UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Information Technology

Risko Ruus

Hunting Down Easter Eggs Online by Exploiting Cross-browser Compatibility Issues: The Konami Code Experiment

Bachelor's thesis (6 EAP)

Supervisor: Peep Küngas, PhD

Author: …………………......…………………......“…..“ June 2011

Supervisor: ………......…………………………...... “…..“ June 2011

Approved for defense:

Professor: …………………..…………………...... “…..“ June 2011

TARTU 2011

Abstract

The *Konami Code* is an input combination (↑ ↑ ↓ ↓ ← → ← → B A) used initially in a 1986 video game called *Gradius* for the *Nintendo Entertainment System*. It was left there accidentally by the developer, who used the code during testing to give the player all the power-ups. The players discovered it, shared it and since then it has been featured in many sequels and other video games. Today there exist several Web pages that reveal an Easter egg when a visitor inputs the Konami Code through its computer's keyboard. Typically the Easter eggs are implemented using AJAX, which means that after the Konami Code is entered, Document Object Model of the Web page is modified to reveal the Easter egg. However, since many Web sites suffer from Cross-Browser Compatibility (CBC) issues, the code might not function properly on all Web browsers. By using the Konami Code scenario as a test case, we show how such CBC issues can be exploited using our tool capable of automatically detecting Konami Codes from Web sites. In the case study we apply our program, which uses *WebDriver* framework, to a list of the world's most popular Web sites. Our goal is to identify their CBC issues with Mozilla Firefox, Internet Explorer and Google Chrome Web browsers with respect to the Konami Code. By exploiting our program on the particular test scenario, we are not only capable of identifying Web sites which use Konami Code to reveal Easter eggs, but also demonstrate that our method could be used for reporting functional CBC issues on Web sites.

Table of Contents

1. Introduction

Konami, a Japanese video game company was developing a port for the *Nintendo Entertainment System* (NES) of *Gradius* (Figure 1), a “side-scrolling shooter”[1] released initially in 1985 as an arcade game. Kazuhisa Hashimoto, a developer who was working on the port for NES, found the gameplay too challenging during testing and decided to implement a cheat code (↑ ↑ ↓ ↓ ← → ← → B A) to give the player’s space ship all power-ups. He accidentally forgot to remove the cheat from the game’s source code and it ended up in the final release version. It was discovered and shared among the players and since then it has been featured in many sequels and other video games. This input combination is known as the *Konami Code* and it was popularized by another classic Konami game, *Contra*, released for the NES in 1988, which the players found too difficult to complete and used the code to get an extra 30 lives[2].

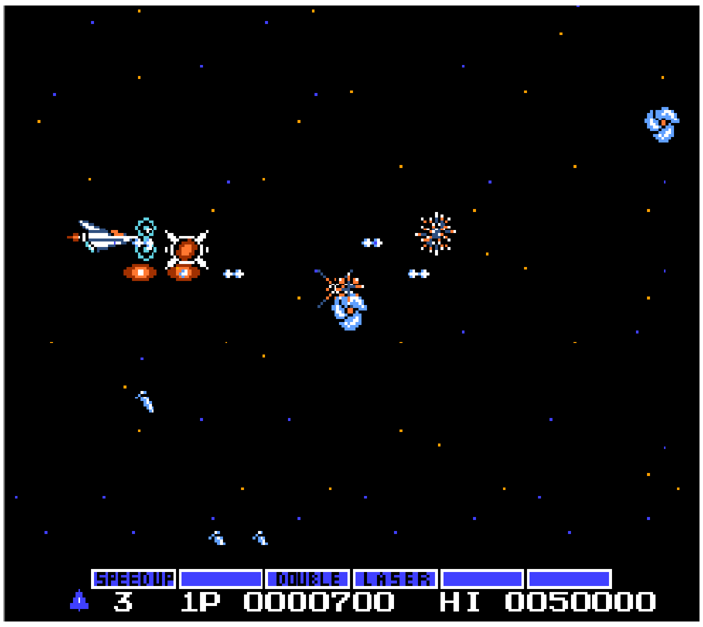


Figure 1: Screenshot from Gradius 1986 NES port with different ship power-ups

References to Konami Code have appeared in various places. For example Konami Code can be used to get root access in the *Palm Pre* smartphone[3] and typing the code on *patrickacarrell.com* will display the player character animation from *Contra* as a hidden Easter egg (Figure 2). The Easter egg on *patrickcarrell.com* is displayed correctly using

[1] http://en.wikipedia.org/wiki/Shoot_%27em_up
[2] http://en.wikipedia.org/wiki/Konami_Code
[3] http://www.engadget.com/2009/06/10/the-secret-to-palm-pre-dev-mode-lies-in-the-konami-code/

Mozilla Firefox 3.6 and *Google Chrome 10*, however, the code has no effect when viewing the page with *Internet Explorer 8*.

This thesis will focus on the automatic detection of Web sites, which use Konami Code, by exploiting the Cross-browser Compatibility (CBC) issues using these versions of Mozilla Firefox, Internet Explorer and Google Chrome Web browsers. We try to evaluate how many of the codes that we are able to detect are also Cross-browser compatible. Konami Code was chosen as an example to demonstrate the highly dynamic nature of the modern Web which is now powered by extensive client side scripting. With every user interaction, the Document Object Model (DOM) [4] of the Web page might be changed to display modified data to the visitor. *WebDriver*[5] is a framework for automated testing of Web applications, which has now been merged into the *Selenium 2.0* [6] project. We use WebDriver to automatically launch the Web browser and send the Konami Code combination to a list of 100 000 world's most popular Web-sites. The changes of DOM are stored and analyzed.

This thesis presents an algorithm to analyze dynamically changing browser behavior and to filter out sites likely to implement the Konami Code. Major changes made to the Web page DOM, like in sites which use the Konami Code, are also common for AJAX-based Web applications. We give a brief overview how AJAX works, how AJAX is automatically tested by current tools and how it could be tested using newly developed methods.

