Industry Practice of JavaScript Dynamic Analysis on WeChat Mini-Programs

Yi Liu†, Jinhui Xie‡, Jianbo Yang‡, Shiyu Guo‡, Yuetang Deng‡, Shuqing Li†, Yechang Wu†, Yepang Liu†
†Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
‡Tencent, Inc., China

{11610522,lsq2017,11711918}@mail.sustech.edu.cn,liuyp1@sustech.edu.cn
{hugoxie,xiaotuoyang,whiteguo,yuetangdeng}@tencent.com

ABSTRACT
JavaScript is one of the most popular programming languages. WeChat Mini-Program is a large ecosystem of JavaScript applications that runs on the WeChat platform. Millions of Mini-Programs are accessed by WeChat users every week. Consequently, the performance and robustness of Mini-Programs are particularly important. Unfortunately, many Mini-Programs suffer from various defects and performance problems. Dynamic analysis is a useful technique to pinpoint application defects. However, due to the dynamic features of the JavaScript language and the complexity of the runtime environment, dynamic analysis techniques were rarely used to improve the quality of JavaScript applications running on industrial platforms such as WeChat Mini-Program previously. In this work, we report our experience of extending Jalangi, a dynamic analysis framework for JavaScript applications developed by academia, and applying the extended version, named WeJalangi, to diagnose defects in WeChat Mini-Programs. WeJalangi is compatible with existing dynamic analysis tools such as DLint, Smemory, and JITProf. We implemented a null pointer checker on WeJalangi and tested the tool’s usability on 152 open-source Mini-Programs. We also conducted a case study in Tencent by applying WeJalangi on six popular commercial Mini-Programs. In the case study, WeJalangi accurately located six null pointer issues and three of them haven’t been discovered previously. All of the reported defects have been confirmed by developers and testers.

KEYWORDS
JavaScript, Program Analysis

1 INTRODUCTION
JavaScript is becoming more and more popular among developers [10]. The ecosystem of JavaScript is surprisingly active and millions of JavaScript dependencies are downloaded from npm.js [9] every week. However, there is another side regarding the rapid evolution of JavaScript. For example, there emerge lots of new features and revisions of JavaScript (e.g. definitions of modules and classes, promise embedded library, generators, and proxies) [11, 12] in recent years. As a consequence, the testing tools are required to update promptly to be compatible with such changes. WeChat, a popular messenger application with over one billion monthly active users [2], also uses JavaScript for its Mini-Programs [18] and Mini-Games [17], which are essential components of the WeChat ecosystem to bridge users and services [19]. Nowadays, there are millions of active Mini-Programs [15] on the WeChat platform, providing various services to users. For example, during the outbreak of COVID-19, many organizations rely on WeChat Mini-Programs to collect their members’ health information and notify the users to take precautions once a potential danger is discovered. Thus, the robustness and performance of WeChat Mini-Programs become vital for both users and program publishers.

Unfortunately, we noticed through WeChat Mini-Program monitoring system that millions of crashes occurred in these programs every day, which seriously affects users’ experience. Such a huge number of crashes motivated us to apply JavaScript program analysis tools to help with crash diagnosis and we attempted to utilize a popular dynamic analysis tool developed by academia, Jalangi, to analyze Mini-Programs [13]. However, we found that Jalangi cannot be directly applied to WeChat Mini-Programs due to several limitations. Firstly, many mini-programs leverage new language features introduced in ES6 while Jalangi only supports up to ES5. Secondly, Jalangi’s instrumentation significantly increases the size of the Mini-Programs, which could cause serious performance problems in dynamic analysis. Thirdly, WeChat Mini-Program platform is a customized JavaScript runtime while Jalangi is implemented for the standard one. Therefore, Jalangi fails to analyze many Mini-Programs.

