Using Jalangi for Automatic Error Detection in JavaScript Games

Andrey Ryzhov
St Cross College
University of Oxford

A dissertation submitted for the degree of
Master of Science in Computer Science
Supervised by Franck van Breugel

Trinity 2014

Acknowledgements

I am grateful to my project supervisor Franck van Breugel for introducing me to this research topic, for guiding my work and significantly helping me with the game localisation process.

I also thank Kaitlin Smith and Owen Lawson of Uken Games for providing us with a great game for our research and for answering our questions.

In addition, I would like to thank Koushik Sen and Manu Sridharan for developing the Jalangi tool, which formed the basis for our research, and assisting us in solving various issues.

My sincere thank goes to my dear parents, without whom, I would not be able to spend this wonderful year at Oxford.

Abstract

Considering that today's computer games are becoming more complex, guaranteeing their correctness is becoming more and more challenging. Numerous game development studios create browser games using HTML and JavaScript, mainly because they can be run on multiple platforms. In this dissertation, we apply an automated framework, called Jalangi, for developing and performing dynamic analyses for the JavaScript language, to the Crime Inc. game provided by the Uken Games company. At first, we present the facilities offered by the tool and review the important technical aspects of writing new or of invoking existing analyses. We further explain the reasons for localising a web application, in particular the Crime Inc. game, to fit the Jalangi requirements. We then describe the process of applying different analyses to the localised version of the game, and explain the results produced from our experiments. The results show various warnings, generated during an execution of the Crime Inc. game, and indicate robustness of the applied analyses.

Contents

Chapter 1

Introduction

According to the statistics provided by RedMonk Programming Language Rankings[1] JavaScript is ahead of all other programming languages[2]. The popularity data is based on the number of projects hosted at GitHub and questions posted at StackOverflow. Figure 1 illustrates the graph obtained by measuring data from the StackOverflow and GitHub sites:

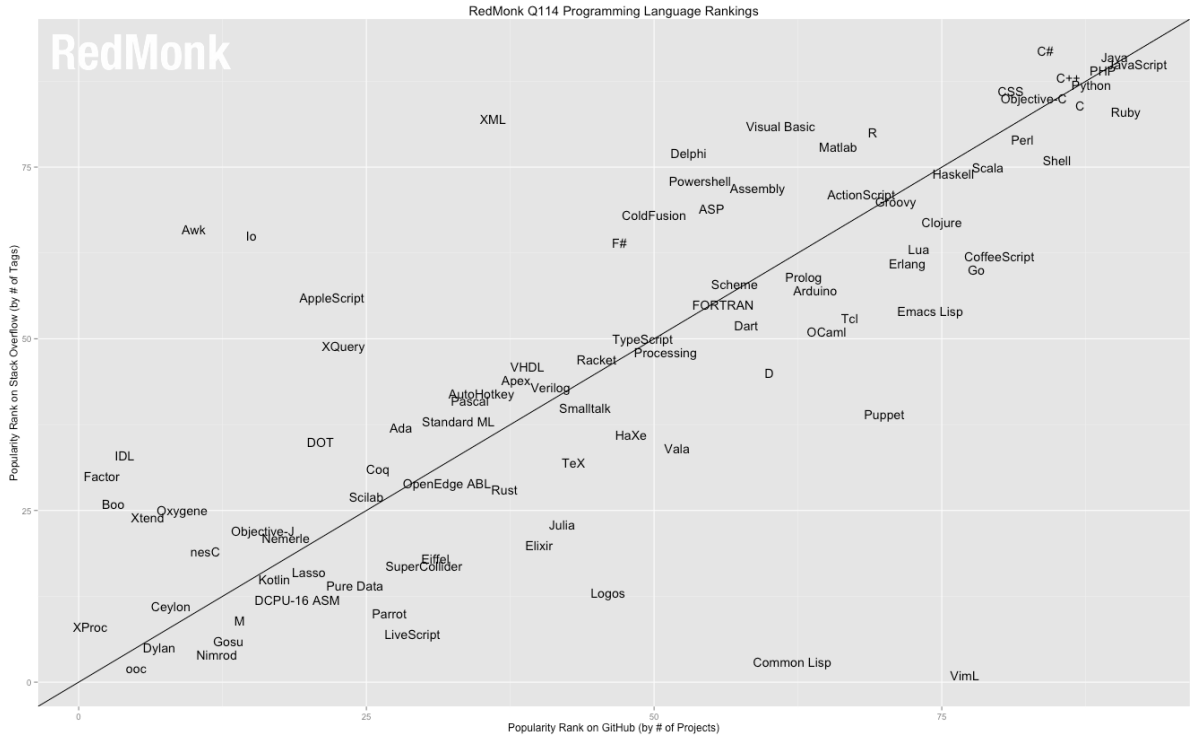

Figure 1.1: Programming Languages Ranking

JavaScript is an object-oriented language developed for simple scripting – extending a web site with client-side executable code [7, 5]. Currently, it plays a substantial role in web appli-

[1] http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/

[2] It might not be clear because of the image resolution, but JavaScript is ahead of Java.

cation development. Its popularity is increasing and it has started to garner academic attention [1, 7]. The key advantage of the JavaScript language is its portability – a single program can be executed on any machine that has a web browser supporting JavaScript. Recent progress in the mobile market and browser technologies have boosted the use of JavaScript in such mobile platforms as Android, iOS, Windows 8, Blackberry, Firefox OS and in Rich Internet Applications [8].

Richards gathered observations that indicate a high popularity of dynamic features of JavaScript that make the language hard to be analysed statically [7]. Unlike many popular programming languages, such as C, C++ or Java, there is a lack of tool support for testing JavaScript code [1, 8]. At the same time, a significant number of game studios develop browser games using JavaScript to enhance the portability of applications, which is one of the most crucial emerging factors due to the enormous variety of platforms [2].

Because we were attracted by the popularity of the JavaScript language and the lack of support for automatic testing, we decided to focus our research on one specific type of applications: games. Initially, we concentrated on finding an optimal tool for analysing JavaScript programs. In particular, we tested our approach on the mobile browser game called Crime Inc., provided by the Uken Games company[3]. To fit the nature of the language, we only considered software that offers dynamic analyses, which are those that take place at an application's run-time. We searched for a framework that would be able to perform the required analyses in the background while the user plays the game. Moreover, for reasons further described in this report, we required a tool that would work in an in-browser mode (Section 2.5.2.2), producing the results immediately during execution. At first, we attempted to apply the tool called Artemis [1], but it did not work, because of the specific JavaScript features (such as *Web Worker*[4]) used in the implementation of the game. We then moved to another tool named Jalangi[5]. By investigating this tool, we found that it allows a user to easily implement new dynamic analyses and run them in a browser window without any issues.

In this report, we describe the Jalangi framework and explain the most essential aspects of writing new dynamic analyses and of using existing ones. We also demonstrate how we applied the tool to the Crime Inc. game and discuss necessary preparations steps. At the end of this paper, we present the analyses that we were able to perform and interpret the obtained results.

[3]http://www.uken.com
[4]http://www.w3schools.com/html/html5_webworkers.asp
[5]https://github.com/SRA-SiliconValley/jalangi

Chapter 2

The Jalangi Tool

Jalangi provides a dynamic analysis framework to allow a user to easily build platform-independent analyses for the JavaScript language. These are the main design decisions that guided the process of development of the Jalangi tool [8]:

- *Independence of browsers and JavaScript version.* The developers aimed to create a tool that is not restricted by a specific JavaScript technology. With this design decision, Jalangi addresses the rapid evolution in web browser technologies and omits the need to refactor or rebuild the framework whenever an underlying browser is updated. Such independence is achieved through a selective record phase described in Section 2.1. One of the key features of Jalangi is an ability to perform an analysis even if some fragments of the source code are not instrumented[1].

- *Independent analysis execution*. Jalangi can perform dynamic analyses on a desktop or a cloud machine. Some dynamic analyses cannot be performed due to insufficient computational resources when working in the online mode[2]. Furthermore, analyses that require access to the file system or other specific resources cannot be launched in the in-browser mode without modifying the browser. This challenge is addressed through the *record-replay* technique described in Section 2.1. The first phase of this technique records and saves the actual execution of an application, whereas the second phase uses a trace of the execution to imitate the recorded behaviour of the program. In this way, the replay phase can be performed in a browser-independent environment, such as a computer desktop.

- *Ease of writing dynamic analyses*. Jalangi supports shadow execution and shadow values (described in Section 2.2) – these features create an opportunity to associate additional

[1] As we will discuss in Section 2.3, Jalangi's instrumentation annotates the JavaScript code with additional information that can be exploited by the analyses.

[2] The online mode provides facilities to analyse an application or record its behaviour directly in a browser window.

information with every JavaScript variable, object or function. Consider the following example of declaring a JavaScript variable:

```
var x = 5;
```

Jalangi allows a user to implement a dynamic analysis that annotates the above statement with shadow information and transforms it into the code below:

```
x = {actual: 5, shadow: "unsigned int"};
```

This replacement also requires modifications in every standard operation (for example, multiplication or boolean operation) such that an updated version first extracts the actual value, and then performs an appropriate operation. The fragment of the source code below represents how Jalangi maintains binary operations:

```
function B(iid, op, left, right) {
    var left_c, right_c, result_c, isArith = false;
    ...
    left_c = getConcrete(left);
    right_c = getConcrete(right);

    switch (op) {
        case "+":
            isArith = true;
            result_c = left_c + right_c;
            break;
        ...
        case "<<":
            isArith = true;
            result_c = left_c << right_c;
            break;
        ...
        case "==":
             result_c = left_c == right_c;
        ...
    }

    ...
    return result_c;
}
```

The full set of supported binary operations is as follows:

```
+, -, *, /, %, <<, >>, >>>, <, >, <=, >=, ==, !=, ===, !==, &, |,  ^,
instanceof, in, &&, ||, regexin
```

Shadow values are propagated along with the execution of a program. For example, consider a declaration of another JavaScript variable:

```
var y = x * 2;
```

Using the previously annotated variable *x* and the binary function *B(iid, op, left, right)*, the above statement is converted into:

```
var y = B(operationID, "*", x, {actual: 2, shadow: "unsigned int"});
```

As the result of the above operation, the *unsigned int* information is propagated to the *y* variable during execution:

```
var y = {actual: 10, shadow: "unsigned int"};
```

One of the major weaknesses of Jalangi is handling native JavaScript code, since this code cannot be instrumented. The following example illustrates the problem:

```
var numbers = [1, 2, 3];
numbers.pop();
```

The array *numbers* can be instrumented, whereas the function *pop()* cannot be, as it is a native function. Since Jalangi generates an *"Analysis exception"* error whenever an analysis executes native code, the *selective record* technique helps to avoid that problem by instrumenting only user-specified parts of the code, skipping any native JavaScript by default. In this way, the recorded execution of a program contains only instrumented code, preventing Jalangi from executing native JavaScript.

The technical implementation of the Jalangi framework is based on two key ideas, further discussed in this chapter: the *selective record-replay* technique and *shadow values* and *shadow execution* [8].

2.1 Selective Record-Replay

Given an application, the *record-replay* technique allows a user (1) to select parts of an application by specifying the source code directory containing those parts, (2) record the program's behaviour, and (3) replay the recorded execution [6]. The latter can be replayed on a platform different from the one it was recorded on. For example, if the user records an application's execution in a web browser, he can later replay it on a desktop platform. There are two phases that form the record-replay technique:

- During the *recording* phase the application is run in a user-specified platform (such as web browser) where the entire program's source code is executed. This stage includes instrumentation and recording of the specified parts of the program.

- During the *replay* phase Jalangi only reproduces those parts of an application that were initially instrumented. While a user records a behaviour of the program on the actual web platform, he can perform an analysis in the replay phase on a desktop or cloud machine, using system resources that are not available in an alternative, browser-implemented solution. Furthermore, an analysis, which purely depends on the instrumented code, provides wider opportunities for applying various dynamic analyses.

There are two key advantages obtained by the division of the record-replay technique into two phases:

- The execution of the JavaScript application can be recorded on an actual platform and replayed on a desktop machine that supports a JavaScript engine (Jalangi uses the *Node.js* framework to enable this technique). Furthermore, it provides an analysis writer with an access to substantially larger computational resources for performing expensive analyses.
- The user can implement a number of dynamic analyses that are built upon the shadow execution technique, assuming that un-instrumented and native code is ignored in the replay phase.

The common way to perform record-replay of an application is to collect and save every memory variable during the record phase and use them in the replay phase to reconstruct the program's behaviour. This approach has two problems associated with it: (1) How to record values of JavaScript objects and functions? (2) How to replay an execution when an instrumented function is called by un-instrumented or native code? Sen *et. al* solved the first problem by assigning a unique numerical identifier to each JavaScript function and object and recording values of those identifiers. The second issue is resolved by an explicit recording and calling of instrumented functions that are invoked by un-instrumented segments of code or dispatched by the JavaScript events dispatcher.

Jalangi provides the following optimization in the record-replay technique: a value of memory load is not recorded if it can be computed during the replay phase by solely executing instrumented code. Jalangi maintains a shadow memory in order to determine which values need to be recorded. While shadow memory is updated along with the actual memory during the execution of instrumented code, it (shadow memory) remains the same when the un-instrumented or native code is executed. During the record phase such convention allows Jalangi to track the difference between the value of the actual memory load and the corresponding shadow memory value and, if such a difference is found, record that memory load. This ensures faithful loading of memory values during the replay phase [8].

2.2 Shadow Values and Shadow Execution

Shadow values allow to associate additional information with any value used in a dynamic analysis. This information depends on the type of the analysis and can contain useful details about a program value, such as symbolic representation. Shadow execution happens in parallel with the actual execution and propagates shadow values until the application terminates [4].

In Jalangi, shadow execution is performed only during the *replay* phase, where only instrumented code is invoked. Therefore, any possible execution of native JavaScript code, which might generate exceptions, is avoided. The code below represents a fragment of the JavaScript implementation of the *ConcolicValue* object that annotates program values:

```
function ConcolicValue (concrete, symbolic) {
    this.concrete = concrete;
    this.symbolic = symbolic;
}

ConcolicValue.getConcrete = function (val) {
    if (val instanceof ConcolicValue) {
        return val.concrete;
    } else {
        return val;
    }
}

ConcolicValue.getSymbolic = function (val) {
    if (val instanceof ConcolicValue) {
        return val.symbolic;
    } else {
        return undefined;
    }
}
```

The above code is used in the Jalangi analyses to wrap a program value into the *ConcolicValue* object. For example, a boolean variable can be transformed into $\{$*concrete: true, symbolic: "x1 - 100* $>$ *0"*$\}$.

Shadow execution enables a user to associate any program value with a shadow value carrying additional information. The example above illustrates one of such annotated values. The object called *ConcolicValue* contains two fields: the field *concrete* that refers to an actual variable value and the field *symbolic* that denotes a symbolic representation. This technique is called *concolic wrapping* – an association of symbolic expression with a program value. It is used in different Jalangi analyses that depend on symbolic representation: tracking of undefined and null values, taint analysis, likely type inference analysis, coverage analysis and others. A more detailed description of concolic testing is given in Section 2.5.1.1.