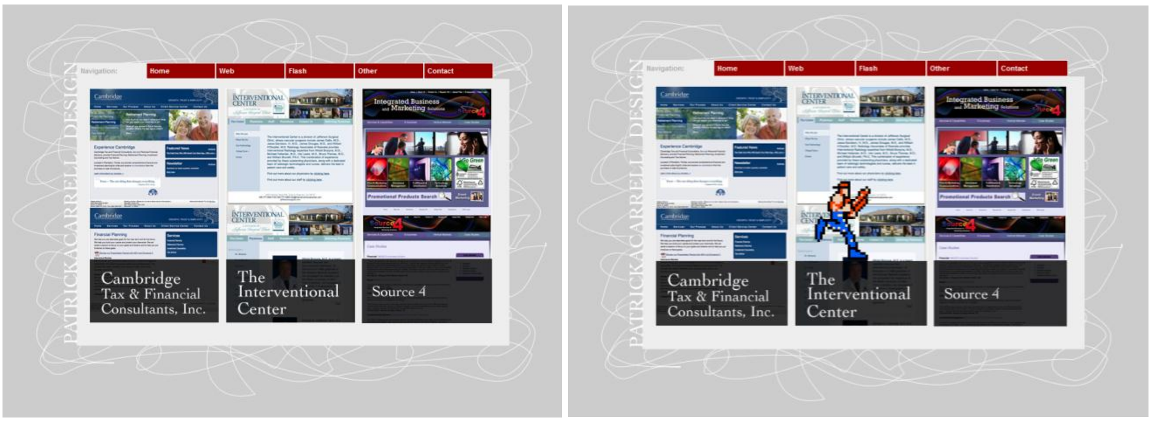

PATRICKACARRELL DESIGN
Navigation:
Home
Web
Flash
Other
Contact
Cambridge Tax & Financial Consultants, Inc.
The Interventional Center
Source 4

Figure 2: *patrickacarrell.com* before and after typing the code

[4] http://www.w3.org/DOM/
[5] http://code.google.com/p/selenium/wiki/GettingStarted
[6] http://seleniumhq.org/

2. Background

2.1. The Difference Between Classic and Modern Web 2.0 Applications

Traditional Web applications such as described in (Ricca, et al., 2001), have been forced to use a multi-page user interface connected by URLs since HTML was not designed with interactive Graphical User Interfaces (GUI) in mind. The user interacts with the application by using synchronous client request and server response interactions like represented in top half of Figure 3. The load times are very high after each user interaction since the whole Web page has to be reloaded. For example unchanged data is being transmitted each time a user completed form is submitted to a Web server and also when the Web server responds and sends the page back. This prevents the creation of more complex and interactive user-interfaces since the systems are not responsive enough. Usually there is not any client-side application processing and therefore the browser is inactive for most of the time and the processing of data is done by the Web server.

The Rich Internet Application (RIA) term was first introduced in (Allaire, 2002) to describe Web applications that would be able to include modern user interfaces. All RIAs use a layer of code between the user inputs and server queries. This layer is responsible for rendering of the Web user interface and for the communication with the server. Most of the RIA technologies, like *JavaFX*, *Microsoft Silverlight* and *Adobe AIR* use specific interpreters, which must be installed to a browser before the user can use the RIA features of the Web page. Usually these plugins are proprietary and therefore non-standard.

The term Web 2.0 (O'Reilly, 2005) was proposed to describe the evolution from read-only static Web pages into highly-interactive browsing experience, where users could create and modify the page content themselves. Notable examples of such Web pages include:

- *Wikipedia*[7] – free online encyclopedia;
- Photo sharing sites like *Flickr*[8] and *Picasaweb*[9]*;*
- Social Networking sites like *MySpace*[10] and *Facebook*[11]*;*

[7] http://www.wikipedia.org/
[8] http://www.flickr.com/
[9] http://picasaweb.google.com/

- Web services that are similar to their desktop counterparts (e.g. a text editor *Google Docs*[12], a photo editing tool *Pixlr*[13]).