In this work, we built a dynamic program analysis tool, named WeJalangi, by modifying Jalangi to pinpoint defects in WeChat Mini-Programs. We make the following contributions:
• We built a scalable JavaScript dynamic analysis framework, named WeJalangi, by extending Jalangi and applied it in industrial contexts. WeJalangi is fully compatible with existing JavaScript analysis tools (e.g. taint analysis [14], JIT profiler [4], code smell detector [5], and memory checker [6]) based on Jalangi [13]. To our best knowledge, there is no existing work reporting experiences of applying dynamic analysis techniques to improve the...
We implemented the framework on top of ASE ’20, September 21–25, 2020, Virtual Event, Australia Yi Liu, et al.

WeJalangi’s testing platform) sends requests to the proxy Jalangi [13], a JavaScript-based dynamic analysis framework, with the client, the callback functions (also known as hooks) in the SUT. Constructor prototype holding to guarantee the functionalities of WeJalangi are executed to detect the defects, which developers are interested for forwards the instrumented SUT. Finally, when executing the SUT in WeJalangi, it instruments the source code of the SUT in the fly and server to retrieve the System Under Test (SUT). As a consequence, WeJalangi instruments the main steps of WeJalangi’s workflow as follows. First, the client (e.g. Mini-Programs’ testing platform) sends requests to the proxy server to retrieve the System Under Test (SUT). As a consequence, WeJalangi instruments the source code of the SUT in the fly and forwards the instrumented SUT. Finally, when executing the SUT in the client, the callback functions (also known as hooks) in WeJalangi are executed to detect the defects, which developers are interested in. Once such defects are found, the context information including call stack and arguments are reported to the log server by WeJalangi for further analysis. Note that WeJalangi only instruments the SUT, and it utilizes several techniques such as scope binding and constructor prototype holding to guarantee the functionalities of the SUT.

We briefly summarize the characteristics of WeJalangi as following:

- **Compatible with ES6 features**: JavaScript standards change rapidly. But the state-of-the-art framework Jalangi [13] only supports ES5. WeJalangi has a full support for ES6, which means it can be directly applied to those modern JavaScript applications with the latest JavaScript language features.

- **Efficient analysis**: WeJalangi utilizes several accelerating techniques including code minimization and short-circuit evaluation. It maintains the minimal runtime for analysis. According to our evaluation, WeJalangi significantly outperforms Jalangi.

- **A robust dynamic runtime**: WeJalangi works for most of the runtimes such as WeChat Mini-Program, Node.js [8], Chrome, etc. For example, we made many modifications to ensure WeJalangi could work normally in the runtime of WeChat Mini-Programs.

Nevertheless, we still strengthened multiple hooks to make it adaptive to different JavaScript runtimes.

3 IMPLEMENTATION

We built WeJalangi on top of Jalangi [13] for industrial use. To our best knowledge, it is the first time that JavaScript dynamic analysis technology has been introduced to industry practice. In this section, we address the challenges and deliver technical solutions for utilizing WeJalangi to analyze modern and large JavaScript applications. Note that WeJalangi could successfully analyze WeChat Mini-Program SDK, which contains more than 100,000 lines of code.

3.1 Instrumentation

Instrumentation is a core step for dynamic analysis. The main idea of instrumentation is to inject various callback functions from the library of WeJalangi into specific nodes on the abstract syntax tree (AST) of the SUT by traversing the AST. As a result, those callback functions will be executed during SUT execution and the runtime information could be tracked and modified in a middle layer (proxy server) between the SUT and the client. In the original implementation of Jalangi, functions will be explicitly hoisted to remain the same scope. We observed some cases, in which the functionality of the SUT was broken due to the mis-hoisting of Jalangi. In our implementation, we remained the context and kept the scope unchanged. Therefore, no specific hoisting is required anymore.

```javascript
function Rt(iid, val, fIid) {
    var aret;
    if (sandbox.analysis & sandbox.analysis._return) {
        aret = sandbox.analysis._return(iid, val, fIid);
        if (aret) {
            val = aret.result;
        }
    } else {
        returnStack.pop();
        returnStack.push(val);
        return (lastComputedValue = val);
    }
}
```