2.3 Instrumentation

The instrumentation phase declares a global variable *sandbox* that contains Jalangi's settings and constants. One of the main properties of this variable is the field *sandbox.analysis* – a user defined object that is invoked during the replay phase or during the in-browser analysis. When Jalangi instruments a program's source code it inserts callbacks to methods defined in the Jalangi *analysis.js* module. When we were writing our own dynamic analysis (Section 3.2.3), implementation of callbacks was an essential part of the development. The code below illustrates a simple analysis that we applied to the Crime Inc. game (Section 4.2) to detect an access to a non-existing property of an object:

```
(function (sandbox) {

    function MyAnalysis () {

        var iidToLocation = sandbox.iidToLocation;
        var info = {};

        this.getFieldPre = function(iid, base, offset) {
            if (offset === undefined) info[iid] = (info[iid] | 0) + 1;
        };

        this.putFieldPre = function(iid, base, offset, val) {
            if (offset === undefined) info[iid] = (info[iid] | 0) + 1;
        };

        this.endExecution = function() {
            sandbox.Utils.printInfo(info, function(x) {
                console.log("Accessed property 'undefined' at " +
                iidToLocation(x.iid) + " "+ x.count + " time(s).");
            });
        };
    }

    sandbox.analysis = new MyAnalysis();
})(J$);
```

The main functions *getFieldPre(iid, base, offset)* and *putFieldPre(iid, base, offset, val)* track the cases where a particular property does not belong to the referenced object and record the error in the *info* variable[3]. The *iid* argument is a unique numerical identifier assigned to every program value during the instrumentation phase. The *base* parameter refers to an object that gets or sets the property. The third argument, *offset*, denotes the name of the property that is being called. The last parameter in the *putFieldPre()* method, *val*, contains the new value that is assigned to the *base[offset]* field – this parameter is not used in the example analysis. When the *endExecution()* function is invoked, the analysis reports all locations recorded in the *info* object.

[3] *(info[iid] | 0)* statement in the example code returns zero, if the *info[iid]* property was not initialised before.

The actual range of callbacks supported by Jalangi is much wider. Initially, the instrumentation phase annotates all the source code statements with intermediate functions which then invoke the appropriate callbacks. Table 2.1 represents the available annotation prefixes, their meaning and relation to the intermediate functions:

Prefix	**Meaning**	**Intermediate function**
J$.Fe	Entering a function	Fe(iid,val,dis,args)
J$.Fr	Function return	Fr(iid)
J$.M	Method call	M(iid, base, offset, isConstructor)
J$.A	Assign value	A(iid, base, offset, op)
J$.P	Put object's field	P(iid, base, offset, val)
J$.G	Get object's field	G(iid, base, offset, norr)
J$.Se	Script enter	Se(iid, val)
J$.Sr	Script exit	Sr(iid)
J$.R	Variable read	R(iid, name, val, isGlobal, isPseudoGlobal)
J$.W	Variable write	W(iid, name, val, lhs, isGlobal, isPseudoGlobal)
J$.T	Object literal	T(iid, val, type, hasGetterSetter)
J$.N	Variable declaration	N(iid, name, val, isArgumentSync, isLocalSync)
J$.B	Binary operation	B(iid, op, left, right)
J$.U	Unary operation	U(iid, op, left)
J$.C	Conditional statement	C(iid, left)
J$.C2	Case label in switch	C2(iid, left)

Table 2.1: Intermediate methods in the *analysis.js*

Prefix	**Method in sandbox.analysis**
J$.Fe	functionEnter(iid,val,dis,args)
J$.Fr	functionExit(iid)
J$.M	invokeFunPre(iid, f, base, args, isConstructor)
J$.A	putFieldPre(iid, base, offset, val)
J$.P	putFieldPre(iid, base, offset, val)
J$.G	getFieldPre(iid, base, offset)
J$.Se	scriptEnter(iid, val)
J$.Sr	scriptExit(iid)
J$.R	readPre(iid, name, val, isGlobal)
J$.W	writePre(iid, name, val, lhs)
J$.T	literalPre(iid, val, hasGetterSetter)
J$.N	declare(iid, name, val, isArgumentSync)
J$.B	binaryPre(iid, op, left, right)
J$.U	unaryPre(iid, op, left)
J$.C	conditionalPre(iid, left)
J$.C2	conditionalPre(iid, left)

Table 2.2: Callback methods in the *sandbox.analysis*

Every intermediate function listed in Table 2.1 invokes a callback method implemented in the *sandbox.analysis* variable. As a result, every instrumentation prefix has an associated callback method invoked from the appropriate intermediate function. Table 2.2 represents the correlation between Jalangi prefixes and the resulting callback methods declared in *sandbox.analysis*.

Notice that for the same prefix, methods from Table 2.1 and callbacks from Table 2.2 may have different number of input parameters. For example, prefix *J$.M* has the intermediate function *W(iid, name, val, lhs, isGlobal, isPseudoGlobal)* taking six arguments, whereas the corresponding callback *writePre(iid, name, val, lhs)* requires only four. This happens because intermediate functions can absorb or produce new variables in the process of invoking the related callback. For example, the *W()* function uses the *isGlobal* and *isPseudoGlobal* variables only to perform additional operations in the record-replay mode, but does not transmit these variables further to the *writePre()* method.

In this way, whenever a JavaScript statement is executed in the replay phase or during the in-browser analysis, the corresponding method in *sandbox.analysis* is called to perform the specified tests. The following example is a statement from the Annex game provided in the Jalangi distribution:

```
board[place[0]][place[1]] = color;
```

The instrumentation phase converts the above code into the following Jalangi statement:

```
J$.P(16513, J$.G(16473, J$.R(16441, 'board', board, false, false), J$.G(
16465, J$.R(16449, 'place', place, false, false), J$.T(16457, 0, 22, false)
, false), false), J$.G(16497, J$.R(16481, 'place', place, false, false), J$
.T(16489, 1, 22, false), false), J$.R(16505, 'color', color, false, false))
```

It starts with the *J$.P* prefix that refers to the property assigning callback – *putFieldPre(iid, base, offset, val)*. The first argument indicates an *iid* of the JavaScript value – a unique number identifying the program objects during the analysis. The second argument refers to the base object *board[place[0]]* that owns the property – *J$.G(16473, J$.R(16441, ...), J$.G(16465, ...), false)*. This argument invokes another Jalangi *"getField"* prefix *J$.G(16465, J$.R(16449, ...), J$.T(16457, ...), false)* that denotes a field access to *place[0]*. The third parameter *offset* presents the property of the object that is being modified – *J$.G(16497, J$.R(16481, ...), J$.T(16489, ...), false)* that refers to *place[1]*. And the last argument, *val*, carries the new value of the object's field – *J$.R(16505, 'color', color, false, false)*.

Table 2.2 only contains methods with *-pre* suffix, such as *putFieldPre()* or *binaryPre()*. This suffix indicates that a callback is invoked before an actual code is executed. For example, *putFieldPre()* is called before the object's property is read. Jalangi extends its capabilities and allows to modify a program's behaviour by changing a return value in callbacks with *-pre* suffix. We used this technique to implement an HTTP redirection feature described in Section 3.3.3. However, it is not mandatory to include return statements in these functions, as demonstrated in

the example analysis at the beginning of Section 2.3. Jalangi also provides alternative versions for the callbacks with *-pre* suffix:

```
invokeFun(iid, f, base, args, isConstructor),
getField(iid, base, offset, val), putField(iid, base, offset, val),
literal(iid, val, hasGetterSetter), read(iid, name, val, isGlobal),
write(iid, name, val, lhs), binary(iid, op, left, right, result_c),
unary(iid, op, left, result_c), conditional(iid, left, ret)
```

The above listed methods allow a programmer to operate with values after an actual program's code was executed. Therefore, these functions do not affect an application's behaviour and also might have different number of arguments than their *-pre* counterparts, as an implementation of a callback might depend on the priority (order) of its execution.

2.4 Installation and Requirements

Since we performed all our tests and experiments on Mac OS X v.10.7.5, further description is related only to this operating system.

Jalangi is a free open-source project and available at `https://github.com/SRA-SiliconValley/jalangi`. It also requires the following frameworks and libraries to be installed on the machine:

- *Node.js* – an asynchronous framework for easily building network applications, including web servers. This framework is essential for applying the record and replay technique to a web application and performing in-browser analysis in Jalangi. It is available for download at `http://nodejs.org/download/`. The downloaded package can be installed via the standard Mac OS Installer.

- *Sun's JDK 1.6* or higher – this library is a part of the Java Standard Edition Development Kit and can be downloaded at `https://edelivery.oracle.com/otn-pub/java/jdk/8u11-b12/jdk-8u11-macosx-x64.dmg`. The standard Mac OS Installer is sufficient to launch the downloaded disk image.

- *GMP* library – an open source library for performing precise arithmetical operations on rational numbers, floating-point numbers, and signed integers. The library is required by Jalangi's concolic analysis that uses *cvc3*[4] and *automaton.jar* to solve constraints. The GMP library package can be downloaded at `https://gmplib.org/#download`. Once downloaded, the package needs to be unzipped (we used Mac OS Archive Utility). This will create a folder named *gmp-ver* in the current directory, where *ver* is the version of the library. Using the Terminal application, a user should navigate to the newly created folder and type the following instructions sequentially:

[4] http://www.cs.nyu.edu/acsys/cvc3/

```
./configure
make
make check
make install
```

The above commands configure the installation settings, check for all the required libraries and install the GMP library into the */usr/local/* directory. Note that Jalangi automatically reports any missing or incorrectly installed libraries during the installation process.

- *Python 2.7* or higher – a programming language used to install Jalangi and run analyses in the offline mode. Available for download at `https://www.python.org/download/releases/2.7.8/` and can be installed using the standard Installer application.

- *Git* – a source code management system that is needed to download Jalangi and available at `http://git-scm.com/download/mac`. This program is not essential, since the Jalangi archive can be downloaded manually. However, it is recommended to use Git, as the framework is still under development and is constantly updated. Git allows a user to update the local repository using the *git pull* command to keep the current version of Jalangi up-to-date.

Once all of the above-listed libraries and tools are properly installed, Jalangi can be copied to the machine using the following command in the Mac OS Terminal: *git clone https://github.com/SRA-SiliconValley/jalangi*. The latest version of the Jalangi tool will be downloaded to the current directory. The next step is to use Terminal to navigate into the downloaded folder and start the installation process using the command below:

```
python ./scripts/install.py
```

Successful installation of the tool produces the following output:

```
---> Installation successful.
---> run 'npm test' to make sure all tests pass
```

The final step is to run the Jalangi tests:

```
npm test
```

The *node_test* package within the Jalangi distribution includes all tests required before launching Jalangi: test cases for the *Node.js* framework, tests for the current version of Java and tests for checking the robustness of the Jalangi modules and analyses. The testing phase might take up to an hour to run entirely, depending on a speed of the machine. Initially, three tests failed to pass, so we reported that issue to the developers of Jalangi and they updated the version for us. Since then, every test passed successfully and we did not experience any other problems

with installing the software. The list below summarises the full set of technologies we used to run Jalangi analyses:

```
64-bit Mac OS X v.10.7.5, Node.js v.0.10.30, Java v.1.8.0_05,
GMP library v.6.0.0, Python v.2.7.1, Git v.1.7.12
```

2.5 Jalangi Analyses

The Jalangi framework provides two different modes of analysis: offline and online. Note that prior to perform any testing, a Mac OS user should navigate to the folder containing Jalangi using the Terminal application.

2.5.1 The Offline Mode

The offline mode is the mode in which analyses are performed on a program units without any user interaction with the program. The main downside of this mode is that it only works with *unit testing* – a technique where a program is decomposed into several units such that each unit contains a collection of JavaScript functions. A unit also specifies inputs to the program and is run independently of other units [9]. To add the support for the unit testing in a *.js* (JavaScript) file a user has to add the following code at the beginning of that file:

```
if (typeof window === "undefined") {
    require('../../src/js/InputManager');
    require(process.cwd()+'/inputs');
}
```

This code supports the tested unit with the Jalangi libraries to automatically generate input at the specified locations, if the unit is not executed in a browser environment. To indicate an input entry, the user has to insert the following statement at the desired location:

```
J$.readInput(arg);
```

An example unit with the input generation is provided in Section 2.5.1.1.

The offline mode relies on two key techniques described earlier in Section 2.1 and 2.2. The record-replay technique enables a tool to instrument user-selected fragments of the application, whereas shadow execution and shadow values allow to attach additional information to program variables.

Initially, when the Crime Inc. game is localised, as described in Chapter 3, all source code is extracted from HTML pages into multiple JavaScript files which are linked to each other within the corresponding HTML. As the JavaScript language has no support for external linking to other *.js* files [3], the extracted JavaScript cannot be divided into isolated modules for performing unit testing. For this reason, we were unable to apply offline analyses in our research.

2.5.1.1 Concolic Testing

Concolic testing (the term is formed from *concrete* and *symbolic*) is a variation of offline analysis that performs symbolic execution along with the concrete execution of a program, generates the logical constraints on the input values, and solves these constraints (using the GMP library) to generate new input values. The generated input is further used by the concolic engine to investigate previously unexplored execution paths of a program [8, 9]. We demonstrate the application of the concolic testing to the JavaScript unit *sample.js*:

```
if (typeof window === "undefined") {
    require('../InputManager');
    require(process.cwd()+'/inputs');
}

var x = J$.readInput(0);
function foo() {
    if (x > 5) {
        if (x > 8) {
            console.log("x > 8");
        }
        else {
            console.log("x > 5 and x <= 8");
        }
    }
}

foo();
```

The code above is prepared for the offline mode as explained earlier in Section 2.5.1. The following command initialises the automated input generation:

```
python scripts/jalangi.py concolic -i 10 sample
```

The parameter *-i 10* specifies the upper bound for the total number of generated inputs, which are saved to the *jalangi_tmp* folder. The concolic analysis produced two sets of inputs for the example unit:

```
jalangi_inputs1.js: J$.setInput("x1",6,[]);
jalangi_inputs2.js: J$.setInput("x1",9,[]);
```

The above inputs specify the values for the *x1* argument that refers to the *x* variable from the source code. The generated values are 6 and 9, respectively. To execute the *sample.js* unit applying these inputs, a user has to invoke the command:

```
python scripts/jalangi.py rerunall sample
```

The concolic analysis uses the generated input to imitate every possible execution and produces the following results:

```
Running sample on jalangi_inputs1.js
x > 5 and x <= 8
Running sample on jalangi_inputs2.js
x > 8
```

Figure 2.1 represents the corresponding execution tree with generated logical constraints denoted on the edges:

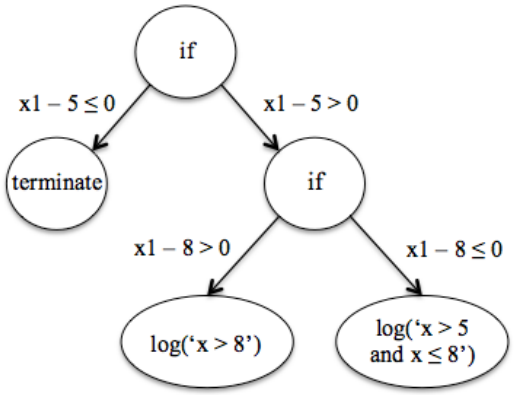

Figure 2.1: Execution tree of the *sample.js* unit

As was outlined in Section 2.5.1, the Crime Inc. source code cannot be decomposed into single units, as all source *.js* files are linked between each other. Moreover, Jalangi's concolic testing cannot be used in the online mode, and therefore we could not apply it to the game.