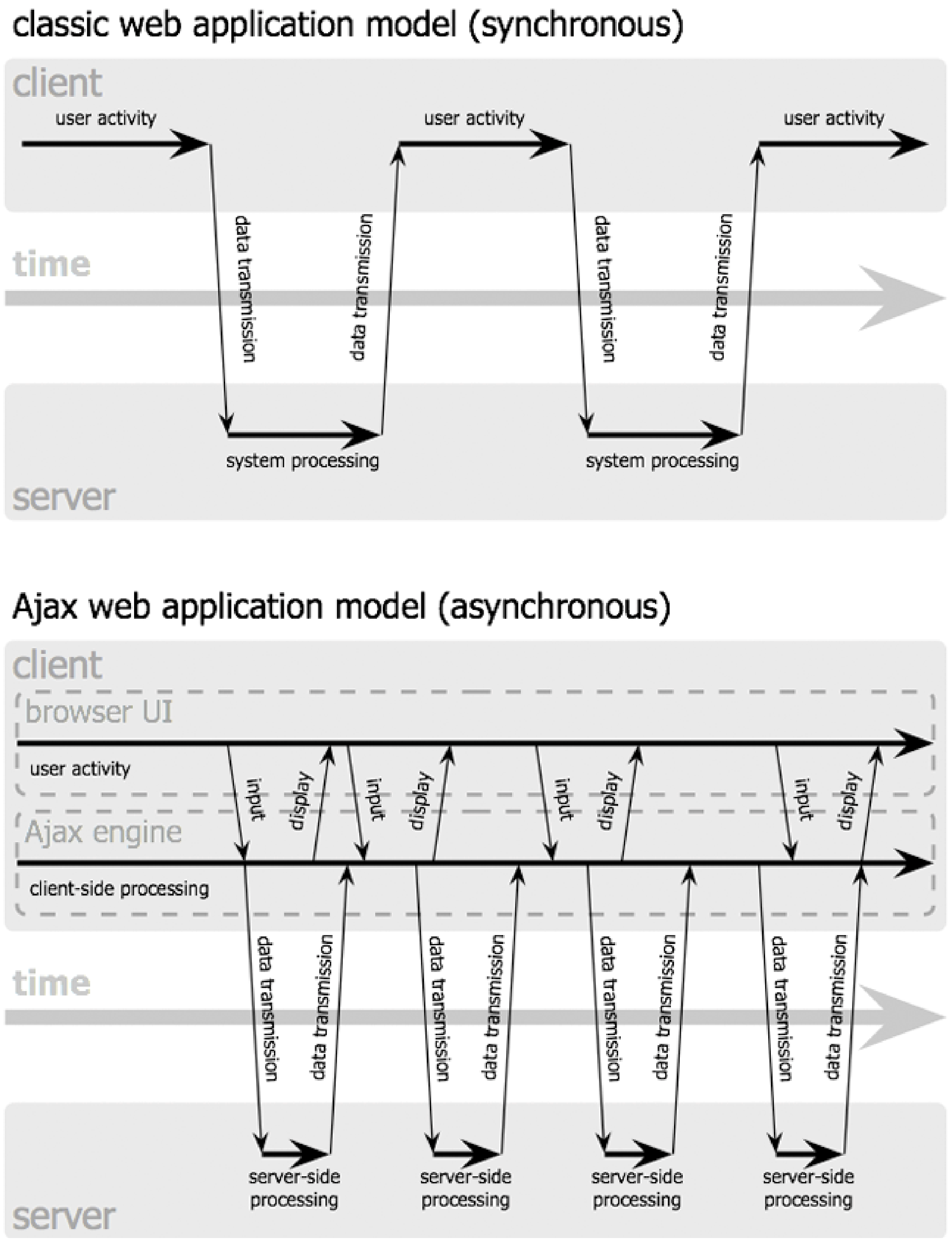


Figure 3: Synchronous and asynchronous communication pattern interaction[14]

[10] http://www.myspace.com/
[11] http://www.facebook.com/
[12] https://docs.google.com
[13] http://pixlr.com/
[14] http://www.adaptivepath.com/publications/essays/archives/000385.php

2.1. Cross-Browser Compatibility Issues

It is widely known that Web browsers tend to render the page content differently (Rode, et al., 2002), (Ricca, et al., 2005). The market share of Internet Explorer 6, which is known to account for a lot of CBC issues, is still as high as 10.97%[15]. Recent trends point to the rising popularity of tablet personal computers and smartphones. Rise in cross-platform variety results also in the rise of cross-browser compatibility issues.

The way the JavaScript code is handled by different Web browsers is one of the main causes for CBC issues. A list of how DOM events are supported by different browser vendors can be seen from *QuirksMode.org*[16]. To the best of our knowledge, at the time of writing this thesis no tool that is capable of automatically detecting functional cross-browser defects has been officially released. Although some studies (Rode, et al., 2002) show that there exists a real need for a tool like this. (Marchetto, et al., 2009) found in their Web fault classification experiment that CBC issues make the most populated class of faults in Web development projects. (Mesbah, et al., 2011) have proposed and implemented a tool that is directed towards solving this problem, however, they have not released it to the public yet. Currently most of the tools like *Browsershots*[17] and *BrowserCam*[18] are focused primarily on solving the CBC layout and appearance issues.

In modern Web applications JavaScript is used to modify HTML DOM extensively and this may result in malformed HTML. In (Artzi, et al., 2008) it is pointed out that malformed HTML is not always portable across all Web browsers. Normally browsers can successfully handle malformed HTML, but while trying to automatically compensate the outcome might result in failures. A good demonstration of this effect can be observed on a Web site *crashie8.com*[19] and also by numerous bugs in the Mozilla bug repository[20].

2.2. AJAX

AJAX - Asynchronous JavaScript and XML (Garrett, 2005) is a set of technologies used in a clever way to create more responsive and user-friendly dynamic Web applications. The

[15] http://marketshare.hitslink.com/browser-market-share.aspx?qprid=2
[16] http://www.quirksmode.org/dom/events/index.html
[17] http://browsershots.org/
[18] http://www.browsercam.com/
[19] http://crashie8.com – Must use IE6 – IE9. Works fine with Firefox 4 for example
[20] See defects: 269095, 320459, and 328937 at https://bugzilla.mozilla.org/show_bug.cgi?

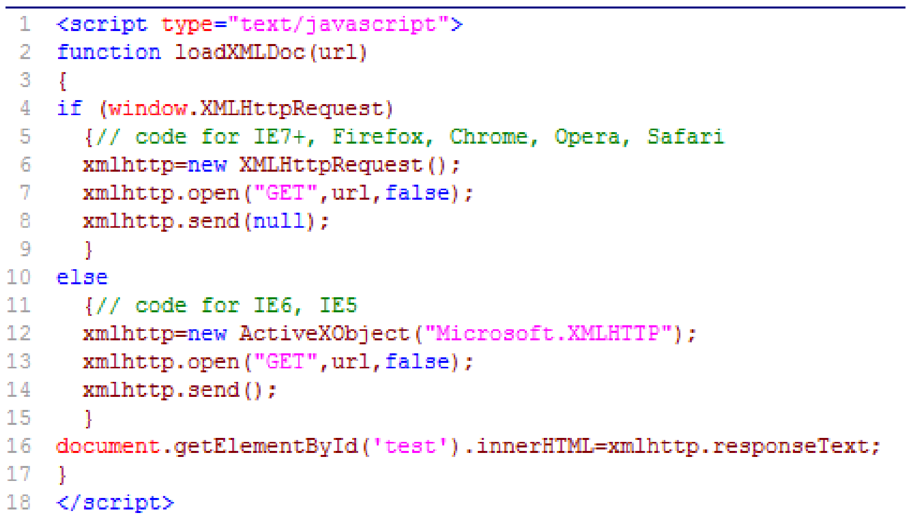
main benefit of this approach is that AJAX is not server platform dependent. One can write the client behavior in JavaScript and have it easily communicate with PHP, Java, .NET and other server side programming languages. Modern Web browsers support all the necessary Web standards that are required to develop RIA using AJAX.

2.2.1. Usage

RIAs are Web applications which are similar to their desktop counterparts. RIA users are not limited to static read-only pages, but are encouraged by the technology to provide their own custom input and create new content. AJAX provides modern Web pages with a more interactive and responsive feel by using JavaScript for asynchronous client-server communication to change the DOM. This method provides more powerful and complex user-interfaces when compared to the Web pages where synchronous request-response method is used to provide dynamic behavior requiring an entire page reload. By using delta communication (Mesbah, et al., 2008) only small parts of a Web page can be separately updated without the need for a full page reload. This results in faster loading times and decreased bandwidth usage.