Listing 1: Return statement hook of WeJalangi

After instrumentation, original semantic information is reserved, and more information is passed to callback functions as arguments. For instance, WXJ$Rt( iid, return_value, function id) indicates that it is a return hook. As for each argument, iid records the location id of the instruction; return_value shows the original return value of the instrumented FUT(function under test); function id indicates the identifier of the instrumented

3.2 Supporting Latest Language Features

JavaScript changed a lot since ES6 language features were published [11, 12]. Unfortunately, many research tools couldn’t analyze modern JavaScript directly since they are implemented based on

```javascript
class Foo {
    property = 1;
}
```

Listing 2: Class property in ES6 language features
Table 1: The performance of WeJalangi and Jalangi on a simple test case

<table>
<thead>
<tr>
<th>Number of Loops</th>
<th>WeJalangi (ms)</th>
<th>Jalangi (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>79</td>
<td>269</td>
</tr>
<tr>
<td>10,000</td>
<td>469</td>
<td>2,452</td>
</tr>
<tr>
<td>100,000</td>
<td>4,553</td>
<td>25,082</td>
</tr>
</tbody>
</table>

3.3 Performance Optimization

Performance is another critical issue when adapting WeJalangi into industrial practice. We focus on two different aspects of performance as follows:

3.3.1 The Size of Instrumented SUT. Keeping a small size of an instrumented SUT is essential due to the high cost of network bandwidth and the limitations of computational resources on various devices. A smaller-sized SUT costs lower network traffic and requires less time for JavaScript compilers to parse the source code.

To optimize the size of the instrumented SUT, we utilized the JavaScript code module bundler, Webpack [16]. Webpack is a well-maintained module bundler that many plugins have been made to extend its functionalities. Besides, optimization for specific AST patterns such as the domain-specific language defined in WeChat’s scripts of WeChat Mini-Program could be quickly done by implementing a webpack plugin. On the other hand, we also make WeJalangi not instrument on all AST nodes like Jalangi to further reduce the size of instrumented SUT. For AST nodes like empty statements, field read/write and so on, WeJalangi does not perform instrumentation.

3.3.2 Execution Performance. Execution performance is critical when applying WeJalangi to WeChat Mini-Program. We optimized WeJalangi to only instrument the selected AST nodes as introduced in Section 3.3.1, which also reduces execution time. Moreover, WeJalangi will not insert try-catch blocks for functions as Jalangi does in the instrumentation process, saving efforts made to reserve contexts. As Table 1 shows, WeJalangi with default settings outperforms Jalangi by at least 50%.

3.4 Robust Analysis Runtime

Multiple unknown bugs existed in the original Jalangi implementation. While applying Jalangi to WeChat Mini-Program, we made lots of efforts to fix unknown bugs. For example, symbol is a new primitive type introduced in ES6 language standard. But the implementation of original Jalangi combines symbol variable with an empty string (that is, "") as shown in Listing 3, which is forbidden in ES6 syntax. Thus, a runtime exception is thrown, and the analyzers of runtime crash. To fix this issue, we implemented WeJalangi to check the type of property at first and invoke directly if it is a symbol variable.

Listing 3: Concat symbol primitive with empty string

```javascript
// Throws "Uncaught TypeError: Cannot convert a Symbol value to a string"
return (prop + "" == '__proto__') || CALL.call(HAS_OWN_PROPERTY, obj, prop);
```

Listing 4: Original implementation of calling native constructor in Jalangi

WeChat Mini-Program is a vast JavaScript application ecosystem with millions of users online every day. The runtime of WeChat Mini-Program has been modified to improve user experience, security, and performance. To analyze Mini-Program, we made many modifications to WeJalangi. A typical example is that: eval is a native function used to execute JavaScript code snippets dynamically supported by most of the JavaScript runtimes including V8 and Node.js. For security reasons, it has been removed by WeChat Mini-Program’s runtime. But the invocation process of new operations (creating different instances of functions) in original Jalangi requires the support of eval, as shown in Listing 4. When constructors are called, Jalangi throws an exception with the error message "Uncaught ReferenceError: eval is not defined" and the SUT behaves unexpectedly. To make up for the lack of built-in eval function,
WeJalangi flattens the arguments and calls the constructor directly, which will fix the issue neatly. By doing so, we can finally utilize WeJalangi to analyze WeChat Mini-Programs.