2.5.1.2 Dynamic Analyses

This section presents a full list of dynamic analyses provided by the Jalangi framework. Even though, these analyses work in the offline mode, all of them can be used in the record-replay analysis of a web application (Section 2.5.2.1) and the majority of them have support or an alternative version for the in-browser mode (Section 2.5.2.2). The table below gives a brief overview including the analysis name, its executable path and support for the in-browser mode:

Name	Executable Path	In-browser
Call Graph	callgraph/CallGraphEngine.js	Yes
Coverage	coverage/CoverageEngine.js	No
Shadow Property	dlint/ShadowProtoProperty	Yes
Undefined Offset	dlint/UndefinedOffset.js	Yes
Eval Analysis	evalusage/EvalUsageAnalysisEngine.js	Yes
LikelyType Inference	likelytype/LikelyTypeInferEngine.js	Yes
NaN Analysis	logNaN/logNaN.js	Yes
Object Allocation	objectalloc/ObjectAllocationTrackerEngine.js	Yes
Object Index	objectindex/ObjectIndex.js	Yes
Taint Analysis	simpletaint/SimpleTaintEngine.js	No
Track Values	trackallvalues/TrackValuesEngine.js	No
Track Undefined	tracknull/UndefinedNullTrackingEngine.js	No
Chained Analysis	ChainedAnalysis.js	No

Table 2.3: Jalangi dynamic analyses

To perform unit testing using any of the analyses listed above in the offline mode, a user has to issue the following command in the Terminal application:

```
python scripts/jalangi.py analyze -a src/js/analyses/path unitFile
```

Where *path* has to be replaced with a concrete analysis path.

The analyses listed in Table 2.3 could not be applied to the Crime Inc. game in the offline mode, for the same reasons explained in Section 2.5.1. However, we used alternative versions of the analyses that have support for the in-browser mode to perform our experiments described in Chapter 4.

2.5.2 The Online Mode

This mode was our main selection for the analysis of the Crime Inc. game. It incorporates support for both, direct and in-direct, web application analyses. On the one hand, the in-direct variation assumes that a user initially records the execution of an application in a web browser and then uses the recorded behaviour to perform dynamic analyses summarised in Table 2.3. On the other hand, the direct version, also called in-browser analysis, can be performed in the real-time web browser environment. This mode allows an analysis to use shadow memory on-the-fly to track bugs or collect statistics along with the actual execution of an application.

2.5.2.1 Record-Replay a Web Application

This analysis requires a manual user interaction with an application to record its behaviour. The rest of this section illustrates an example of a record-replay analysis on the Crime Inc. game.

The user starts with navigating to the Jalangi folder using the Mac OS Terminal. The instrumentation of an application is initialised by issuing the following command:

```
node src/js/commands/instrument.js --outputDir /tmp tests/uken/game
```

It creates an instrumented version of the game in the */tmp/* directory on the user's machine. Jalangi only instruments JavaScript files found in the specified directory (in this example – *tests/uken/game*) and the related sub-directories. Every *.js* file is instrumented in accordance with the prefix rules summarised in Table 2.1. Before starting to play the game, the user has to ensure that no living processes exist in *Node.js*:

```
killall node
```

Then the user should be able to launch the application on the record-replay server called *rrserver*. Note that the application will be opened in the default browser specified by the operating system. For this example we used Google Chrome v.36.0 in combination with the standard Chrome iPhone simulator, which is needed to run the Crime Inc. game (it only has the mobile version). The command below initialises the recording phase of the instrumented version:

```
python scripts/jalangi.py rrserver file://localhost/tmp/game/home.html
```

Figure 2.2: Instrumented home page of the Crime Inc. game

After the browser window has opened, the user can start performing actions on the displayed HTML page (Figure 2.2). To complete the tracing the user has to press *Shift+Alt+T* combination on the keyboard: the JavaScript alert message "*trace flush complete*" should appear in the browser window, indicating that the recording phase has terminated. This generates the *jalangi_traceX* file in the current Terminal directory, where *X* is a serial number based on the count of the previously generated trace files. Before performing an analysis, Jalangi requires the trace file to be in the same location, as the instrumented version of the game. The following command copies the trace file to the *tmp/game* directory:

```
cp jalangi_trace1 /tmp/game
```

Afterwards, we can perform for example the *ObjectAllocationTrackerEngine.js* analysis on the trace file by issuing the instruction below:

```
node src/js/commands/replay.js --tracefile /tmp/game/jalangi_trace1
--analysis src/js/analyses/objectalloc/ObjectAllocationTrackerEngine
```

This command launches the replay phase, applying the object allocation analysis to the behaviour recorded in the *jalangi_trace1* file. Below is a fragment of the log produced by the analysis (the actual log is too large to submit it in the report):

```
Location (js0.js:2688:118) has created object 330 times
      with max last access time since creation = 0 instructions
      and average last access time since creation = 0 instructions
      and seems to be Read Only
Location (js0.js:1447:32) has created object 180 times
      with max last access time since creation = 274617 instructions
      and average last access time since creation = 2528.00555 instructions
```

The output represents the numbers of objects created in the period of the recorded execution and the frequency of accessing these objects. For example, the statement at line *(js0.js:1447:32)* created 180 objects:

```
return new r.fn.init(e, t);
```

Value *274617* from the log indicates the maximum number of program instructions[5] that were executed between the last two accesses of a single object created at location *(js0.js:1447:32)*. Value *2528.055* denotes the average number of instructions that were invoked between the last two accesses of a single object, among all objects that were created at line *(js0.js:1447:32)*. The object allocation tracker analysis might be useful in identifying inefficient memory usage of a specific function or a whole application.

The major benefit of a record-replay of a web application is its ability to analyse the generated trace file independently of a browser or operating system. In this way, an analysis, checking the recorded behaviour, has significantly larger computational resources than the in-browser alternative. Such analysis can also access the machine's file system, whereas the in-browser mode does not support this feature. Furthermore, a user can perform various dynamic analyses on the same trace file without a need to re record the application's behaviour.

As explained in Chapter 3, in order to play the instrumented version of the game, we have developed an analysis that redirects original HTTP requests to our own server. Incompatible with this design decision, the record-replay variation of the online mode does not allow to inject additional analyses while recording the behaviour of a web application. Therefore, in this mode, we could not embed HTTP redirection in the instrumented version of the game and were only able to analyse the *home.html* page without a possibility to navigate further in the application.

[5] An instruction is a single call of a method from Table 2.2, excluding *functionEnter()*, *functionExit*, *scriptEnter()* and *scriptExit()*.

2.5.2.2 In-Browser Analysis

This type of analysis played a key role in our research, as we applied it not only to find bugs in the Crime Inc. game, but to also localise it (Chapter 3). From a user's perspective, an in-browser analysis is simple to use in comparison to others. The user only needs to specify the analysis and the tested application:

```
node src/js/commands/instrument.js --inbrowser --smemory --analysis
src/js/analyses/logNaN/LogNaN.js --outputDir /tmp tests/uken/game
```

The above command instruments the Crime Inc. game, enables the in-browser mode, allows an analysis to use the shadow memory and attaches the *LogNaN.js* analysis. As before, the instrumented game is copied into the */tmp/* directory. The user can start executing the application along with the associated analysis by launching the instrumented version:

```
open file://localhost/tmp/game/home.html
```

Some of the analyses from Table 2.3 support both, offline and in-browser, modes by utilising the *sandbox.Constants.isBrowser* variable that denotes the current mode. For example, the fragment of code below illustrates the modification made to the *LogNaN.js* analysis to expand it for the in-browser mode:

```
if (sandbox.Constants.isBrowser) {
    window.addEventListener('keydown', function (e) {
        if (e.altKey && e.shiftKey && e.keyCode === 84) {
            sandbox.analysis.endExecution();
        }
    });
}
```

The code attaches the specific event listener to a browser window, if an analysis is performed in a browser environment. This listener awaits for the control keyboard sequence *Shift+Alt+T* and invokes the *endExecution()* method that has to be declared in the *sandbox.analysis* variable. In the case of the *LogNaN.js* analysis, *endExecution()* prints all occurrences of not-a-number (NaN) values detected during the execution of an application.

The remaining part of the report demonstrates the reasons for using the in-browser mode (such as HTTP requests redirection) and provides more low-level details in combination with our research results.

Chapter 3

Localising the Crime Inc. Game

We focused our research on the development of general and systematic approach suitable for analysing any JavaScript game. As Jalangi only allows to instrument source code located on a local machine, any web application, initially, needs to be downloaded from a remote server. To accomplish this task, we created the *Localizer* tool that downloads the specified HTML page, extracts all JavaScript code and saves it into separate files. In order to achieve a complete localisation of the Crime Inc. game, we implemented our own dynamic analysis named *Fly.js* (Section 3.2.3). Furthermore, in the process of development, we discovered the need to set up a local server on our machine (Section 3.3.1) and add an HTTP redirection feature to the *Fly.js* analysis (Section 3.3.3).

3.1 Localising the Home Page

We developed the Localizer tool (Appendix A) that performs two actions with the URL provided to it: downloads the linked HTML file and decomposes it. In the context of this research, HTML decomposition is a process of downloading source images, external CSS and JavaScript files into the folder located on our machine, in combination with extracting embedded JavaScript and CSS code from the target HTML page. To enhance the convenience of using the Localizer tool, it is accompanied with a *game.loc* file that contains settings. The below example is a set of parameters that we used to localise the *home.html* page from the Crime Inc. game:

```
url = http://m.staging.crimeinc.uken.com/?udid=v&app_launch=1&
app_version=2.00&ios_bundle_id=com.uken.crimeinc&language=en&
system_version=9
user.agent = iphone4
directory = /Library/WebServer/jalangi/tests/uken/game/
cookies = JSESSIONID="779ce000afb0e1dc";
```

The above properties specify the target URL, the user agent[1], the directory in which the downloaded and decomposed files are saved, and the request cookies. Setting up the user agent is critical if Localizer attempts to download a page from a mobile version of a web application, as it was in case of the Crime Inc. game. The cookies contain specific information without which a download request can be redirected or blocked. We copied the cookies from a browser log by accessing the *home.html* page in the original version of the game.

The main Localizer's work flow consists of several stages that process, extract and modify the content of the downloaded HTML file:

- *Set up a connection* – reads properties from the *game.loc* file.
- *Download HTML* – downloads the target HTML page using the URL specified in the *game.loc* file. This step also creates the Java *Document* object and parses the content of the downloaded HTML file into that object, using the *Jsoup* library[2]. All further modifications of the HTML page (such as code extraction) are only applied to the newly created Document object.
- *Process links to external JavaScript files* – searches for external JavaScript links within the source HTML page and downloads them to the specified directory. The links to these JavaScript files are rewritten within the Document object, so that new links refer to the *localhost* directory – the root directory of our local server, as explained in Section 3.3.1.
- *Process embedded JavaScript* – this stage extracts the JavaScript code embedded into the target HTML page. When a particular fragment of code is extracted into an external *.js* file, it is replaced by a link to that file. Such a link is also attached to the *localhost* directory.
- *Process links to external CSS* – downloads all CSS files to the specified directory. Since a CSS file might contain links to other web application resources, its content is parsed, extracting any external links to JavaScript, HTML, images, fonts or other CSS. For every found link, the corresponding file is downloaded and the related URL is properly modified.
- *Process all embedded CSS* – this step works in almost the same way as the previous one. The only difference is a creation of a CSS file: it is extracted from the original HTML page and saved into the specified directory. Extracted CSS files are also parsed to process any external links, as described in the previous step.

[1]The user agent is part of an HTTP request that provides a web application with information about the device that sends the request.

[2]`http://jsoup.org`

- *Process image links* – identifies all image links found in the target HTML page and downloads them locally. In comparison to a *.js* file, an image does not need to be linked to the *localhost* address, because it is not processed by an *ajax()* request (see Section 3.3.1). However, the image file has to be saved into the appropriate directory. For instance, consider the image URL derived from the *home.html* page:

  `/d2tqjmbrfd3d8u.cloudfront.net/assets/static/chevron_retina.png`

  This image is saved locally to `dir/d2tqjmbrfd3d8u.cloudfront.net/assets/static/`, where *dir* is the directory specified in the *game.loc* file. In this way, the image can further be accessed by a localised version of the game using the `file://dir/d2tqjmbrfd3d8u.cloudfront.net/assets/static/chevron_retina.png` URL. In other words, a localised HTML page is able to access linked images using the machine's file system.

- *Saving the document* – the last step in the localisation process is to save the Document object. The file saved is the modified version of the original HTML page, where all JavaScript and CSS code is completely extracted and the associated links are properly modified.

To use the Localizer tool, a user needs to navigate to the folder containing the *Localizer.java* file using the Terminal application and type in the following instruction:

```
javac Localize.java -cp jsoup-1.7.3.jar:commons-io-2.4.jar:.
```

It compiles the specified Java file, using two external JAR libraries that are located in the Localizer folder. Once the class is compiled, the user can launch the tool by issuing the command below:

```
java -cp jsoup-1.7.3.jar:commons-io-2.4.jar:. Localize game.loc
```

This instruction applies the settings from the *game.loc* file and downloads the requested HTML into the specified directory.

Applying the Localizer tool on the Crime Inc. game provided us with an exact copy of the original *home.html* page. The resulting folder contained 16 separate JavaScript files named *js0.js, ..., js15.js*, where *js0.js* is an external iPhone jQuery library[3]. We also downloaded the *css0.css* file and two HTML files: *home.html* – the main page, and *html1.html* – the embedded HTML file extracted from the *css0.css* file. Initially, *css0.css* contained some outdated links to images which our Localizer tool detected. We reported these outdated links to the developers of the game. After the required update, all image files were properly located in sub-directory folders so that launching of the localised version in the Chrome browser produced the expected result, identical to the original home page (Figure 2.2).

[3] `http://jquery.com`

3.2 Downloading Missing URLs

After we localised the *home.html* page, we started to apply different types of Jalangi analyses to it. The instrumentation phase proceeded successfully and all JavaScript code was properly annotated. As explained in Section 2.5.1, we concluded that Jalangi's offline mode is not suitable for the Crime Inc. game. Therefore, we moved to the online analyses category where we explored the need to localise more than one page, in order to obtain interesting results. Since Jalangi instruments only locally stored JavaScript code, we needed to download other game pages to our machine and decompose them.

