errors, misuse of APIs, and uncommon values.

All these tools have their own features and disadvantages, either. Our proposed method takes advantages of the features of those tools and avoids their disadvantages.

3. Architecture and features of CDLint

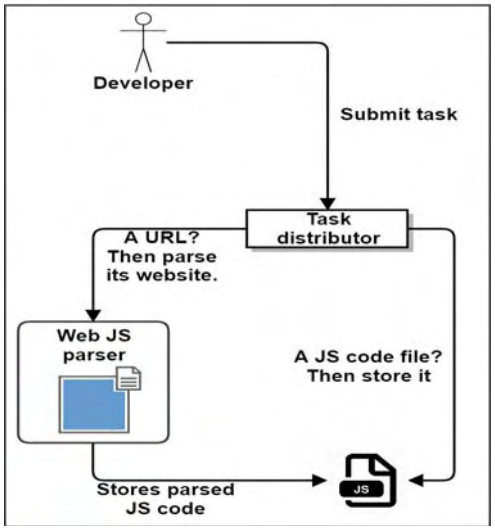
To make our system usable without any installation of programs, CDLint is deployed on the cloud. The services in CDLint can be provided as a Web application. Figure 1 illustrates the 3-tier layer architecture of CDLint.

At the client layer of CDLint, developers can access the services of CDLint through web on the cloud layer. There are two approaches for developers to use the CDLint: passing the URL of target website to analyze the JavaScript code runs on it or uploading the JavaScript code directly. The web agent at the cloud layer will send whether the URL of target website or the JavaScript code to the task distributor at the server layer.



(Figure 1) The 3-tier layer of CDLint. Developers can use a web browser to access CDLint. The main functions of CDLint are running on a Node.js server.

Figure 2 presents the task distribution mechanism. Once the task distributor receives the analysis task as a URL, it will parse the corresponding website to find all JavaScript code runs on it and saves it as a temporary JavaScript source code file then distribute it to the DynChecker. Otherwise, if the task distributor receives the analysis target as a JavaScript source code file, it will be send to the DynChecker directly.

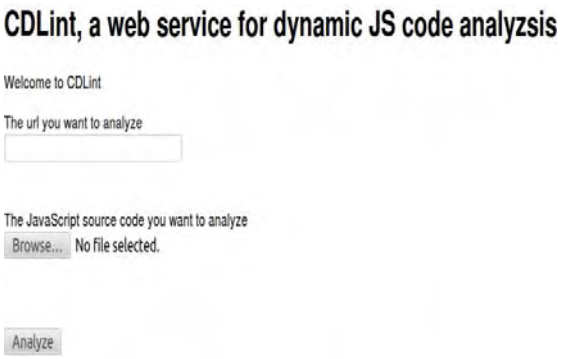


(Figure 2) The task distribution mechanism of CDLint. This mechanism decides how to store target JavaScript code from website URL or source code.

Since the CDLint is developed based on DLint which has the function that can extend the JavaScript code checkers as much as users want. CDLint also allows developers to upload their own JavaScript checkers as a .js file to CDLint system to let them work together with the original checkers in CDLint.

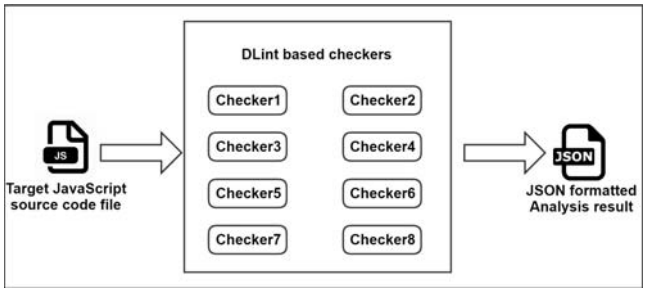
Both of the task distributor and DynChecker are run on a Node.js server. The features of Node.js such as asynchronous and nonblocking can provide excellent performance on task handling and processing.

Figure 3 illustrates the interface of CDLint on web browser. Developers can access this web page and submit either the URL or JavaScript code through it. The analysis result of JavaScript code will also be displayed on web.



(Figure 3) Web interface for developers to submit either URL of target website or JavaScript code

The progress of checking the JavaScript code is handled on the back-end of CDLint system. CDLint will enable the checkers registered in DLint and thoses checkers will analyze the JavaScript code. Then the analysis result will be stored in a JSON file. Figure 4 presents the processing progress of JavaScript code checking mechanism.



(Figure 4) The analysis process in the back-end of CDLint

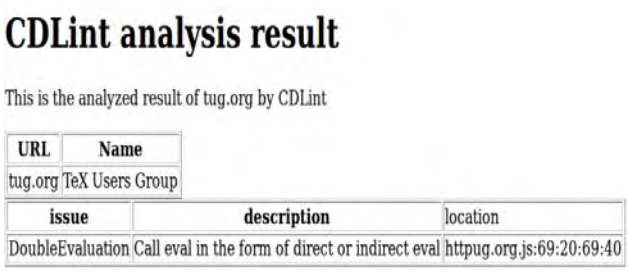
<Table 1> Available checkers supported on CDLint. Additional checkers can be added by uploading the checker file to CDLint.