### 4 USABILITY VALIDATION

Null pointer is a common kind of defects in JavaScript applications. We evaluated the crash records of WeChat Mini-Programs from 1st May 2020 to 30th May 2020 and found that null pointer problems took more than 10% of these records every day. On top of WeJalangi, we implemented a null pointer checker for JavaScript applications and used it to test the tool’s usability on 152 randomly sampled open-source Mini-Programs.

After that, we collected six popular and commercial WeChat Mini-Programs (HLDDZ, TYZGXQ, ELSB, MHTCS, TTDDZ, FKDC), which have more than 100,000 lines of code and more than 100,000 active users to perform a deeper evaluation manually. All of the null pointer defects found by our checker for these programs (each program has been found one defect) have been pinpointed and confirmed by developers and testers. Half of these defects were not recognized by original testing tools previously since these exceptions were caught by the framework and errors were thrown in other places, which makes it confusing and hard to find the real fault locations. In summary, WeJalangi could be applied to industry practice and help developers to pinpoint null pointer defects.

#### 4.1 Case Study

```
// gameThirdScriptError
Cannot set property 'active' of null at setTimeout callback
var _temp;
try {
  t = _createGraphics();
  e = _temp;
  e.clear(!1);
} catch (e) {
  _createGraphics();
}

Listing 5: The stacktrace of manually replaying
```

```
// _updateGraphics()

Listing 6: The context information collected in log server
```

```
function _updateGraphics() {
  // fix here
  t = _this._graphics;
  e = _this._graphics;
  e.clear(!1);
}

Listing 7: Fixing for the defect with the help of context information
```

We conducted a case study on ELSB, one of the six selected Mini-Programs introduced above. To boost the analysis process, we gathered the bug report, which developers sent to testers for help when they encountered bugs, as shown in Table 2. It contains WeChat Mini-Program SDK Version, application ID, error message, and replay procedure.²

Following the procedure shown in Figure 2, we manually reproduced the null pointer issue (shown in Listing 5). Once the defect has been reached, the callback functions would provide synthesized context information and send it to the log server. Then, with the pieces of information in the context log, we could easily find the fault location via an instruction ID map generated by WeJalangi (shown in Listing 6). Finally, with the help of context information, we fixed the defect and found the root cause: The game engine does not initialize the graph instance.

#### 5 RELATED WORK

As JavaScript applications become increasingly popular and sophisticated, JavaScript application analysis is becoming more and more challenging. In the last few years, several techniques have been proposed in the literature to achieve automated analysis of JavaScript applications. In this section, we briefly discuss the most notable existing solutions and their limitations, which motivate the need for a practical JavaScript dynamic analysis frameworks.

Dynamic Analysis is a notable paradigm for analyzing JavaScript applications [3]. Existing approaches like Dlint [5], JITProf [4], and Smemory [6] have been widely used to analyze the defects of JavaScript applications. Unfortunately, they are all based on Jalangi [13], which cannot be directly applied to industry practice. To our best knowledge, WeJalangi is the first framework applied to industry practice, and all of the tools mentioned above can be compatible with WeJalangi.

#### 6 CONCLUSION AND FUTURE WORK

In this work, we built WeJalangi, an extensible dynamic analysis framework for JavaScript applications based on Jalangi. We have successfully applied WeJalangi on WeChat Mini-Programs and demonstrated its effectiveness and efficiency. In the future, we plan to further evaluate WeJalangi in industrial settings and build domain-specific automated testing tools based on it.

### ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grants #61932021 and #61802164). The authors would like to thank developers from Tencent, Inc. and Chao Li for providing valuable suggestions and comments for evaluating WeJalangi.

²Some confidential information is not shown in the table.
REFERENCES