3.2.1 The Localizer Approach

The simplest way to download missing pages would be to use the Localizer tool that we developed earlier. However, this approach led us to undesired results due to technical aspects of the game implementation. As we investigated, the original version of the game uses only one static HTML page, *home.html*, and loads the rest of its content dynamically during the process of playing. It means that whenever a user plays the game and performs a trigger action, the game sends an HTTP request to the `http://m.staging.crimeinc.uken.com` server and downloads dynamic HTML and JavaScript code to inject it into the current page. For example, the following is the home page address of the original Crime Inc. game (tested in the Chrome browser with iPhone 4 emulator):

```
http://m.staging.crimeinc.uken.com/?udid=v&app_launch=1&app_version=2.00&
ios_bundle_id=com.uken.crimeinc&language=en&system_version=9
```

When the user presses the *"Battle"* button, an HTTP response from the server returns HTML together with JavaScript code which is embedded into the current *home.html* using an *ajax()* call (the *ajax()* function is described further in Section 3.2.3). After embedding the new content, the source code of the current page is modified and the new URL becomes:

```
http://m.staging.crimeinc.uken.com/?udid=v&app_launch=1&app_version=2.00&
ios_bundle_id=com.uken.crimeinc&language=en&system_version=9#url=battle
```

When we apply the Localizer tool to the above URL, it downloads the page called *battle.html* that consists of content almost identical to the *home.html* page, but with some additional fragments of HTML and JavaScript code that was loaded dynamically. In this way, we not only receive a lot of redundant content, but we also cannot utilise the downloaded *battle.html* page in the instrumented version of the game. On the one hand, the *ajax()* function requires dynamically loaded content to be unique. On the other hand, the *battle.html* page constructed by the Localizer tool is just another version of *home.html* with some additional code, rather than an individual page with absolutely unique content.

The alternative option was a replacement of the *ajax()* calls with the standard HTML *href*[4] attributes so that when a user presses the *"Battle"* button, the *home.html* page is redirected to a completely new *battle.html* page, instead of dynamically loading its content. This approach intervenes with the internal game mechanics and breaks the redirection process, as some game scripts are only invoked when particular dynamic content is loaded on the current page.

Due to the reasons described above, we had to find other ways of downloading the missing HTML files such that they contain only unique game code.

3.2.2 The Wget Tool

Wget[5] is a command line tool for retrieving files via the most widely-used Internet protocols, such as HTTP or FTP. The initial plan was to use this utility to automatically download missing HTML pages from the remote server. At first, we tried to imitate the original request manually and download the previously mentioned *battle.html* by issuing the following command in the Mac OS Terminal:

```
wget
--header="Host: m.staging.crimeinc.uken.com"  /
--header="User Agent: Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_0 like Mac
OS X; en-us) AppleWebKit/528.18 (KHTML, like Gecko) Version/4.0 Mobile/
7A341 Safari/528.16"  /
--header="Cookie: u_vln=31c435f4f5d8bf9d06eac40be5af1180; NTPClockOffset=
23|1408121028017"
http://m.staging.crimeinc.uken.com/?udid=v&app_launch=1&app_version=2.00&
ios_bundle_id=com.uken.crimeinc&language=en&system_version=9#url=battle
```

The above request fully imitates the one, which is created when playing the original version of the Crime Inc. game. Nevertheless, the file retrieved by executing the above command is identical to the one constructed by the Localizer tool – an HTML page with a lot of redundant content. Therefore, we had to search for another solution.

3.2.3 The Fly.js Analysis

After our unsuccessful attempts, we tackled the problem in a different way: we decided to detect a missing URL during an actual HTTP request. This approach required deeper understanding of the game and of processing dynamically loaded content. For this purpose, we developed a Jalangi analysis (Appendix B), named *Fly.js*, that works in the in-browser mode, detects missing URLs and saves the linked files to our machine.

The initial step was a detection of the common source of HTTP requests, which is responsible for loading of dynamic content. This step required a detailed analysis of a browser logs and the game source code. We used the Firefox browser v.31.0 in combination with the User

[4]The *href* attribute explicitly specifies the URL of the page to which it is linked.
[5]http://www.gnu.org/software/wget/

Agent Switcher v.0.7 add-on[6] set to iPhone 4. Figure 3.1 and 3.2 represent the example logs that we were examining:

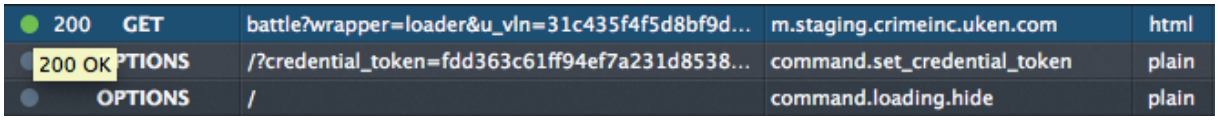

Figure 3.1: The Firefox console

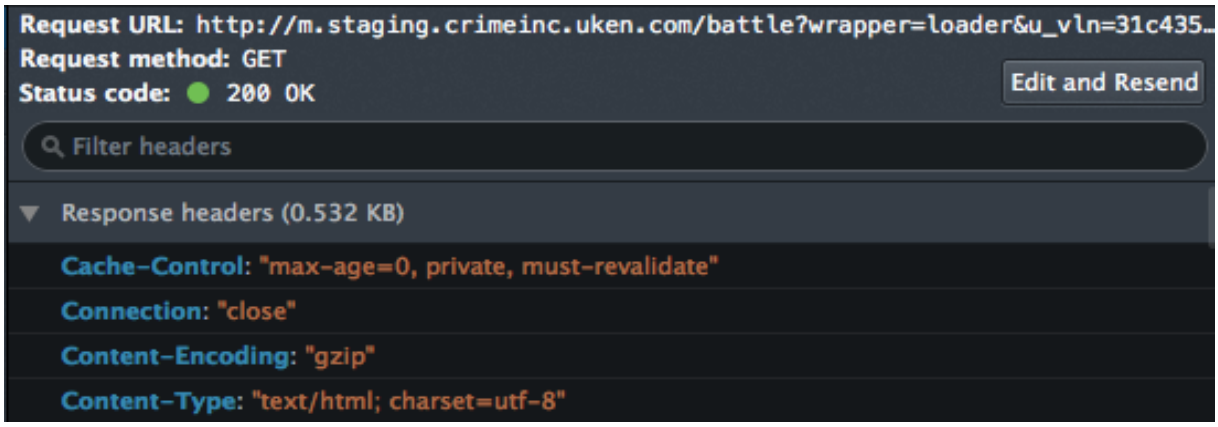

Figure 3.2: The request details

We examined the source code of the earlier decomposed *home.html* page, searching for the request details observed from the browser logs. As we found out, the only function responsible for loading of dynamic content was *ajax()*:

```
$.ajax( arguments );
```

Ajax()[7] is a jQuery function used to send various types of request (such as GET or POST) to a server and dynamically process or embed the received content into the current web page.

Having found the common source of HTTP requests, we needed to detect it automatically at run-time, that is, while playing the game. We decided to use the Jalangi framework for this, as it provides a wide range of callbacks (Section 2.3), among which we selected *invokeFunPre(iid, base, f, args, isConstructor)*. This method traces invocations of all JavaScript functions together with their properties. The arguments of this callback are: the unique Jalangi object id, the invoking object, the function body, its arguments, and a boolean variable indicating if the function is a constructor or not. The simplest way would be to detect the *ajax()* calls by tracking the function name:

```
this.invokeFunPre = function(iid, base, f, args, isConstructor) {
    var fname = f.toString().match(/function ([^\(]+)/));
    fname = fname || 'anonymous';
}
```

[6]https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher/

[7]The $ sign refers to the jQuery library call.

However, this solution did not work, as the *ajax()* function is interpreted as an anonymous function – a function without a name.

A more complicated solution implied an analysis of the body of the *ajax()* function in order to find a particular statement responsible for sending HTTP requests. For this purpose, we used the jQuery library (the *js0.js* file) downloaded earlier via the Localizer tool. By annotating the body of the *ajax()* function with *console.log()* statements, we were able to examine its behaviour and to find the statement that initialises all dynamic requests in the Crime Inc. game:

```
y.send(a || r.data == null ? null : r.data);
```

The next question that we had to solve was how to track the invocation of the above statement. A systematic approach was to use Jalangi to find any unique details about the above statement. As mentioned in the beginning of Section 3.2, we were able to instrument the jQuery library, in which we found the Jalangi equivalent of the above statement:

```
J$.M(301657, J$.R(301593, 'y', y, false, false), 'send', false) (J$.C
(20448, J$.C(20440, J$.R(301601, 'a', a, false, false)) ?  J$._() :
J$.B(26618, '==', J$.G(301617, J$.R(301609, 'r', r,  false, false),
'data'), J$.T(301625, null, 25, false))) ? J$.T(301633, null, 25,
false) : J$.G(301649, J$.R(301641, 'r', r, false, false), 'data'));
```

By taking advantage of using the unique object identifier *301657* derived from the method call *J$.M* specified in Table 2.2, we implemented a callback to print the details of the *y.send()* function:

```
this.invokeFunPre = function (iid, base, f, args, isConstructor) {
    if (iid == 301657) {
        console.log("Base = " + base + ". Function = " + f + ".
        Arguments = " + args);
    }
}
```

We tested the sample analysis in the online mode using the Chrome browser and received the following output:

```
Base = function send() {[native code]}. Function = [object XMLHttpRequest].
Arguments = [object Arguments]
```

Note that we could not use object identifiers (*iid*) to trace the *ajax()* calls for the following reason: Jalangi associates a unique id with every JavaScript variable, function and object, based on the number of previously assigned ids. Whenever source code of a program is changed, the new instrumentation may generate different id values for the same objects, because the total number of variables in the code has changed. Therefore, the final version of the callback method, detecting the *ajax()* calls, is based on the function's details, rather than on a concrete identifier value:

```
this.invokeFunPre = function (iid, base, f, args, isConstructor) {
    if (base == "function send() { [native code] }" && f == "[object
        XMLHttpRequest]")
```

```
    { urlSent = true; }
}
```

Using this method, the analysis sets the global *urlSent* flag to *true*, whenever the *y.send()* statement is reached within the *ajax()* body.

After we were able to detect an HTTP request, the next task for us was to derive the URL from that request. We accomplished this by implementing another callback method in our analysis – *readPre(iid, name, val, isGlobal)*. It records every variable read in a program. When we analysed the *y.send(a || **r.data** == null ? null : **r.data**)* statement, we observed that it sends a request accompanied with the object ***r***. Moreover, as we further examined the *ajax()* function body, we found that it contains the ***r.url*** property – the target URL that we needed to detect automatically in our analysis. The *readPre()* function allows us to track the name of the variable that is being read. In combination with the earlier defined *urlSent* flag, we developed the following conditional statement to catch missing URLs:

```
this.readPre = function (iid, name, val, isGlobal) {
    if (urlSent && name == "r" && val == "[object Object]")
    {
        urlSent = false;

        // check that val.url does not exist on the local
        // machine and save it to the download map
        ...
    }
}
```

To avoid redundant downloads, all URLs found during the *Fly.js* analysis execution are stored in a *Map* object. A user has to execute the *Shift+Alt+T* keyboard combination to download the links recorded in *Map*. The downloading of files is implemented using the *SaveToDisk()* function written by Khan[8].

When the user plays the instrumented version of the game using the *Fly.js* analysis, a combination of the implemented callbacks *invokeFunPre()* and *readPre()* successfully recognises an *ajax()* request and derives the corresponding URL link from it. For example, when the user presses the *"Battle"* button, our analysis catches the following URL:

```
battle?wrapper=loader&u_vln=31c435f4f5d8bf9d06eac40be5af1180&udid=v&
app_launch=1&app_version=2.00&ios_bundle_id=com.uken.crimeinc&language=en&
system_version=9&rt_con=2003454&rt_evt=2&rt_dur=231&_=1408209568751
```

This URL is then used to download the corresponding file from the original `http://m.staging.crimeinc.uken.com/` server. The download process only works in the Firefox browser, as Chrome complains about the provided URL and produces an *"unknown error"* message.

[8] http://muaz-khan.blogspot.ru/2012/10/save-files-on-disk-using-javascript-or.html

In our experiments with the *Fly.js* analysis, we used the Firefox v.31.0 together with the User Agent Switcher v.0.7 to emulate the iPhone device. We applied the analysis to download missing URLs using the commands described in Section 2.5.2 to instrument the source code and initialise the in-browser mode:

```
node src/js/commands/instrument.js --inbrowser --smemory --analysis
src/js/analyses/Fly.js --outputDir /Library/WebServer/jalangi/tests/
tests/uken/game

open file://localhost/Library/WebServer/jalangi/tests/game/home.html
```

Notice that the location of the instrumented version has changed to */Library/WebServer/-jalangi/tests/* – the purpose of this is explained in Section 3.3.2.

After the browser window was opened, we performed as many game actions as was possible to collect missing links that were available at the *home.html* page. Pressing *Shift+Alt+T* saved the linked files to our hard drive. These files contained only unique HTML and JavaScript code, as was required by our Jalangi analysis.

3.2.4 Processing the Downloaded Files

By this moment, we had a folder of HTML files downloaded in the process of applying the *Fly.js* analysis to the *home.html* page. Before adding these files to the instrumented version of the game, we had to decompose them in a similar way to the one described in Section 3.1.

For this purpose, we created a new version of the Localizer tool that could work in the offline mode, that is, decompose the pages stored on our hard drive. The new tool utilises the code from Localizer to extract images and JavaScript code from the target HTML. It starts by exploring the given root directory, decomposes all the HTML files found there, and recursively proceeds to the sub-directories until all the paths are examined. The new version also incorporates a slightly different technique of modifying JavaScript links for the reason described below.

Default mapping – a default redirection is widely used in the implementation of the Crime Inc. game. To understand the behaviour of that feature, consider the following scenario: *ajax()* sends a request to the `http://hostname/battle/?wrapper=...` address and waits for the response. Since the initial request does not specify the name of the requested file (*/battle/?wrapper=...* does not contain the name of an HTML file, but only contains the parameters), the default mapping will redirect the request to the new address `http://hostname/battle/index.html?wrapper=...`. In this way, whenever a URL misses the name of an HTML file, the default mapping automatically substitutes the *index.html* string.

To address the default mapping in the original version of the game, we had to restructure the folders containing our *.html* files downloaded in the process of the *Fly.js* analysis. Every

default HTML file with name *urlname.html* was renamed to *index.html* and relocated into the appropriate folder */urlname/index.html*. Using this technique, we were able to properly locate the game resources of the instrumented version to further imitate HTTP requests in a strict accordance to the original ones.

In summary, we used the *Fly.js* analysis to detect and download missing URLs that were later decomposed by the modified localisation tool. Nevertheless, the newly downloaded pages were unavailable for access from our analysis, as the *ajax()* function was sending HTTP requests to the original `http://m.staging.crimeinc.uken.com/` server. In order to perform in-browser analyses, the requests had to be redirected to the instrumented HTML and JavaScript files stored on our machine. The next section describes how we achieved this.

3.3 Merging the Instrumented Files

Even though we were able to download and decompose the additional HTML pages from playing the game, we still had to find a way to redirect the original requests to these pages at the analysis run-time. Before discussing the implementation of the redirection technique (Section 3.3.3), we first introduce the need to set up a local server.

3.3.1 Setting up the Apache Server

When we tested the redirection of HTTP requests to the instrumented files, we discovered a violation of the *Same Origin* policy. For example, when the *ajax()* function sends a request to the `file://localhost/Library/WebServer/tests/game/battle/` address, the request goes to the machine's file system – ***file://localhost/***. In this case, any HTTP request sent to the file system fails, because the file system sends back a response annotated with a null-origin header, as illustrated in Figure 3.3:

```
XMLHttpRequest cannot load file://localhost/Library/WebServer/tests/game/battle/.
Received an invalid response. Origin 'null' is therefore not allowed access.
```

Figure 3.3: The null-origin error from the Chrome browser console

For this reason, we decided to create a local server that would respond to the incoming requests with an appropriate header.