Checker name	Checking purpose
CheckNaN	Find NaN result operations
ConcatUndefinedToString	Find concatenating string and undefined
NonObjectPrototype	Find prototype setting to non-object
SetFieldOfPrimitive	Find prototype writing to primitive value
OverflowUnderFlow	Find numerical overflows and underflows
StyleMisuse	Find comparison between CSS and string
ToStringGivesNonString	Find no-string-returned from toString()
UndefinedOffset	Find access to undefined property
NoEffectOperation	Find no effect property writing
AddEnumerablePropertyToObject	Find enumerable property to object
FunctionToString	Find Function.toString()
NonNumericArrayProperty	Find non-numeric array property
GlobalThis	Find using 'this' on global object
ForInArray	Find iterating over array with for-in loop
EmptyClassInRegexp	Find empty class in regular expression
ObjectFunctionMisuse	Find misuses on objection functions
UseArrObjConstrWithoutArg	Find no parameter on array and object

Each of the JavaScript code checkers registered in CDLint has three components: the order number to decide the sequence of when this checker will be executed; the main checking function for checking a specific code issue; and the description of this checker to explain it to developers. Table 1 shows a part of the available checkers provided by CDLint.

To verify the usability of CDLint, we used CDLint to test a website named 'tug.org' which is a TeX users group website. Figure 5 demonstrates the analysis result of 'tug.org'. CDLint provides a function to parse the output JSON file and display it on the web browser.

In the analysis result, CDLint firstly displayed the URL and website name which the developer submitted. Then the issue discovered by the checkers in CDLint is presented. The description of this issue is invoked from the checker description which was discussed before. The location of this issue is also pointed out that the developer can locate this issue in the source code file.



(Figure 5) A test analysis result of CDLint for the website called 'tug.org'.

4. Comparison of CDLint and other systems

We compared CDLint with other similar systems: JSLint, JSHint, ESLint, and DLint. The comparison result is presented in Table 2.

<Table 2> The comparison of CDLint with other similar Systems.

System name	Checkers	Extensible	No installation	Auto Web JS parsing	Environment configuration
JSLint	7	X	O	X	O
JSHint	1	X	X	X	X
ESLint	20	X	X	X	O
Dlint	28	O	X	X	O
CDLint	28	O	O	O	O

CDLint shows the best performances on all of the features: number of checkers, system extensibility, use of without installation, automatically parsing JavaScript code from web, and environment configuration for JavaScript code execution. All these features make a remarkable usability of JavaScript code analysis in CDLint.

5. Conclusion

We introduced CDLint, a cloud service for dynamic analysis of JavaScript code. According to the growth of technologies such as Node.js, the uses of JavaScript are extended from web front-ends to back-ends. Since the back-end of web guarantees its system runs healthily, it is becoming more necessary and important to verify the correctness of JavaScript code especially when it executed at the back-end.

CDLint is developed based on DLint which is a dynamic JavaScript code checking system. We deployed CDLint on a cloud so that developers can use our system without installing any of programs to check the violations or vulnerabilities of their JavaScript code. CDLint provides several JavaScript checkers inherits from DLint. Additional checkers also can be registered by developers. We tested the usability of CDLint through checking the JavaScript code runs at a website named ‘tug.org’.

We also compared CDLint with other similar systems including JSLint, JSHint, ESLint, and DLint. The result of comparison shows that CDLint has the best performances concerning on the following criteria: the number of JavaScript code checkers, system extensibility, no installation, auto web JavaScript code parsing, and environment.

In the future, we will keep working on optimizing CDLint to improve the usability by adding a code correction suggestion mechanism. This mechanism is supposed to be able to tell developers how to correct their JavaScript code when detected as a violation by CDLint.

ACKNOWLEDGEMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (NRF-2016R1D1A1B03936453).

*Corresponding author: Gyun Woo (Pusan National University, woogyun@pusan.ac.kr).

References

[1] David Flanagan "JavaScript: The definitive guide: Activate your web pages". O'Reilly Media, Inc.

[2] Mike Cantelon et al. "Node. js in Action". Manning.

[3] Stefan Tilkov and Vinoski Steve "Node. js: Using JavaScript to build high-performance network programs". IEEE Internet Computing.

[4] Douglas Crockford "JavaScript: The Good Parts: The Good Parts". O'Reilly Media, Inc.

[5] Holger M. Reviewer-Kienle "Effective JavaScript: 68 specific ways to harness the power of JavaScript by David Herman". ACM SIGSOFT Software Engineering Notes.

[6] Lawrence Spencer and Seth Richards "Reliable JavaScript: How to Code Safely in the World's Most Dangerous Language". John Wiley & Sons.

[7] Stephen C. Johnson "Lint, a C program checker". Murray Hill: Bell Telephone Laboratories.

[8] Liang Gong et al. "DLint: Dynamically checking bad coding practices in JavaScript". Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM.

[9] Harry Cummings "Learning Node.js for .NET Developers". Packt Publishing Ltd.

[10] Adriano L. Santos, Marco Tulio Valente, and Eduardo Figueiredo "Using JavaScript Static Checkers on GitHub System: A First Evaluation".