2.2.2. Technology

The basic components of AJAX are: HTML and CSS, DOM, `XMLHttpRequest`, XML and JavaScript. HTML and CSS are used to present the information and DOM is required to access and change the presented data. The key of AJAX lies in the `XMLHttpRequest` object (Figure 4), which can be accessed in JavaScript to interact with the Web server. The data returned by the Web server can be used to change the browser's DOM without the need of a full Web page reload.



```
<script type="text/javascript">
function loadXMLDoc(url)
{
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  xmlhttp.open("GET",url,false);
  xmlhttp.send(null);
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  xmlhttp.open("GET",url,false);
  xmlhttp.send();
  }
document.getElementById('test').innerHTML=xmlhttp.responseText;
}
</script>
```

Figure 4: Example JavaScript code demonstrating `XMLHttpRequest` usage

AJAX enables developers to use asynchronous communication, instead of the traditional synchronous communication where a request is sent to the server and a response is returned after the server has finished processing. Figure 3 illustrates the main differences between synchronous and asynchronous communication. Traditional *click-and-wait* style of navigating the Web pages can be therefore significantly changed. Requests can be sent to a Web server without any visible change to the user and without stopping the component execution because AJAX engine is handling the HTTP requests and responses in the background by utilizing an event listener. Developers can use different event handlers to change the component state in the client-side whenever an asynchronous response is received.

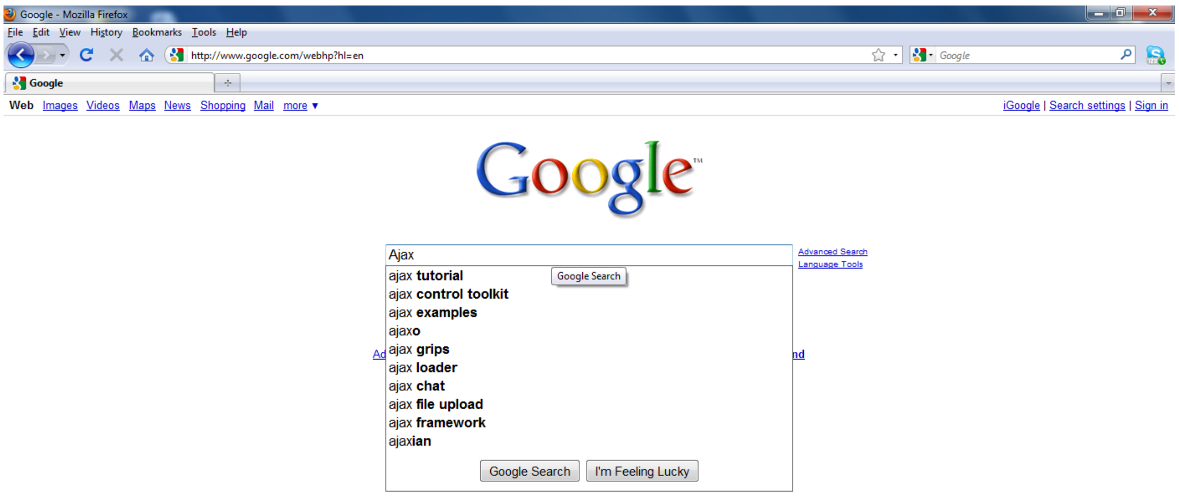


Figure 5: An example screen shot of Google Suggest

By using AJAX, separate elements can be updated independently from an entire Web page update. Every element may be linked with an event listener and once a valid action is performed a server request is generated, response is retrieved and a particular element is updated. Figure 5 of Google Suggest illustrates this behavior. As the user types in the search words, a query is performed in the background and the element is updated without the full reload of the entire Web page.

2.2.3. Automated Testing of AJAX Applications

A lot of Web applications that fit into the Web 2.0 definitions, are based on AJAX and therefore it is vital to know how to test these rapidly growing applications. According to the study (Torchiano, et al., 2009) Web applications contain 35% more defects in the application layer than their desktop counterparts. The more interactive behavior of an AJAX Web page is to more likely to contain errors, since the data is being transmitted asynchronously between the client and the server, and the DOM tree is being extensively modified. Furthermore, JavaScript is a weakly typed programming language and because of this, fewer errors are caught during compile time.

The testing approach of classical Web applications was to issue a request for a response from the server by using a hyperlink and then verify the resulting HTML page. This technique, however, is not suitable for AJAX Web pages where only a single page URL could be used for exposing the functionality behind the application and everything else on the client side is implemented through DOM tree manipulation. The Web page might even never be entirely reloaded. General testing techniques can be used to test the server-side of AJAX applications and tools like *JsUnit* [21] can be used to create functional tests for the client-side JavaScript code. However, complementary testing techniques, tailored specifically for AJAX, are required to find bugs that are hard to find using existing Web testing techniques as suggested in (Marchetto, et al., 2008).

Currently testing tools like *Selenium IDE* [22], and *Sahi* [23] are used to capture and replay test cases of AJAX applications. Unfortunately, such tests are time-consuming and labor-intensive since a tester needs to plan, capture and maintain the test cases. Web applications can be tested by modeling with Finite State Machines (Andrews, et al., 2005) and in (Marchetto, et al., 2008) a way is proposed to reduce the manual labor of writing and maintaining test cases. This is done by extracting the finite-state model of an AJAX application and by using semantically interacting events to automatically derive test cases for the capture and replay tools. A method used in (Wang, et al., 2008) focuses on static analysis of the Web site source code ignoring the application's client-side behavior, which we know is vital to test in AJAX applications. To perform automatic testing of Web applications (Benedikt, et al., 2002) have combined the Web spider and the automatic filling of forms during test execution. This technique however is outdated since common Web spiders are only capable of crawling the Web site by following URLs. Anyway, a

[21] http://www.jsunit.net/
[22] http://seleniumhq.org/projects/ide/
[23] http://sahi.co.in/w/

new Web spider *Crawljax*[24] (Mesbah, et al., 2008), capable of crawling and testing AJAX-powered Web sites, has been developed. Besides following URLs it is capable of navigating through the Web site by identifying clickable elements (which may change with every state change) that modify the state within the browser's DOM.

2.2.4. WebDriver

WebDriver is a framework that can be used to carry out automatic testing of Web applications in an actual browser window. WebDriver provides support for natural user actions like "click", "hover" etc., and can be used with popular testing frameworks like *JUnit* [25] and *TestNG* [26].

WebDriver has currently support for the following drivers:

- Mozilla Firefox (2 and above);
- Microsoft Internet Explorer (6 and above);
- Google Chrome (4.0 and above);
- Opera (released by Opera themselves)[27];
- HtmlUnit (a GUI-less browser)[28];
- AndroidDriver[29] (uses the RemoteWebDriver)[30] (2.3 and above);
- iPhone and iPad (uses UIWebView)[31] (iOS 3.2 and above).