Apache[9] allows a user to set up a web server locally. This application was installed on our machine by default (Mac OS X v.10.7.5), so we only had to tune some settings to switch the server on and update the server address. To enable the local server, we modified Apache's configuration file */etc/apache2/httpd.conf*. We uncommented the below statement by removing the '#' sign:

[9]http://www.apache.org

```
#LoadModule php5_module libexec/apache2/libphp5.so
```

This instruction enables the PHP library and initialises the local host. The next step was to edit the root directory of the server. For the reason further explained in Section 3.3.2, we attached the local host directory to the Jalangi folder. We modified paths in the configuration file as follows:

```
DocumentRoot "/Library/WebServer/jalangi"
<Directory "/Library/WebServer/jalangi">
```

Afterwards, the Apache server had to be restarted by issuing the following instruction in the Terminal application:

```
sudo apachectl restart
```

Once the local server started to work properly, we repeated the *ajax()* request to the previously mentioned *battle* URL, but now using the *localhost* address – `http://localhost/tests/game/battle/`. This time, the *Same Origin* policy generated another type of exception, complaining about Cross-Origin-Resource-Sharing (CORS), as shown in Figure 3.4:

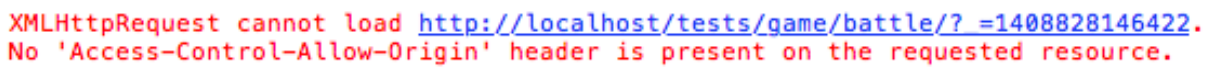

Figure 3.4: Cross-Origin-Resource-Sharing error

The error indicates that the *Same Origin* policy forbids any form of communication between two servers with different domains, unless they explicitly specify the settings that allow external requests. In order to enable CORS, we had to set up a virtual host container to annotate the responses with the required headers. The configuration file for virtual hosts *httpd-vhosts.conf* is located in the */etc/apache2/extra/* directory. We updated it by creating a new virtual container and enabling CORS as follows:

```
<VirtualHost *:80>
    ServerName localhost

    Header always set Origin "*"
    Header always set Access-Control-Request-Method "GET, POST, PUT, DELETE
       , OPTIONS"
    Header always set Access-Control-Request-Headers "Authorization,
    X-Requested-With, X-Prototype-Version, X-CSRF-Token, Content-Type,
    Accept, Origin"

    Header always set Access-Control-Allow-Origin "*"
    Header always set Access-Control-Allow-Methods "GET, POST, PUT, DELETE,
        OPTIONS"
    Header always set Access-Control-Allow-Headers "Authorization,
    X-Requested-With, X-Prototype-Version, X-CSRF-Token, Content-Type,
    Accept, Origin"

    DocumentRoot /Library/WebServer/jalangi
</VirtualHost>
```

These settings allow external requests to communicate with our local host by annotating the outgoing responses with CORS-friendly headers. Finally, we restarted the Apache server to apply the recent changes:

```
sudo apachectl restart
```

After the server enabled support for cross-origin requests, the *Fly.js* analysis gained access to the instrumented files located on our machine.

3.3.2 Locating the Instrumented Files

As described in the previous section, we set up the local server to overcome the *Same-Origin* policy and maintain the external HTTP requests sent to our server. By that moment, our analysis was able to access URLs starting with `http://localhost/`. For this reason, we modified all JavaScript and HTML links within the instrumented files and attached them (links) to the `http://localhost/` address (Section 3.1). In order for an HTTP request to find an instrumented file linked to the *localhost* address, this file needs to be located in the server's root directory.

Instead of transferring an instrumented content to the *localhost* location every time an instrumentation phase is performed, we decided to set the Jalangi folder as the server's root directory:

```
/Library/WebServer/jalangi/
```

Additionally, the example below, extracted from the instrumented version of the *battle/index.html* page, shows how Jalangi annotates source HTML files with additional JavaScript links:

```
<head>
...
<script src="/Library/WebServer/jalangi/src/js/Constants.js"></script>
<script src="/Library/WebServer/jalangi/src/js/Config.js"></script>
<script src="/Library/WebServer/jalangi/src/js/Globals.js"></script>
...
</head>
```

These links are automatically injected by Jalangi to import the required source files during an in-browser analysis. When a user performs an action that invokes the *ajax()* request sent to `http://localhost/battle/?wrapper=...`, the links listed above are also processed dynamically. In this case, the *ajax()* method, when embedding the *battle/index.html* page, will attempt to search for the specified links in the file system at `file://localhost/` by default. Since any HTTP request sent to the file system `file://` fails (Section 3.3.1), we also had to automatically redirect Jalangi source requests to the *localhost* address, where these files were located, because we set the Jalangi folder as the root directory.

3.3.3 Redirecting the Requests

We decided to expand the existing *Fly.js* analysis by implementing an additional callback function, responsible for redirection of the original requests to our local server. When studying the *ajax()* method, we discovered that the object ***r*** contains the ***url*** field, as was discussed in Section 3.2.3. In order to redirect an HTTP request, we had to detect the moment, when this property is instantiated and change the base domain to our local server. For this purpose, Jalangi provides the *putField(iid, base, offset, val)* callback that is triggered by every *"put"* access to an object's field.

In comparison to the previously defined callbacks that detect missing URLs, the *putField()* method does not provide the name of the object that accesses a field. Due to this limitation, we had to examine the specific properties that belong to the ***r*** object and use them, as the part of the conditional statement that detects the access to the ***r.url*** field:

```
this.putFieldPre = function (iid, base, offset, val) {
    if (base.constructor.name == "Object" && base.hasOwnProperty("url") &&
        base.hasOwnProperty("type") && offset == "url" && urlSent)
    {
        // change the URL
        ...
        return newURL;
    }
    return val;
}
```

In fact, using two fields ***type*** and ***url*** in combination with the earlier declared *urlSent* flag was sufficient for the successful recognition of the ***r.url*** field access. In our analysis, we observed various types of HTTP requests where each of them had to be treated differently:

- *Jalangi source file* – this type of request attempts to access the Jalangi source libraries linked from within the instrumented HTML files. It is the only kind of request that starts with the `file://localhost/` address. An example URL is as follows:

  `file://localhost/Library/WebServer/jalangi/src/sourceLibrary.js`

  Since the */Library/WebServer/jalangi/* folder is the local host address, the extended version of *Fly.js* redirects this request to:

  `http://localhost/src/sourceLibrary.js`

- *Content requests* – these requests attempt to load new content on the current page. Such request URLs start without a domain specified. Below is an example of the original request:

  `/battle/rivals?wrapper=loader&...`

  For the above address, our analysis builds the new URL as follows:

`http://localhost/tests/game/battle/rivals?wrapper=loader&...`

Notice that the redirected request has an intermediate path */tests/game/* – this is the directory where the source files are saved after instrumentation.

- *Default requests* – this type is similar to the previous one, except that the *ajax()* function applies the default mapping technique to the original request:
  `empire/?wrapper=...`

  To imitate the original behaviour, the extended analysis automatically implements the default mapping and the new address becomes:
  `http://localhost/empire/index.html?wrapper=...`

- *Redirected requests* – sometimes the analysis detects a URL that was already modified in the process of analysis. Such a URL starts with the `http://localhost/` string and is simply ignored.

With accordance to the new callback function and the Apache local server running, we updated the downloading part of the *Fly.js* analysis so that it can decide if a certain URL is missing by accessing the local host directory and sending a test request to that file. If a request receives a *"not found"* message in response then the URL is saved to the download list.

After the *Fly.js* analysis was expanded to a new version that automatically supports redirection, we finally could access newly instrumented files from the *home.html* page and continue to download missing game resources by accessing the newly decomposed pages.

Chapter 4

Experiments and Results

After collecting a sufficient number of HTML pages from the Crime Inc. game, we decomposed them and stored the obtained files in the *Library/WebServer/jalangi/tests/uken/game* directory. We then instrumented the source code and started to apply different in-browser analyses offered by the Jalangi framework, summarised in Table 2.3. At first, we give an overview of additional functions implemented in every analysis that we performed in this research.

4.1 Preparation

The redirection function was used to enable an analysis to freely navigate within the instrumented version of the game. We updated the original code of every analysis by inserting the *putField(iid, base, offset, val*) callback that we developed earlier (Section 3.3.3). This modification provided an opportunity to access the files stored in the local server directory *Library/WebServer/jalangi* by using the `http://localhost/` address.

During our experiments with different analyses, we noticed that Jalangi only records the location of an error at the lowest level of a program's source code and does not provide any facility to find the initial statement that caused this error. For example, the original version of the *CheckNaN.js* analysis produces the following output:

```
Observed NaN at (js0.js:6498:28) 5 time(s).
```

The *(js0.js:6498:28)* location refers to:

```
pageX: (Math.min.apply(Math, n) + Math.max.apply(Math, n)) / 2
```

Using the obtained information, we can conclude that this particular line of code generates a not-a-number (NaN) value, but we cannot trace the initial cause of that error at higher levels of the source code. For this purpose, we added the *CallStack* object to the global *sandbox* variable described in Section 2.3. The source code is presented below:

```
function CallStack () {
    var iidToLocation = sandbox.iidToLocation;
    var stack = [];

    this.functionEnter = function(iid, val, dis, args) {
        stack.push(val);
    }

    this.functionExit = function(iid) {
        stack.pop();
    }

    this.toString = function() {
        var cs = "";
        for (var i = stack.length - 1; i >= 0; i--)
            cs += iidToLocation(stack[i]) + (i != 0 ? " -> " : "");
        return cs;
    }
}
sandbox.CallStack = new CallStack();
```

This object provides two callbacks to trace the function call stack of a program: *functionEnter(iid, val, dis, args)* and *functionExit(iid)*. In this way, every analysis can retrieve the current stack by referring to the *sandbox.CallStack.toString()* method.

4.2 Chained Analysis

In the beginning of the research, Jalangi did not support chained analysis in the in-browser mode. However, in the process of work, the Jalangi developers released a new version of the framework that introduced a modified chained analysis working in online mode.

Chained analysis is a supporting unit that does not perform any tests by itself, but serves to combine other analyses and run them sequentially. We updated the original *ChainedAnalysis2.js* code to incorporate the redirection feature and the call stack, as described in the previous section. We named the new version *ModifiedChained.js* and used it in combination with six other dynamic analyses, accompanied with the *Utils.js* file that provides a method to print the results:

```
node src/js/commands/instrument.js --analysis2 --analysis src/js/analyses2/
    ModifiedChained.js --analysis src/js/analyses2/Ryzhov/Utils.js --
    analysis src/js/analyses2/Ryzhov/CheckNaN.js --analysis src/js/analyses2
    /Ryzhov/FunCalledWithMoreArguments.js --analysis src/js/analyses2/Ryzhov
    /CompareFunctionWithPrimitives.js --analysis src/js/analyses2/Ryzhov/
    ShadowProtoProperty.js --analysis src/js/analyses2/Ryzhov/
    ConcatUndefinedToString.js --analysis src/js/analyses2/Ryzhov/
    UndefinedOffset.js --outputDir /Library/WebServer/jalangi/tests/ tests/
    uken/game
```

The above instruction launches the instrumentation phase that uses the *analysis2.js*[1] module to insert callback functions. The command also specifies the main *ModifiedChained.js* unit that takes a list of actual analyses to be combined.

After the instrumentation phase terminated, we played the game for a short period of time and pressed the *Shift+Alt+T* combination to print the results in the browser console. Since the full log is huge, we only present the most interesting samples.

CheckNaN.js

This unit tracks if an execution invokes a NaN value, records the location of the error and the number of occurrences of that error. The following observations were made by applying this analysis to the instrumented game:

```
(1) Observed NaN at (js0.js:6498:28) 22 time(s).
CallStack: (js0.js:6493:24) -> (js0.js:6432:31) -> (js0.js:6411:51)

(2) Observed NaN at (js2.js:1567:28) 1 time(s).
CallStack: (js2.js:1559:3) -> (js2.js:1317:3) -> (js2.js:1193:25)
```

The first NaN observation is a pure jQuery error. The value occurs at the following line:

```
(1) pageX: (Math.min.apply(Math, n) + Math.max.apply(Math, n)) / 2
```

The recorded function stack consists of the following elements:

```
getCenter(t) -> collectEventData(t, r, i) -> onTouch(t, r, u)
```

The interpretation is: the *onTouch(t, r, u)* method invokes *collectEventData(t, r, i)* which in turn invokes *getCenter(t)* which contains *(1) pageX:* .... The call chain indicates that a NaN value appears every time a user performs a click action in a browser window that invokes the *onTouch(t, r, u)* method.

The second error refers to the *js2.js* file – a part of the game source code. The corresponding line is:

```
(2) currentPage_zindex = parseInt(currentPage.css("z-index")) || 1;
```

The *parseInt(currentPage.css("z-index"))* function generates a NaN value, whereas the final result of the *(NaN || 1)* operation produces 1. It means that the analysis mistakenly records a NaN value that is not assigned to any concrete variable. The issue was emailed to Koushik Sen, who is responsible for the development of the Jalangi analyses.

[1] *Analysis2.js* is an advanced interface for performing in-browser analyses. It is more efficient and less error prone than the previous version *analysis.js*.

FunCalledWithMoreArguments.js

FunCalledWithMoreArguments records the number of occurrences where a particular function was invoked with more arguments than expected. Native functions are ignored, as Jalangi does not instrument native JavaScript code. Since this analysis generated a large number of warnings, mostly related to the *js0.js* file (jQuery), we selected the most promising samples:

```
(1) Function at (js2.js:1022:29) called 8 time(s) with more arguments than
   expected.
(2) Function at (js2.js:1321:5) called 7 time(s) with more arguments than
   expected.
(3) Function at (battle/battlejs0.js:100:9) called 1 time(s) with more
   arguments than expected.
(4) Function at (missions/missionsjs0.js:99:9) called 1 time(s) with more
   arguments that expected.
(5) Function at (missions/missionsjs1.js:53:3) called 1 time(s) with more
   arguments that expected.
```

The first two examples complain about the respective statements:

```
(1) var loadBodyPerfEvent = PerfMonitor.loadBody(url);
(2) PerfMonitor.loadSuccess(otherData.url, otherData.loadBodyPerfEventId);
```

The source of these errors is a specific implementation of the *PerfMonitor*, which is part of the *jQuery* library. As we found out, methods of this object are generated dynamically, as shown in the example from the *js2.js* file:

```
perfEventSequences = [ ... , { name: 'loadBody', setupFunction:
   setupFromUrl}, ...,  name: 'loadSuccess', hasAsyncParent: true,
   setupFunction: setupFromUrlAndParentId }, ... ]

initPerfMonitor(perfEventSequences, 87, 2003454, ... );
```

By testing the behaviour of the *PerfMonitor* object, we figured out that a function call made from this object is initially converted into the *getFakePerfEvent()* function, which does not take any arguments. Therefore, the analysis evaluates all function invocations from the *PerfMonitor* object as an arguments mismatch error.

Samples 3 and 4 refer to the following lines of code from the *battlejs0.js* and *missionsjs0.js* files respectively:

```
(3) $("#Energy_amount").U_BasicAnim('stop').U_BasicAnim('flash', {textFlash
   : true, textShadow: true, ...});
(4) $("#money.price").U_BasicAnim('stop').U_BasicAnim('flash', {textFlash:
   true, textShadow: true, ...});
```

The cause of the errors is the *U_BasicAnim(n)* function, provided by the jQuery library. To understand the details of the problem, we examined the body of this function:

```
U_BasicAnim: function (n) {
    var r = arguments;
    ...
}
```

The above definition shows that the function takes a single input parameter *n*. Nevertheless, it can read all arguments passed to it, storing them into the *r* variable. Such behaviour is explained by the nature of the JavaScript language: every function is annotated with the default *arguments* variable that stores a full list of parameters provided to that function [3]. According to the definition, the analysis works correctly and records all cases, where the *U_BasicAnim(n)* function is invoked with more than one parameter, even if an error does not have a harmful impact on the execution of an application.

The example 5 is associated with the following statement from the *missionsjs1.js* file:

```
(5) $("#overlay").after("<div class...");
```

The analysis signals an error due to another jQuery function *after()*. We investigated the details of that method and found the definition listed below:

```
after: function () {
    ...
    if (arguments.length) {
        ...
    }
}
```

By default the function takes no arguments, but can read them dynamically during run-time. Similar to errors 3 and 4, this one does not affect the program's behaviour in a negative way.

All other errors, recorded by *FunCalledWithMoreArguments* were similar to the samples illustrated above.

CompareFunctionWithPrimitives.js

This analysis reports an error if a program performs a comparison operation between a function and a primitive[2]. The list of included comparison operators is: ==, ===, !=, !==, <, >, <=, >=. This test did not produce any warnings and indicated that all comparisons in the source code of the game are performed correctly.

ShadowProtoProperty.js

ShadowProtoProperty tests if a property of an object, created during the application execution, shadows a property defined in the prototype of that object. A prototype is an additional object associated with every JavaScript object and can be accessed using *Object.prototype*. The latter object inherits the properties from the prototype. For example, an array created using the constructor *new Array()* inherits all its properties from *Array.prototype* [3]. All errors reported

[2]In the context of *CompareFunctionWithPrimitives*, primitive is any JavaScript number, string or boolean.

by this analysis referred to the jQuery file *js0.js*. Some samples are presented below:

```
(1) Written property length at (js0.js:1482:72) 4018 time(s) and it shadows
    the property in its prototype.
CallStack: (js0.js:1479:23) -> (js0.js:1448:21) -> (js0.js:3845:202) -> (
   js0.js:3886:15) -> (js0.js:3873:13) -> (js0.js:3910:15)
(2) Written property selector at (js0.js:1493:52) 3 time(s) and it shadows
   the property in its prototype.
CallStack: (js0.js:1479:23) -> (js0.js:1448:21) -> (js2.js:2168:21) -> (js0
   .js:1558:24) -> (js0.js:1721:108)
```

For all generated errors, the call stack was registering the same initial location *(js0.js:1479:23)*. We examined the body of the related function and found the following code:

```
init: function (e, n) {
    ...
    (1) this.length = 1;
    ...
    (2) this.selector = e;
    ...
}
```

The above statements, indeed, rewrite the prototype properties *length* and *selector*. Nevertheless, these changes do not affect the program's behaviour.

ConcatUndefinedToString.js

ConcatUndefinedToString analysis checks if any string concatenation contains *undefined* arguments. The analysis reported two errors illustrated below:

```
(1) Concatenated undefined to a string at (js7.js:15:5) 6 time(s).
CallStack: (js7.js:7:1) -> (js7.js:22:1)

(2) Concatenated undefined to a string at (js3.js:40:30) 3 time(s).
CallStack: (js3.js:21:24) -> (js3.js:154:53) -> (js0.js:1316:5) -> (js0.js
   :2164:17) -> (js0.js:2072:167)
```

The first error points to the code statement at the *js7.js* file:

```
$('#banner_carousel.carousel_button_' + $oldItem.data('index')).remove()
```

In the above line, the *data('index')* function returns the *undefined* result. This function belongs to the jQuery library and returns a value associated with the provided parameter. In this case, the error indicates that no element with the *index* key is found in the data storage. The call stack refers to the following function chain:

```
(1) showCarouselItem(index) -> nextCarouselItem()
```

As we found out by further examining the *js7.js* file, *nextCarouselItem()* is a function that updates the advertisement banner initialised at:

```
carouselIntervalId = setInterval(nextCarouselItem, 8000);
```

The above line of code illustrates that the error caught by the analysis is generated every eight seconds, signalling that the *'index'* element is missing on the current web page.

The second error is generated during the execution of the *js3.js* module:

```
(2) $('.chat_header').find("[data-tab=" + tab + "]").addClass('selected');
```

The error shows that the *tab* variable is evaluated as *undefined*. Below is the interpretation of the recorded call stack:

```
chatTabClick(tab) ->
$('.segmentedControlBase .tab').live("click", function() {
    var tab = $(this).data('tab');
    chatTabClick(tab);
}); -> o(e) -> ...
```

The important segment in the above trace is the second function *live("click", function() ...* that creates the *tab* object using the jQuery function *data('tab')*. Similarly to the first error produced by the analysis, this one signals that the *'tab'* element is missing on the web page.

UndefinedOffset.js

The *UndefinedOffset* analysis was given as an example in Section 2.3. It records all cases where a program attempts to access a non-existing property of an object. Note that this test does not validate if a value of a property is *undefined*, but only checks if it belongs to a particular object. The output of this analysis reported a single error, which is originated at jQuery:

```
Accessed property 'undefined' at (js0.js:1813:29) 45 time(s).
CallStack: (js0.js:1807:21) -> (js0.js:2093:17) -> (js0.js:2359:17)
```

The location *(js0.js:1813:29)* referred to the following statement from the jQuery library:

```
o = r ? s[i] : i;
```

The above line indicates that the *i* property of the *s* object is evaluated as *undefined*. The interpretation of the call stack did not lead us to any meaningful conclusion, because all related functions were specific to jQuery. Apart from that error, we can conclude that all property referencing statements are correctly implemented in the source code of the game.

4.3 Likely Type Inference

This analysis tracks if a function, object or array created at a particular program location can assume multiple inconsistent types. It uses the shadow memory described in Section 2.2 to annotate the program variables with their types and a location of creation. The shadow values are propagated along with the execution of an application and stored in a *Map* object. When the result printing method is invoked (*Shift+Alt+T*), the analysis compares types of identical objects recorded in the map and generates a warning if an object was saved with more than one type.

We added the redirection method to the *LikelyType.js* module, but we did not implement the function call stack, because the data recorded by the original analysis provides sufficient information to identify the source location of an error. To instrument the game we instructed the following command in the Terminal window:

```
node src/js/commands/instrument.js --inbrowser --smemory --analysis src/js/
   analyses/Ryzhov/LikelyType.js --outputDir /Library/WebServer/jalangi/
   tests/ tests/uken/game/
```

After playing the instrumented game for a short period, we printed the results using the *Shift+Alt+T* combination. The vast majority of warnings originated at the jQuery library, and only two of them were related to the actual game code:

```
(1) Warning: arg3 of function originated at (js2.js:1019:3) has multiple
   types:
    1a) undefined found at (js2.js:2508:9),
    1b) null found at (js2.js:2083:5)

(2) Warning: return of function originated at (js2.js:692:3) has multiple
   types:
    2a) object originated at (js2.js:654:17) found at (js2.js:606:22),
    2b) object originated at (js2.js:653:17) found at (js2.js:606:22),
    2c) object originated at (js2.js:681:15) found at (js2.js:606:22)
```

The first sample indicates that the third argument of the function, invoked at the location *(js2.js:2508:9)*, is *undefined*. The associated code for the warning is below:

```
1a) var selector = a.attr("data-parent");
    loadBody(href, {}, selector, options, noScroll);
1b) loadBody(url, {}, null);
```

The cause of the error is that the *selector* variable in the *1a* statement is *undefined*. The *attr("data-parent")* function, which produces the incorrect result, is part of the jQuery library that serves to get the value of an attribute that matches the provided element name. In this way, the *undefined* value is returned due to absence of the *"data-parent"* attribute on the current web page.

The second warning is associated with a potential inconsistency of returned objects. We analysed the related code statements:

```
2a) '/battle': {contentAreaSelector: ..., navButton: ..., beforeLoad:
   function() {...}, advisorClass: ...}
2b) '/home': {contentAreaSelector: ..., navButton: ..., advisorClass: ...}
2c) '/clan': {contentAreaSelector: ..., advisorClass: ...}
```

As presented above, all object have a different number of properties: *battle* has four, *home* has three and *clan* has two. The warnings are generated when we play the game and access different game tabs by pressing the navigation buttons. Our analysis detects types inconsistency, because the page loader block at *(js2.js:606:22)* processes the same object with various number of properties.

In conclusion, the *LikelyType* analysis works properly on the instrumented game and detects potential errors accordingly to the Jalangi specification. However, none of the reported warnings pointed on an actual bug, indicating that all type definitions in the game code are consistent.

4.4 Statistical Analyses

This section briefly describes two other dynamic analyses provided by Jalangi that work in in-browser mode: these analyses do not detect any potential bugs in the execution of an application, but serve to collect source code information that could be useful for the application developers to examine concrete functions and their usage statistics.

Call Graph Analysis

This unit implements its own call stack that traces all function invocations during a particular period of execution. Additionally, it saves set of called methods for every recorded function. Native JavaScript code is not considered due to the Jalangi restrictions. Below is an example entry from the log produced by the analysis:

```
Function setupRealtimeEventHandlers defined at (js2.js:443:1) was invoked 1
time(s) and it called:
  function anonymous defined at (js0.js:1446:21) 1 time(s) at call site (
     js2.js:444:7)
  function anonymous defined at (js0.js:2395:23) 1 time(s) at call site (
     js2.js:444:7)
```

The example illustrates the information about the *setupRealtimeEventHandlers()* function defined in the *js2.js* file. It shows details about the methods that were invoked by that particular function and the related call locations. Notice that jQuery functions are evaluated as anonymous.

Although this analysis does not provide any technique for automatic error detection, the game developers can use the obtained statistics to debug the source code manually by tracking the behaviour of the concrete function.

Object Allocation Tracker

The example output of the allocation tracker was illustrated in Section 2.5.2.1. In the in-browser mode, this analysis traces the number of objects created at a particular execution period of a web application and evaluates frequency of accessing these objects. On the one side, the results of applying *ObjectAllocationTracker* to the Crime Inc. game were too huge and, therefore, were indefinite for us, external observers. On the other side, the game developers can use the obtained statistics to precisely analyse memory efficiency of a particular function.

Chapter 5

Conclusion

In this research, we explored the capabilities of the Jalangi framework and investigated the details of implementing new dynamic analyses and of applying previously existing ones. We also explained the reasons for the incompatibility of Jalangi's offline mode with complex web applications that contain more than one JavaScript source file. For this research we selected the online mode, which is suitable for analysing JavaScript games of any scale.

We demonstrated a need to localise a web application and to extract the JavaScript code in order to apply the Jalangi analyses directly in a browser window. For this purpose, we developed the Localizer tool to download and decompose application's HTML pages, particularly testing our approach on the Crime Inc. game. In addition, we discussed the various attempts at expanding the localised version of the game to obtain a larger set of original HTML pages. To address the problem, we wrote the *Fly.js* analysis that is able to detect the missing resources on a user's machine and save them locally. We also explained the difficulties associated with dynamic content loading caused by the jQuery library. We resolved the issue by setting up a local server and implementing the redirection feature. We expanded the *Fly.js* analysis so that the new version allows a user to play the instrumented game by redirecting the original *ajax()* requests to the local server and to download the external resources missing on that server.

We also showed the required preparation procedures for applying the Jalangi analyses to the instrumented game. We performed various analyses on the Crime Inc. game and interpreted the results. Even though our experiments only reported warnings that do not affect the game behaviour, we demonstrated the robustness and correctness of the Jalangi analyses on the Crime Inc. game. We can conclude that the game is written in a professional manner and the code does not contain bugs of the specific types that we were looking for.

In summary, we developed a systematic approach to analysing JavaScript games of any scale using the Jalangi tool. Because jQuery is the most popular JavaScript library[1], we can

[1] http://blog.jquery.com/2014/01/13/the-state-of-jquery-2014/

claim that our technique is applicable to a large set of JavaScript applications that use the *ajax()* function to dynamically load additional HTML and JavaScript content.