[24] http://crawljax.com/
[25] www.junit.org/
[26] http://testng.org/doc/index.html
[27] https://github.com/operasoftware/operadriver
[28] http://htmlunit.sourceforge.net/
[29] http://code.google.com/p/selenium/wiki/AndroidDriver
[30] http://code.google.com/p/selenium/wiki/RemoteWebDriver
[31] http://code.google.com/p/selenium/wiki/IPhoneDriver

3. The Konami Code Experiment

Konami Code is an input combination that can be used in a Web page, which reveals a hidden secret, left there by the developer for the users to find. Finding an Easter egg by accident is highly unlikely given that the code combination (↑ ↑ ↓ ↓ ← → ← → BA) is quite long and complex. Back in the eighties a NES game controller had only arrow keys, A and B buttons, plus an additional START and SELECT button, which were not used to play an actual game, but were meant to start/stop the game and navigate in game start menus. Computer keyboards however have a lot more keys and that makes finding Konami Codes in Web sites by chance highly unrealistic.

Several Web sites describe how Konami Code can be integrated to a Web page[32] [33]. There even exists an implementation for the iPhone, but since the iPhone does not have actual keys, the code uses directional gestures on the device screen[34]. Most of the sites currently known to contain Konami Codes have been found by people called Konami Code "hunters" or Easter egg "hunters". They are mostly retro gamers who are aware of this input sequence and sometimes try the code on some random Web site to see if it has any effect or not. For example a popular sports news page *ESPN* contained an Easter egg[35] that made unicorns pop-up on their Web site[36], but the owners of the site soon removed this secret content after they became aware of it. It was probably left there on purpose by one of the site's Web developers. Sometimes the developers themselves have left little hints for the "hunters", informing the visitor that the site may contain a Konami Code. An example of this is a Web page *Konami Code Sites*[37] which home page contains only text "Perform the Konami Code to access this website". Only people who know the combination can get access to a list of 91 collected URLs that contain Konami Code Easter eggs.

3.1. The Idea

So far the lists of known Konami Codes have been created manually by collecting hints from Konami Code enthusiasts. We decided to create a systematic process for finding Konami Codes automatically. Some sites may require a slightly different combination than the traditional Konami Code like typing an additional Enter key after the code or require

[32] http://snaptortoise.com/konami-js/
[33] http://www.yourinspirationweb.com/en/fun-with-javascript-jquery-and-konami-code/
[34] http://www.youtube.com/watch?v=qZyqpteOTUs
[35] http://www.youtube.com/watch?v=DuGubWnjiPA&feature=related
[36] http://www.joystiq.com/2009/04/27/konami-code-turns-espn-com-into-a-lisa-frank-wonderland/
[37] http://konamicodesites.com

typing Konami Code three times in a row. Our approach will currently focus only on automatic detection of Web sites that use the traditional Konami Code.

The idea for automatic Konami Code detection is based on checking if the Web page DOM has changed after the code input - if it has, then we probably have found a page which includes a Konami Code. To automatically detect pages revealing an Easter egg after insertion of Konami Code, a Web browser must be used for inserting the code and inspecting changes in DOM. For example using *`wget`* to download page source files and later doing static analysis on them might have little effect in our settings since the JavaScript code might be included in a library and the code would not be simply visible for inspection. Therefore it is necessary to observe the runtime behavior of the Web page under test so that the JavaScript gets executed by the browser's JavaScript engine.

To automate this process we decided to write our own program in Java and picked WebDriver to control the Web browser for the following reasons:

- WebDriver supports interaction with actual browser windows;
- WebDriver has support for different Web browsers, which makes it a perfect candidate for CBC testing;
- WebDriver is written in Java and its API is straightforward and easy to use.

3.2. Algorithm

Considering the Konami Code length, we were worried that the DOM might change already in the midway of the input. In order to reduce the amount of false positives, we decided to implement an algorithm that checks for changes in the DOM after only parts of the Konami code have been typed. This approach has its strengths and weaknesses. By sending only the arrow input first and then each letter separately we can ensure that the DOM does not change midway. The arrow or character "B" input might trigger the execution of JavaScript which might change the HTML DOM. If we had typed the code in one sequence and had checked the page DOM only before and after the input, we would have had no idea if the whole sequence or only parts of the code had been the cause of the DOM change. Algorithm 1 describes the way our program works.

Algorithm 1: *`detectKonamiCode`*`(`*`inputFile`*`)`**:**
Input: *`inputFile`* contains a list of URLs that need to be tested
Output: result for each URL whether Konami Code was found or not

```
for i <- 1 to inputFile.length do
      URL <- inputFile[i]
      launchPage(URL)
      initialDOM <- currentDOM
      element <- findElement(By.xpath(„*‟))
      type(arrows(element))
      afterTypingArrowsDOM <- currentDOM
            if initialDOM = afterTypingArrowsDOM then
                  type(‘B’(element))
                  afterTypingBDOM <- currentDOM
                        if initialDOM = afterTypingBDOM then
                              type(‘A’(element))
                              afterTypingADOM <- currentDOM
                                    if intitialDOM = afterTypingADOM then
                                          return „No code detected‟
                                    else return „Possible code detected‟
                        else return „DOM is changed after typing B‟
            else return „DOM is changed after typing arrows‟
```

`inputFile` – contains a list of URLs that need to be tested for Konami Code

`launchPage` – a method for opening an URL

`currentDOM` – Web site DOM at the time of capture

`initialDOM` – Web site DOM after the page load is complete

`findElement` – a method for finding a DOM element

`element` – an element to send the Konami Code input to

`type` – a method for sending an input to the element

`arrows` – an input sequence that consists of arrow keys

`afterTypingArrowsDOM` – Web site DOM after arrows have been typed

`afterTypingBDOM` – Web site DOM after B has been typed

`aftertypingADOM` – Web site DOM after A has been typed

The source code of our project can be downloaded from the Google Code repository[38].