Appendix A

Localizer Source Code

```
import java.awt.image.RenderedImage;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.net.URL;
import java.net.URLConnection;
import java.nio.channels.Channels;
import java.nio.channels.FileChannel;
import java.nio.channels.ReadableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import javax.imageio.ImageIO;
import org.apache.commons.io.IOUtils;
import org.jsoup.Jsoup;
import org.jsoup.nodes.DataNode;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.parser.Tag;
import org.jsoup.select.Elements;

public class Localizer
{
    // mode
    private static boolean verbose;

    // name of directory in which all files are stored
    private static String directory;
```

```
// default directory
private static final String DEFAULT_DIRECTORY = File.separator + "tmp"
    + File.separator + "d" + System.currentTimeMillis() + File.separator
    ;

// cookies
private static String cookies;

// default cookies
private static final String DEFAULT_COOKIES = "";

// user agent
private static String userAgent;

// default user agent
private static final String DEFAULT_USER_AGENT = "Java/" + System.
    getProperty("java.version");

// predefined user agents
private static final Map<String, String> USER_AGENT = new HashMap<
    String, String>();
static
{
    USER_AGENT.put("iphone4", "Mozilla/5.0 (iPhone; U; CPU iPhone OS 4
        _2_1 like Mac OS X; en-us) AppleWebKit/533.17.9 (KHTML, like
        Gecko) Version/5.0.2 Mobile/8C148 Safari/6533.18.5");
    USER_AGENT.put("ipad4", "Mozilla/5.0 (iPad; CPU OS 7_0 like Mac OS
        X) AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile
        /11A465 Safari/9537.53");
    USER_AGENT.put("x11", "Mozilla/5.0 (X11; Linux x86_64; rv:30.0)
        Gecko/20100101 Firefox/30.0");
}

// counter used to name HTML files
private static int HTMLcounter = 0;

// map from already localized HTML files to their counter
private static Map<String, Integer> HTMLmap = new HashMap<String,
    Integer>();

/**
 * This exception is thrown whenever a non-supported feature is
     encountered.
 */
private static class NotSupportedException extends Exception
{
    /**
     * Initializes this exception with the given error message.
     *
     * @param message the error message of this exception.
     */
    NotSupportedException(String message)
    {
        super(message);
```

```
    }
}

/**
 * This exception is thrown whenever something has been skipped.
 */
private static class SkippedException extends RuntimeException
{
    /**
     * Initializes this exception with the given error message.
     *
     * @param message the error message of this exception.
     */
    SkippedException(String message)
    {
        super(message);
    }
}

// map from URLs which are not supported to their exception
private static Map<String, NotSupportedException> notSupportedMap = new
    HashMap<String, NotSupportedException>();

/**
 * Reads the file with the given name and stores it in a file.
 *
 * @param name the name of the file.
 * @return the name of the created file.
 * @exception MalformedURLException if the given name is a malformed
    URL.
 * @exception IOException if an IO error occurred during reading or
    writing
 * the file.
 * @exception NotSupportedException if the type of the file is not
    supported.
 * @exception UnknownHostException if the given name contains an
    unknown host.
 */
private static String read(String name) throws MalformedURLException,
   IOException, NotSupportedException, UnknownHostException
{
    if (notSupportedMap.containsKey(name))
    {
        throw notSupportedMap.get(name);
    }
    else
    {
        try
        {
            // determine the type of the URL
            URL url = new URL(name);
            HttpURLConnection connection = (HttpURLConnection) url.
               openConnection();
            connection.setRequestMethod("HEAD");
            connection.connect();
```

```
            String type = connection.getContentType();

            if (type == null)
            {
                // type unknown
                NotSupportedException exception = new
                    NotSupportedException("type of file " + name + " is
                    unknown");
                notSupportedMap.put(name, exception);
                throw exception;
            }
            else if (type.contains("html"))
            {
                return readHTML(name);
            }
            else if (type.contains("javascript"))
            {
                return readJS(name);
            }
            else if (type.contains("css"))
            {
                return readCSS(name);
            }
            else if (type.contains("image"))
            {
                return readImage(name);
            }
            else if (type.contains("font"))
            {
                return readFont(name);
            }
            else
            {
                NotSupportedException exception = new
                    NotSupportedException("type (" + type + ") of file "
                     + name + " is not supported");
                notSupportedMap.put(name, exception);
                throw exception;
            }
        }
        catch (ClassCastException e)
        {
            // (HttpURLConnection) failed
            NotSupportedException exception = new NotSupportedException
                ("type of file " + name + " is unknown");
            notSupportedMap.put(name, exception);
            throw exception;
        }
    }
}


/**
 * Reads the HTML file with the given name and stores it in a file
 * named htmlx.html, where x is value of the HTML counter.
```

```
 *
 * @param name the name of the HTML file.
 * @return the name of the created HTML file.
 * @exception MalformedURLException if the given name is a malformed
    URL.
 * @exception IOException if an IO error occurred during reading or
    writing
 * the HTML file.
 */
private static String readHTML(String name) throws
   MalformedURLException, IOException
{
    String fileName;

    if (HTMLmap.containsKey(name))
    {
        fileName = directory + "html" + HTMLmap.get(name) + ".html";
    }
    else
    {
        URL url = new URL(encode(name));
        URLConnection connection = url.openConnection();
        connection.setRequestProperty("Cookie", cookies);
        connection.setRequestProperty("User-Agent", userAgent);
        connection.connect();

        InputStream stream = null;
        ReadableByteChannel streamChannel = null;
        FileOutputStream fileOutput = null;
        FileChannel fileChannel = null;

        try
        {
            stream = connection.getInputStream();
            streamChannel = Channels.newChannel(stream);
            fileName = directory + "html" + HTMLcounter + ".html";
            fileOutput = new FileOutputStream(fileName);
            fileChannel = fileOutput.getChannel();
            fileChannel.transferFrom(streamChannel, 0, Long.MAX_VALUE);

            HTMLmap.put(name, HTMLcounter);
            HTMLcounter++;

            if (verbose)
            {
                System.out.println("Downloading HTML file " + name + "
                   as " + fileName);
            }
        }
        finally
        {
            if (stream != null)
            {
                stream.close();
            }
```

```
            if (streamChannel != null)
            {
                streamChannel.close();
            }
            if (fileOutput != null)
            {
                fileOutput.close();
            }
            if (fileChannel != null)
            {
                fileChannel.close();
            }
        }
    }

    return fileName;
}

// counter used to name JavaScript files
private static int JScounter = 0;

// map from already localized JavaScript files to their counter
private static Map<String, Integer> JSmap = new HashMap<String, Integer
    >();

/**
 * Reads the JavaScript file with the given name and stores it in a
     file
 * named jsx.js, where x is value of the JavaScript counter.
 *
 * @param name the name of the JavaScript file.
 * @return the name of the created JavaScript file.
 * @exception MalformedURLException if the given name is a malformed
     URL.
 * @exception IOException if an IO error occurred during reading or
     writing
 * the JavaScript file.
 */
private static String readJS(String name) throws MalformedURLException,
     IOException
{
    String fileName;

    if (JSmap.containsKey(name))
    {
        fileName = directory + "js" + JSmap.get(name) + ".js";
    }
    else
    {
        URL url = new URL(encode(name));
        URLConnection connection = url.openConnection();
        connection.setRequestProperty("Cookie", cookies);
        connection.setRequestProperty("User-Agent", userAgent);
        connection.connect();
```

```
        InputStream stream = null;
        ReadableByteChannel streamChannel = null;
        FileOutputStream fileOutput = null;
        FileChannel fileChannel = null;

        try
        {
            stream = connection.getInputStream();
            streamChannel = Channels.newChannel(stream);
            fileName = directory + "js" + JScounter + ".js";
            fileOutput = new FileOutputStream(fileName);
            fileChannel = fileOutput.getChannel();
            fileChannel.transferFrom(streamChannel, 0, Long.MAX_VALUE);

            JSmap.put(name, JScounter);
            JScounter++;

            if (verbose)
            {
                System.out.println("Downloading JavaScript file " +
                   name + " as " + fileName);
            }
        }
        finally
        {
            if (stream != null)
            {
                stream.close();
            }
            if (streamChannel != null)
            {
                streamChannel.close();
            }
            if (fileOutput != null)
            {
                fileOutput.close();
            }
            if (fileChannel != null)
            {
                fileChannel.close();
            }
        }
    }

    return fileName;
}

/**
 * Writes the given content to a file named htmlx.js, where x is
 * value of the HTML counter.
 *
 * @param content the content of the HTML file.
 * @return the name of the created HTML file.
 */
private static String writeHTML(String content) throws
```

```
    FileNotFoundException
{
    String fileName = directory + "html" + HTMLcounter + ".html";
    HTMLcounter++;

    writeHTML(content, fileName);

    return fileName;
}

/**
 * Writes the given content to a file with the given name.
 *
 * @param content the content of the HTML file.
 * @param the name of the created HTML file.
 */
private static void writeHTML(String content, String name) throws
    FileNotFoundException
{
    PrintWriter output = new PrintWriter(name);
    output.println(content);
    output.close();

    if (verbose)
    {
        System.out.println("Writing HTML file " + name);
    }
}

// counter used to name CSS files
private static int CSScounter = 0;

// map from already localized CSS files to their counter
private static Map<String, Integer> CSSmap = new HashMap<String,
    Integer>();

/**
 * Reads the CSS file with the given name and stores it in a file
 * named cssx.css, where x is value of the CSS counter.
 *
 * @param name the name of the CSS file.
 * @return the name of the created CSS file.
 * @exception MalformedURLException if the given name is a malformed
    URL.
 * @exception IOException if an IO error occurred during reading or
    writing
 * the CSS file.
 * @exception FileNotFoundException if the CSS file cannot be found.
 */
private static String readCSS(String name) throws MalformedURLException
    , IOException
{
    String fileName;

    if (CSSmap.containsKey(name))
```

```
{
    fileName = directory + "css" + CSSmap.get(name) + ".css";
}
else
{
    URL url = new URL(encode(name));
    HttpURLConnection connection = (HttpURLConnection) url.
       openConnection();
    connection.setRequestProperty("Cookie", cookies);
    connection.setRequestProperty("User-Agent", userAgent);
    connection.connect();

    InputStream stream = null;
    PrintWriter output = null;

    try
    {
        stream = connection.getInputStream();

        // contents of the CSS file
        StringWriter writer = new StringWriter();
        IOUtils.copy(stream, writer);  // which encoding?
        StringBuffer contents = writer.getBuffer();

        // remove comments
        int begin = contents.indexOf("/*");
        while (begin != -1)
        {
            int end = contents.indexOf("*/");
            contents.delete(begin, end + 2);
            begin = contents.indexOf("/*", begin);
        }

        // read embedded URLs and modify contents
        int index = 0;
        do
        {
            index = contents.indexOf("url(", index + 1);
            if (index != -1)
            {
                // extract the URL
                begin = index + 4;
                int end = contents.indexOf(")", begin);
                String part = contents.substring(begin, end);

                if (part.length() > 0) // skip empty URLs
                {
                    if (part.charAt(0) == '"' || part.charAt(0) ==
                       '\'')
                    {
                        // remove " or ' at beginning and end
                        part = part.substring(1, part.length() - 1)
                           ;
                    }
```

```
                try
                {
                    if (part.startsWith("#"))
                    {
                        // skip
                    }
                    else
                    {
                        part = processURL(part, name);
                        String replacement = read(part);
                        contents.replace(begin, end,
                           replacement);
                    }
                }
                catch (NotSupportedException e)
                {
                    System.out.println("Warning: " + e.
                       getMessage());
                }
                catch (FileNotFoundException e)
                {
                    System.out.println("Warning: file " + e.
                       getMessage() + " could not be found");
                }
                catch (IllegalArgumentException e)
                {
                    System.out.println("Warning: URL " + part +
                        " could not be handled (" + e.
                       getMessage() + ")");
                }
                catch (IOException e)
                {
                    System.out.println("Warning: URL " + part +
                        " could not be handled (" + e.
                       getMessage() + ")");
                }
                catch (SkippedException e)
                {
                    // data URLs are skipped
                }
            }
        }
    }
    while (index != -1);

    // write the modified CSS file
    fileName = directory + "css" + CSScounter + ".css";
    output = new PrintWriter(fileName);
    output.println(contents.toString());

    CSSmap.put(name, CSScounter);
    CSScounter++;

    if (verbose)
    {
```

```
                System.out.println("Downloading CSS file " + name + "
                   as " + fileName);
            }
        }
        catch (FileNotFoundException e)
        {
            throw e;
        }
        catch (IOException e)
        {
            throw e;
        }
        finally
        {
            if (stream != null)
            {
                stream.close();
            }
            if (output != null)
            {
                output.close();
            }
        }
    }

    return fileName;
}

/**
 * Writes the given content to a file named jsx.js, where x is value of
     the
 * JavaScript counter.
 *
 * @param content the content of the JavaScript file.
 * @return the name of the created JavaScript file.
 */
private static String writeJS(String content) throws
   FileNotFoundException
{
    String fileName = directory + "js" + JScounter + ".js";
    JScounter++;

    PrintWriter output = new PrintWriter(fileName);
    output.println(content);
    output.close();

    if (verbose)
    {
        System.out.println("Writing a JavaScript file");
    }

    return fileName;
}

/**
```

```
 * Writes the given content to a file named cssx.css, where x is value
    of the
 * CSS counter.
 *
 * @param content the content of the CSS file.
 * @param base the URL of the HTML that contains the CSS.
 * @return the name of the created CSS file.
 * @throws FileNotFoundException if the file cannot be written.
 * @throws MalformedURLException if content contains any malformed URLs
    .
 * @throws IOException if something goes wrong with IO.
 */
private static String writeCSS(String content, String base) throws
   FileNotFoundException, MalformedURLException, IOException
{
    // read embedded URLs and modify contents
    StringBuilder copy = new StringBuilder(content);
    int index = 0;
    do
    {
        index = copy.indexOf("url(", index + 1);
        if (index != -1)
        {
            // extract the URL
            int begin = index + 4;
            int end = copy.indexOf(")", begin);
            String part = copy.substring(begin, end);
            if (part.charAt(0) == '"' || part.charAt(0) == '\'')
            {
                // remove " or ' at beginning and end
                part = part.substring(1, part.length() - 1);
            }

            try
            {
                part = processURL(part, base);
                String replacement = read(part);
                copy.replace(begin, end, replacement);
            }
            catch (NotSupportedException e)
            {
                System.out.println("Warning: " + e.getMessage());
            }
            catch (FileNotFoundException e)
            {
                System.out.println("Warning: file " + e.getMessage() +
                    " could not be found");
            }
            catch (IllegalArgumentException e)
            {
                System.out.println("Warning: URL " + part + " could not
                     be handled (" + e.getMessage() + ")");
            }
            catch (UnknownHostException e)
            {
```

```
                System.out.println("Warning: host of URL is unknown ("
                    + e.getMessage() + ")");
            }
            catch (SkippedException e)
            {
                // data URLs are skipped
            }
        }
    }
    while (index != -1);

    String fileName = directory + "css" + CSScounter + ".css";
    CSScounter++;

    PrintWriter output = new PrintWriter(fileName);
    output.println(copy);
    output.close();

    if (verbose)
    {
        System.out.println("Writing a CSS file");
    }

    return fileName;
}

// images that have been downloaded
private static Map<String, String> imageMap = new HashMap<String,
    String>();

/**
 * Dowmloads the image file with the given name.
 *
 * @param name the name of the image file.
 * @return the name of the created image file.
 * @exception MalformedURLException if the given name is a malformed
     URL.
 * @exception IOException if an IO error occurred during reading or
     writing
 * the image file.
 * @exception IllegalArgumentException if reading the image fails.
 */
private static String readImage(String name) throws
    MalformedURLException, IOException, IllegalArgumentException
{
    if (imageMap.containsKey(name))
    {
        return imageMap.get(name);
    }
    else
    {
        URL url = new URL(encode(name));

        // create directory
        String temp = name.substring("http://".length()); // remove
```

```
   http:// prefix
temp = temp.replaceAll("/", File.separator); // replace / with
   OS specific file separator
int separator = temp.indexOf(File.separator);
while (separator != -1)
{
    String path = temp.substring(0, separator);
    File file = new File(directory + path);
    file.mkdir();
    separator = temp.indexOf(File.separator, separator + 1);
}

// download the file
URLConnection connection = url.openConnection();
connection.setRequestProperty("Cookie", cookies);
connection.setRequestProperty("User-Agent", userAgent);
connection.connect();

InputStream stream = null;
ReadableByteChannel streamChannel = null;
FileOutputStream fileOutput = null;
FileChannel fileChannel = null;

try
{
    stream = connection.getInputStream();
    streamChannel = Channels.newChannel(stream);
    fileOutput = new FileOutputStream(directory + temp);
    fileChannel = fileOutput.getChannel();
    fileChannel.transferFrom(streamChannel, 0, Long.MAX_VALUE);
}
finally
{
    if (stream != null)
    {
        stream.close();
    }
    if (streamChannel != null)
    {
        streamChannel.close();
    }
    if (fileOutput != null)
    {
        fileOutput.close();
    }
    if (fileChannel != null)
    {
        fileChannel.close();
    }
}

imageMap.put(name, directory + temp);

if (verbose)
{
```

```
            System.out.println("Downloading image file " + (directory +
                temp));
        }

        return directory + temp;
    }
}

// fonts that have been downloaded
private static Map<String, String> fontMap = new HashMap<String, String
    >();

/**
 * Dowmloads the font file with the given name.
 *
 * @param name the name of the font file.
 * @return the name of the created font file.
 * @exception MalformedURLException if the given name is a malformed
     URL.
 * @exception IOException if an IO error occurred during reading or
     writing
 * the font file.
 * @exception IllegalArgumentException if reading the font fails.
 */
private static String readFont(String name) throws
    MalformedURLException, IOException, IllegalArgumentException
{
    URL url = new URL(encode(name));
    URLConnection connection = url.openConnection();
    connection.setRequestProperty("Cookie", cookies);
    connection.setRequestProperty("User-Agent", userAgent);
    connection.connect();

    // create directory
    name = name.substring("http://".length()); // remove http:// prefix
    name = name.replaceAll("/", File.separator); // replace / with OS
        specific file separator
    int separator = name.indexOf(File.separator);
    while (separator != -1)
    {
        String path = name.substring(0, separator);
        File file = new File(directory + path);
        file.mkdir();
        separator = name.indexOf(File.separator, separator + 1);
    }

    InputStream stream = null;
    ReadableByteChannel streamChannel = null;
    FileOutputStream fileOutput = null;
    FileChannel fileChannel = null;

    try
    {
        stream = connection.getInputStream();
        streamChannel = Channels.newChannel(stream);
```

```
            fileOutput = new FileOutputStream(directory + name);
            fileChannel = fileOutput.getChannel();
            fileChannel.transferFrom(streamChannel, 0, Long.MAX_VALUE);
        }
        finally
        {
            if (stream != null)
            {
                stream.close();
            }
            if (streamChannel != null)
            {
                streamChannel.close();
            }
            if (fileOutput != null)
            {
                fileOutput.close();
            }
            if (fileChannel != null)
            {
                fileChannel.close();
            }
        }

        return directory + name;
    }

    /**
     * Encodes the given URL.
     *
     * @param name the name of the URL.
     * @return the encoding on the URL.
     */
    private static String encode(String name)
    {
        return name.replaceAll(" ", "%20");
    }

    /**
     * Extracts the URLs from the given CSS import element.
     *
     * @param imports the CSS import element.
     * @return the URLs from the given CSS import element.
     */
    private static String[] extractImports(String imports)
    {
        String[] parts;
        if (imports.contains(";"))
        {
            parts = imports.split("(\\s)*;(\\s)*");
        }
        else
        {
            parts = new String[1];
            parts[0] = imports;
```

```
    }
    for (int i = 0; i < parts.length; i++)
    {
        if (parts[i].startsWith("@import"))
        {
            parts[i] = parts[i].substring("@import".length()).trim();
            if (parts[i].startsWith("url"))
            {
                parts[i] = parts[i].substring("url".length()).trim();
            }
            if (parts[i].charAt(0) == '(')
            {
                assert parts[i].charAt(parts[i].length() - 1) == ')' :
                    "Warning: not well formed CSS import";
                parts[i] = parts[i].substring(1, parts[i].length() - 1)
                    .trim();
            }
            assert parts[i].charAt(0) == '"' : "Warning: not well
                formed CSS import";
            assert parts[i].charAt(parts[i].length() - 1) == '"' : "
                Warning: not well formed CSS import";
            parts[i] = parts[i].substring(1, parts[i].length() - 1).
                trim();
        }
        else
        {
            assert false : "Warning: unsupported CSS import";
        }
    }

    return parts;
}
/**
 * Processes the given URL with respect to the given base URL.
 *
 * @param url the url to be processed.
 * @param host the base URL.
 * @return the processed url.
 * @throws MalformedURLException if the base is malformed.
 * @throws IllegalArgumentException if the url is empty.
 * @throws SkippedException if the url is a data URL.
 */
private static String processURL(String url, String base) throws
    MalformedURLException, IllegalArgumentException, SkippedException
{
    if (url.startsWith("data:"))
    {
        // skip data URIs
        throw new SkippedException("data URL");
    }
    else if (url.length() == 0)
    {
        throw new IllegalArgumentException("empty URL");
    }
    else
```

```
    {
        URL temp = new URL(new URL(base), url);
        return temp.toString();
    }
}

// iframes that have been downloaded
private static Map<String, String> iframeMap = new HashMap<String,
   String>();

public static void main(String[] args)
{
    if (args.length < 1)
    {
        System.out.println("Use: java Localize <properties file>");
    }
    else
    {
        InputStream stream = null;

        try
        {
            stream = new FileInputStream(args[0]);
            Properties properties = new Properties();
            properties.load(stream);

            String baseURL = properties.getProperty("url");

            userAgent = properties.getProperty("user.agent", "
               DEFAULT_USER_AGENT");
            if (USER_AGENT.containsKey(userAgent))
            {
                userAgent = USER_AGENT.get(userAgent);
            }

            cookies = properties.getProperty("cookies", DEFAULT_COOKIES
               );

            directory = properties.getProperty("directory",
               DEFAULT_DIRECTORY);
            int separator = directory.indexOf(File.separator);
            while (separator != -1)
            {
                String path = directory.substring(0, separator);
                File file = new File(path);
                file.mkdir();
                separator = directory.indexOf(File.separator, separator
                    + 1);
            }

            verbose = Boolean.parseBoolean(properties.getProperty("
               verbose", "false"));

            String HTMLfile = readHTML(baseURL);
```

```
File file = new File(HTMLfile);
Document document = Jsoup.parse(file, "UTF-8");

// process all links to external JavaScript files
Elements elements = document.select("script[src]");
for (Element element : elements)
{
    // extract the URL of the JavaScript file
    String javaScriptURL = element.attr("src");
    javaScriptURL = processURL(javaScriptURL, baseURL);

    try
    {
        // download the JavaScript file
        String JSfile = readJS(javaScriptURL);

        // modify the HTML
        element.attr("src", JSfile);
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Warning: file " + javaScriptURL
            + " cannot be found");
    }
    catch (IOException e)
    {
        System.out.println("Warning: something went wrong
           with reading file " + javaScriptURL);
    }
}

// process all JavaScript embedded in the HTML
elements = document.select("script");
for (Element element : elements)
{
    String data = element.data();

    // write JavaScript to a file
    if (data != null && data.length() != 0)
    {
        String JSfile = writeJS(data);

        // modify the HTML
        element.attr("src", JSfile);
        element.attr("type", "text/javascript");
        element.text("");
    }
}

// process all links to external CSS files
elements = document.select("link");
for (Element element : elements)
{
    String type = element.attr("type");
    String rel = element.attr("rel");
```

```
    if (type.equals("text/css") || rel.equals("stylesheet")
       )
    {
        String CSSURL = element.attr("href");
        CSSURL = processURL(CSSURL, baseURL);

        try
        {
            String CSSfile = readCSS(CSSURL);

            // modify the HTML
            element.attr("href", CSSfile);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Warning: file " + CSSURL +
               " cannot be found");
        }
    }
}

// process all CSS embedded in the HTML
elements = document.select("style");
for (Element element : elements)
{
    String type = element.attr("type");
    if (type.equals("text/css"))
    {
        String data = element.data().trim();

        // write CSS to a file
        if (data != null && data.length() != 0)
        {
            if (data.startsWith("@import"))
            {
                // process link to external CSS files
                for (String CSSURL : extractImports(data))
                {
                    CSSURL = processURL(CSSURL, baseURL);

                    try
                    {
                        String CSSfile = readCSS(CSSURL);

                        // add to the HTML
                        Element node = new Element(Tag.
                           valueOf("link"), "");
                        node.attr("type", "text/css");
                        node.attr("rel", "stylesheet");
                        node.attr("href", CSSfile);
                        element.after(node);
                    }
                    catch (FileNotFoundException e)
                    {
                        System.out.println("Warning: file "
```

```
                                + CSSURL + " cannot be found");
                        }
                    }
                    // remove from the HTML
                    element.remove();
                }
                else
                {
                    String CSSfile = writeCSS(data, baseURL);

                    // modify the HTML
                    Element replacement = new Element(Tag.
                       valueOf("link"), "");
                    replacement.attr("type", "text/css");
                    replacement.attr("rel", "stylesheet");
                    replacement.attr("href", CSSfile);
                    element.replaceWith(replacement);
                }
            }
        }
    }

    // process all the images
    elements = document.select("img");
    for (Element element : elements)
    {
        String imageURL = null;
        try
        {
            imageURL = element.attr("src");
            imageURL = processURL(imageURL, baseURL);

            String imageFile;
            if (imageMap.containsKey(imageURL))
            {
                imageFile = imageMap.get(imageURL);
            }
            else
            {
                imageFile = readImage(imageURL);
                imageMap.put(imageURL, imageFile);
            }

            // modify the HTML
            element.attr("src", imageFile);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Warning: image file " +
               imageURL + " could not be found");
        }
        catch (MalformedURLException e)
        {
            System.out.println("Warning: URL " + imageURL + "
               is malformed");
```

```
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Warning: URL " + imageURL + "
            could not be handled (" + e.getMessage() + ")");
    }
    catch (SkippedException e)
    {
        // data URLs are skipped
    }
    catch (IOException e)
    {
        System.out.println("Warning: IO failure when
            reading image file " + imageURL);
    }
}

// process all iframes
elements = document.select("iframe");
for (Element element : elements)
{
    String iframeURL = null;
    try
    {
        iframeURL = element.attr("src");
        iframeURL = processURL(iframeURL, baseURL);

        String iframeFile;
        if (iframeMap.containsKey(iframeURL))
        {
            iframeFile = iframeMap.get(iframeURL);
        }
        else
        {
            iframeFile = readHTML(iframeURL);
            iframeMap.put(iframeURL, iframeFile);
        }

        // modify the HTML
        element.attr("src", iframeFile);
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Warning: image file " +
            iframeURL + " could not be found");
    }
    catch (MalformedURLException e)
    {
        System.out.println("Warning: URL " + iframeURL + "
            is malformed");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Warning: URL " + iframeURL + "
            could not be handled (" + e.getMessage() + ")");
```

```
                    }
                    catch (SkippedException e)
                    {
                        // data URLs are skipped
                    }
                }

                // write modified HTML
                writeHTML(document.toString(), directory + "index.html");
            }
            catch (MalformedURLException e)
            {
                System.out.println("URL is malformed: " + e.getMessage());
                e.printStackTrace();
            }
            catch (IOException e)
            {
                System.out.println("IO failed: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }
}
```

Appendix B

Fly.js Source Code

```
(function (sandbox) {

    function Fly() {

        var missingURLs = {};
        var existingURLs = {};
        var urlSent = false;

        // checks if a URL exists on the localhost
        function UrlExists(url) {
            var http = new XMLHttpRequest();
            http.open('HEAD', url, false);
            try { http.send(); } catch (e) { return false; }
            return http.status != 404;
        }

        // detects "r.url" usage after y.send() method is recognised,
        // and record a missing URL if it does not exist on the localhost
        this.readPre = function (iid, name, val, global) {

            if (urlSent && name == "r" && val == "[object Object]") {
                urlSent = false;

                if (val.url != undefined && val.url) {
                    var sourceUrl = val.url;
                    var lh = "http://localhost/tests/uken/game";

                    console.log("JALANGI: reading URL " + val.url + ". Type
                        = " + val.type);
                    if (sourceUrl.indexOf(lh) != -1) sourceUrl = sourceUrl.
                       replace(lh, "");

                    // construct a basic key for the URL
                    var urlKey = sourceUrl;
                    var qmi = urlKey.indexOf('?');
                    if (qmi > -1) urlKey = urlKey.substr(0, qmi);

                    if (sourceUrl.indexOf("http") < 0 && sourceUrl.indexOf
                        ("localhost") < 0) {
                        sourceUrl = "http://m.staging.crimeinc.uken.com" +
```

```
                    sourceUrl;
            }

            // download the missing URL (reload_partial does not
               work this way)
            if (!missingURLs[urlKey] && val.url.indexOf("http://
               localhost") != -1 && !existingURLs[urlKey]) {
                console.log("JALANGI: Checking " + sourceUrl);
                if (!UrlExists(val.url)) {
                    missingURLs[urlKey] = sourceUrl;
                    console.log("JALANGI: Missing URL found. Key =
                       " + urlKey + ". Downloading " + sourceUrl);
                }
                else {
                    existingURLs[urlKey] = sourceUrl;
                    console.log("JALANGI: URL found on the
                       localhost " + val.url);
                }
            }
        }
    }

}

// trace ajax() method invoking y.send() statement
this.invokeFunPre = function (iid, base, f, args, isConstructor) {
    if (base == "function send() { [native code] }" &&
         f == "[object XMLHttpRequest]") {
        urlSent = true;
    }
}

// detect "r.url = ..." and redirect it to the localhost if not
   already
this.putFieldPre = function (iid, base, offset, val) {

    if (base.constructor.name == "Object" && offset == "url" &&
        base.hasOwnProperty("url") && base.hasOwnProperty("type"))
    {
        var newUrl =  val;

        // dealing with file system addresses that start with file
           ://
        if (newUrl.indexOf("file://") > -1)
        {
            newUrl = newUrl.replace("file://localhost", "");

            // localhost final directory
            var lh = "/jalangi";

            // rewriting links to JALANGI source files (e.g, file
               ://localhost/Library/WebServer/jalangi/src/js to /
               src/js)
            // apparently, JALANGI instrumentation phase adds
               source files to .html header
```

```
                var ilh = newUrl.indexOf(lh);
                if (ilh > - 1) {
                    newUrl = newUrl.replace(newUrl.substring(0, ilh +
                        lh.length), "");
                }
            }

            // link instrumented files to the http://localhost/tests/
                game/ directory
            if (val.indexOf(lh) < 0) newUrl = "/tests/game" + newUrl;

            newUrl = "http://localhost" + newUrl;

            // check if the URL is a default URL
            var qmi = newUrl.indexOf("?");
            var defUrl = newUrl.substr(0, qmi) + "/" + newUrl.substr(
                qmi);
            if (qmi > -1 && UrlExists(defUrl)) {
                console.log("JALANGI: DEFAULT redirecting " + val + "
                    to " + defUrl);
                return defUrl;
            }

            console.log("JALANGI: redirecting " + val + " to " + newUrl
                );
            return newUrl;
        }

        return val;
    }

    // Saves a linked file to a hard drive
    // http://muaz-khan.blogspot.ru/2012/10/save-files-on-disk-using-
        javascript-or.html
    function SaveToDisk(fileURL, fileName) {
        var save = document.createElement('a');
        save.href = fileURL;
        save.target = '_blank';
        save.download = fileName || fileURL;
        var evt = document.createEvent('MouseEvents');
        evt.initMouseEvent('click', true, true, window, 1, 0, 0, 0, 0,
            false, false, false, false, 0, null);
        save.dispatchEvent(evt);
        (window.URL || window.webkitURL).revokeObjectURL(save.href);
    }

    // downloads the missing URLs (only works in Firefox)
    this.endExecution = function() {
        for (var key in missingURLs) {
            console.log("JALANGI: Saving " + key); SaveToDisk(key);
        }
    }
}

sandbox.analysis = new Fly();
```

```
    if (sandbox.Constants.isBrowser) {
        window.addEventListener('keydown', function (e) {
            // keyboard shortcut is Alt-Shift-T for now
            if (e.altKey && e.shiftKey && e.keyCode === 84) sandbox.
               analysis.endExecution();
        });
    }
}(J$));
```

Bibliography

[1] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of Javascript Web Applications. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering*, pages 571–580, Waikiki, Honolulu, HI, USA, May 2011. ACM.

[2] Kevin Curran and Ciaran George. The future of web and mobile game development. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 1(1):25–34, 2012.

[3] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2002.

[4] Gong, Liang and Nguyen, Cuong. A Shadow Execution and Dynamic Analysis Framework for LLVM IR and JavaScript. 2013.

[5] ECMA International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[6] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[7] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. 45(6):1–12, 2010.

[8] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. pages 488–498, 2013.

[9] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE'05*. ACM, September 2005.