[38] http://code.google.com/p/the-konami-code-project/

3.2.1. Typing Arrows

After the Web page has been loaded, the arrow sequence (↑ ↑ ↓ ↓ ← → ← →) *arrows* is sent as an input. Now, if the DOM has changed, we save the *initialDOM* and the *afterTypingArrowsDOM*. A record is written into the report CSV file stating that Konami Code could not be detected at particular URL. This is done because after typing arrows the DOM had already changed, thus we cannot detect Konami Code in such sites. We are aware that Web pages with such behavior can also contain Konami Codes, but in order to keep the algorithm simple we decided to exclude these from further research. We can solve this problem, by taking *afterTypingArrowsDOM* as the initial model and then continue typing letters B and A. However, it is not guaranteed that the DOM could keep on changing automatically.

3.2.2. Typing B

If DOM does not change after pressing the arrow keys, then letter "B" is typed. Again a check is made between the *initialDOM* and *afterTypingBDOM*. If the DOMs are not identical then a record is written into the report CSV file that Konami Code could not be detected because typing B already changed the DOM. A similar logic we applied for pressing the arrows could be applied to search for Konami Codes among Web sites which DOM is changed after a character input. Typing B separately is a particularly important step to reduce false positives, because a lot of Web sites with integrated search functionalities will start to execute AJAX to provide suggestions to the user based on the input that was received.

3.2.3. Typing A

If both typing arrows and typing B did not produce a change in the DOM, input "A" is typed. When this results in a DOM change then a possible code has been detected and the result is saved into the report along with the *initialDOM* and the *aftertypingADOM*. However when the DOM has not been changed then it can be said that from the given URL no Konami Code was found. False positives can occur for example when the DOM of a Web page is changed starting from the second character input like described in the previous paragraph in case of the first letter. An example of this is an online dictionary Web site *pons.eu*[39], which makes an *XMLHttpRequest* after a second character is typed. Another false positive reported after our experiment was a Web site loading time

[39] http://www.pons.eu/

comparison application *Which loads faster?*[40]. Here the page DOM is changed because typing letter “A” opens “About this project” iframe content. To avoid such false positives an additional process could be used to test if the page DOM changes already after typing in characters “B” and “A”.

[40] http://whichloadsfaster.com/

4. Empirical Evaluation

A case study was conducted to verify if and how Konami Codes can be automatically detected from Web pages. The experiment was aimed at addressing the following research questions.

RQ1 How many Konami Codes can be automatically detected from the list of world's top 100 000 Web sites?
RQ2 What is the precision and recall of our automated method for detecting Konami Codes?
RQ3 How many of the Web pages with Konami Codes work on different Web browsers without CBC issues?

To evaluate these research questions, we have created a Java program that uses WebDriver to open a Web page, send the Konami Code as a keyboard input and verify if the DOM of the Web page changes after the set of key inputs.

4.1. Setup

4.1.1. Program Setup

Our program has been tested using WebDriver 2.0b2[41]. It can be launched using the choice of right parameters, shown in Table 1.
It was run from the command line like this:

```
java -jar Controller.jar 0 0 100000 C: \\URL_source_file.csv
report.csv chrome M65
```

Table 1: Launch Parameters

Parameter description	Example
Deprecated parameter, currently not in use	0
Starting index	0
Last index to check	100000
URL source file	C:\\URL_source_file.csv
Name of the output file	Report_chrome.csv
Browser to use	chrome
Windows admin username	M65

41 http://seleniumhq.wordpress.com/2011/02/15/selenium-2-0b2-released/

We explain the last four parameters of Table 1 in more detail:

URL Source File

The URL source file must contain URLs in a CSV file format like represented in Table 2. The first column is the id and the second is the URL.

Table 2: The contents of a sample input file

Id,URL
`1,http://google.com`
`2,http://facebook.com`
`3,http://youtube.com`
`4,http://yahoo.com`
`5,http://live.com`
`6,http://baidu.com`
`7,http://wikipedia.org`

The first 100 000 URLs were taken from the *alexa.com's* top 1 million sites list[42] on 23.02.2011, but they did not have a protocol in front of them. We modified the list and added *http://* as a prefix to the input file because WebDriver.get() method requires an URL parameter type which must have a protocol.

Output Report

The program output is a CSV file (Table 3) and consists of six columns. First is the index column which represents the URL's popularity and second is the URL itself.

Third column contains a categorized result of the visited URL. It can have a value from six different values which range from 0 to 5. In Table 4 possible codes with their interpretation are summarized. Columns four and five contain two UNIX timestamps which are saved in the initial- and final step of the algorithm so that the detection time for each URL could be calculated. The last column is meant to hold different exceptions which are detailed in Table 9, Table 10 and Table 11. When no exception occurred, then text "null" is used.

[42] http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

Table 3: The contents of a sample output report file

```
Id,URL,Code,Time before opening the Web page, Time after typing code, Exception
0,http://google.com,0,1300909931,1300909945,"null"

1,http://facebook.com,0,1300909946,1300909957,"null"

2,http://youtube.com,1,1300909959,1300909971,"null"

3,http://yahoo.com,1,1300909973,1300909990,"null"

4,http://live.com,1,1300909993,1300910005,"null"

5,http://baidu.com,0,1300910007,1300910019,"null"

6,http://wikipedia.org,0,1300910021,1300910032,"null"
```

Table 4: Explanation of report codes

Result code	Description
0	No Code was found because after typing the code the DOM had not changed
1	Code could not be detected because after typing the arrows the DOM had changed
2	Code could not be detected because after typing “B” the DOM had changed
3	Potential code was found, because after typing “A” the DOM had changed
4	An exception occurred (The exception is added as a comment into the final column)
5	A timeout occurred and the next URL had to be taken (the `onLoad` event for the site might not have been triggered either because the URL did not resolve to a Web page or some content was not loaded in two minutes since the last modification of the report file)

Web Browser

The Web Browser parameter can be either “`firefox`”, “`iexplore`” or “`chrome`” and will launch an instance of the currently installed Web browser - please see section 2.2.4 to see the list of currently supported versions of these browsers. The program will launch and start using the provided Web browser until the end of program execution. Recently Opera released their implementation of WebDriver however our program has not been tested using Opera and the driver is not included.

Windows Admin Username

This parameter is required because WebDriver does not remove temporary files from “`C:\Users\Admin_Username\AppData\Local\Temp`” folder due to an issue in

WebDriver[43]. We experienced this behavior with all three Web browsers and at one time during testing the machine ran out of disk space. We have set the program to delete some folders from the Temp folder every time a page load timeout occurs (report entry 5). The program requires the Windows user's admin account name to get the correct folder path.

4.1.2. Environment Setup

Two test machines were used during the execution process. One was a virtual machine provided by the university (Table 5) and the other was a personal computer (Table 6). The machine in Table 5 ran the first 50 000 most popular URLs and the machine in Table 6 ran the next 50 000.

Table 5: Environment of the virtual machine

Components and software	**Parameters and version**
Processor	Intel Xeon 5110 @ 1.6 GHz (2 cores)
RAM	4 GB
Java version	1.6.0_23
Mozilla Firefox version	3.6.15
Internet Explorer version	8.0.7600
Google Chrome version	10.0.648

Table 6: Environment of the personal computer

Components and software	**Parameters and version**
Processor	Intel Core 2 Duo E7200 @ 2.53 GHz (2 cores)
RAM	4 GB
Java version	1.6.0_24
Mozilla Firefox version	3.6.16
Internet Explorer version	8.0.7600
Google Chrome version	10.0.648

4.2. Results

4.2.1. List of Sites Known to Contain the Konami Code

Three initial test runs were made with each browser on a list of 64 URLs, known to possibly contain a Konami Code, before the executing the program on 100 000 Web pages. The 64 URLs were selected from Konami Code Sites[37] based on the site's info. Although the site contains 91 unique URLs, no URLs were selected which required additional input like typing of an Enter key.

[43] http://code.google.com/p/selenium/issues/detail?id=1131

At first all 64 sites were verified manually by visiting each site's main page with three Web browsers and typing in the Konami Code.

Table 7: Program results using 64 sample Web sites from Konami Code Sites

Result code	**Firefox manual**	**Average of 3 automatic runs**	**IE manual**	**Average of 3 automatic runs**	**Chrome manual**	**Average of 3 automatic runs**
Total 0	19	16	42	37.67	19	16
Total 1	N/A	7.33	N/A	14.67	N/A	10.67
Total 2	N/A	0.67	N/A	0.33	N/A	1
Total 3	**45**	**32.67**	**22**	**9**	**45**	**34.33**
Total 4	N/A	3.33	N/A	0	N/A	0.67
Total 5	N/A	4	N/A	2.33	N/A	1.33

On manual inspection we verified that the code was present in 45 Web pages. However 19 Web sites did not reveal an Easter egg on the URL's home page. Possible reasons for this include a required login, an additional Enter key input or the Konami Code support might have been removed from the site altogether by now.

Firefox and Chrome reported both the same set of results, however in case of Internet Explorer, Konami Code was not found in 47.6% of the Web sites. Using Internet Explorer we did not find any unique URLs with Konami Code which the other two browsers did not already contain.

Then our program was run three times using the same set of 64 URLs on all three Web browsers to evaluate the automatic detection capabilities of our program. From Table 7 it can be seen that the results are fairly consistent between Mozilla Firefox and Google Chrome with correct Konami Code detection capabilities of 72.6% and 76.3% respectively while IE managed to detect the code on 40% of the time on average.

4.2.2. World's 100 000 Top Web Pages

World's top 100 000 Web pages were examined using three different Web browsers in search of Konami Codes.

Table 8 shows that 0.9% (Firefox), 1.2% (IE) and 2% (Chrome) of the Web sites checked were reported to contain a possible Konami Code (code 3 in Table 4). This result seemed likely to contain a lot of false positives and this was confirmed by manually inspecting a couple of the reported URLs.

Table 8: Program results using 100 000 world's most popular Web sites

Result code	**Mozilla Firefox**	**%**	**Internet Explorer**	**%**	**Google Chrome**	**%**
Total 0	81582	81.58	61693	61.69	66264	66.26
Total 1	13803	13.80	24690	24.69	18651	18.65
Total 2	1148	1.15	2248	2.25	3241	3.24
Total 3	**900**	**0.9**	**1241**	**1.24**	**2039**	**2.40**
Total 4	891	0.89	4438	4.44	2462	2.46
Total 5	1676	1.68	5690	5.69	7343	7.34

Figure 6 represents the total time that took to visit every URL with each Web browser. There is a noticeable difference in time it took to visit the first top 50 000 and second set of 50 000 URLs. This is probably related to the fact that different environments (Table 5 and Table 6) were used to visit the first and second set of URLs.

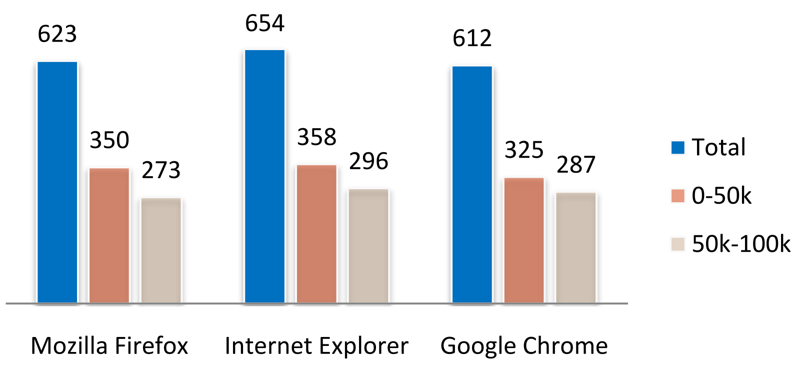


Figure 6: Hours spent for each Web browser to visit all the URLs

Tables Table 9, Table 10 and Table 11 show different exceptions (code 4 in Table 4) that were reported for each Web browser.

Table 9: Most common exceptions for Firefox representing 99.3% of the total exceptions

Exception	**Description**	**Count**
ElementNotVisibleException: Element is not currently visible and so may not be interacted with	Thrown to indicate that although an element is present on the DOM, it is not visible, and so is not able to be interacted with.	833
StaleElementReferenceException: Element not found in the cache	Indicates that a reference to an element is now "stale" --- the element no longer appears on the DOM of the page.	44
WebDriverException: java.net.SocketException: Connection reset	Thrown to indicate that there is an error in the underlying protocol such as a TCP error	8

From Table 9 it can be seen that Mozilla Firefox reported for 833 Web pages that the element that had been selected was not visible. This happened because WebDriver will not allow access to such elements, since a user cannot read text in a hidden element as well.

Table 10: Most common exceptions for IE representing 99.3% of the total exceptions

Exception	**Description**	**Count**
WebDriverException: Unable to find element with xpath == *	An element could not be located. This exception happened because the Web site content could not be loaded.	4330
ElementNotVisibleException: Element is not displayed	Thrown to indicate that although an element is present on the DOM, it is not visible, and so is not able to be interacted with.	58
StaleElementReferenceException: Element is no longer valid	Indicates that a reference to an element is now "stale" --- the element no longer appears on the DOM of the page.	21

The exceptions in the most populated class with 4330 occurrences reported for Internet Explorer were caused by the Web page URL not being resolved to a Web page (Table 10). Mozilla Firefox reported such pages with code 0 and Google Chrome with code 5 because of the different ways these Web browsers display the information in case a Web page could not be opened.

Table 11: Most common exceptions for Chrome representing 98.4% of the total exceptions

Exception	**Description**	**Count**
ElementNotVisibleException: Element was not visible	Thrown to indicate that although an element is present on the DOM, it is not visible, and so is not able to be interacted with.	1648
NoSuchElementException: Was not on a page	Caused by a known bug in ChromeDriver[44]	693
StaleElementReferenceException: Element is obsolete	Indicates that a reference to an element is now "stale" --- the element no longer appears on the DOM of the page.	81

Google Chrome reported 1648 exceptions regarding elements that matched the XPath expression, but were not visible and therefore could not be selected (Table 11). For 693 URLs a *NoSuchElementException* was given and on later inspection we found that this was caused by a known bug in the current ChromeDriver[44].

[44] http://code.google.com/p/selenium/issues/detail?id=427

4.2.3. Modified Algorithm for Web Sites with Code 3

In order to get more accurate results and reduce the number of false positives, we decided to apply an additional technique to the sites reported to contain an Easter egg behind a Konami Code.

We modified Algorithm 1 by adding a fixed time to wait after the page had been loaded and before starting to type the arrows. To select the proper time limit we conducted an experiment on the 64 URLs that we also used in paragraph 4.2.1 and observed URLs which DOM changed automatically after the page had been loaded. We chose the time of the URL which took the longest to automatically change and multiplied it by two to avoid pages which change after even a longer wait period. The longest change was 12.5 seconds. After the page had been loaded and the initial DOM had been saved, the program was made to wait for 25 seconds. After the waiting period was over, the DOM was saved again for a second time and then both DOMs were compared. If the DOMs were equal then the algorithm resumed its usual work, otherwise the page was reported using a new code - 6.

Table 12: Program results on Web sites reported to use the Konami Code in Table 8

Result code	**Firefox**	**%**	**IE**	**%**	**Chrome**	**%**
Total 0	127	14.11	148	11.93	129	6.33
Total 1	10	1.11	18	1.45	3	0.15
Total 2	12	1.33	11	0.89	0	0
Total 3	**41**	**4.55**	**15**	**1.21**	**25**	**1.22**
Total 4	1	0.11	15	1.21	2	0.01
Total 5	9	1	40	3.22	167	8.19
Total 6	700	77.77	994	80.1	1713	84.02

The program, now with the forced wait time, was run with URLs reported in Table 8 as code 3 results, as the list of input URLs. Results in Table 12 indicate that pages with dynamic content (code 6), were the main reason why so many false positives had occurred 77.8% (Mozilla Firefox), 80% (Internet Explorer) and 84% (Google Chrome) of the total Web sites checked for each Web browser. Then we manually observed the new code 3 results for each browser to see how many of the reported Web sites actually contained the Konami Code.

Table 13: Figures after manual inspection of the reported Konami Codes in Table 12

Status	**Mozilla Firefox**	**Internet Explorer**	**Google Chrome**
No code	9	5	2
Code present	**32**	**10**	**23**
Accuracy	78%	67%	92%
Codes total	41	15	25

After manually inspecting the sites with potential Konami Codes, we were able to conclude that our approach was able to tell if a site contained the Konami Code 78% (Mozilla Firefox), 66.7% (Internet Explorer) and 92% (Google Chrome) of the time.

4.2.4. Manual Inspection of Web Sites

To evaluate the approximate size of false negatives that might have been incorrectly discarded using the automatic approach with a forced waiting time, we manually examined URLs that were initially found as potential codes in Table 8. Summary of manual inspection can be seen in Table 14 which shows that 60 pages with Mozilla Firefox, 18 with Internet Explorer and 35 with Google Chrome were found. This means that the method where the program waited for 25 seconds discarded 28 (46.7%) Konami Codes when using Mozilla Firefox, 8 (44.4%) codes when using Internet Explorer and 12 (34.3%) codes when using Google Chrome.

Table 14: Konami Codes found using manual inspection on Web sites reported in Table 8

Web browser	**Mozilla Firefox**	**Internet Explorer**	**Google Chrome**
Initially reported	900	1241	2039
No code	841	1226	2008
Code present	**60**	**18**	**35**

4.2.5. Cross-Browser Compatibility

To evaluate the CBC of Konami Code from the list of newly detected Web sites, we decided to manually examine a union set of unique URLs taken from reported sites in Table 12 and Table 13.

Table 15: CBC of the new Web sites with Konami Code

Status	**Mozilla Firefox**	**Internet Explorer**	**Google Chrome**
No code	1	23	0
Code present	**59**	**37**	**60**
Total	60	60	60

From Table 15 it can be seen that Mozilla Firefox and Google Chrome reported exactly the same set of results besides one URL which did not react to the Konami Code input in Firefox - http://konigi.com. Also two sites did not open with IE and Chrome with *http://* prefix and *http://www.* had to be used to test these pages for Konami Code presence. This is because Firefox uses a feature called *Domain Name Guessing* and automatically adds www to the URL. As could be already predicted by the initial test results in Table 7, pages with IE reported significantly less codes (38.3%) than the two other Web browsers.

4.2.6. Konami Code Intersection Between Different Browsers

We also decided to examine the set of URLs reported as possible Konami Codes in Table 8 by comparing every browser combination using intersection between the sets. From Figure 7 it can be seen that the intersection of all three Web browsers contained the code 15 times out of 16 (93.6%). This result shows that by exploiting the CBC issues it is possible to find Konami Codes very accurately using this method. The results with intersections between two browsers contain more Konami Codes, but are not that accurate. The intersection Chrome ∩ Firefox contains more (39.6%) Konami Codes than the intersections IE ∩ Firefox (31.7%) and IE ∩ Chrome (19.3%). This method, which relies on the CBC of the Web site, was also able to find Konami Codes, but is less effective when compared to the results of our method with forced wait time (Table 13).

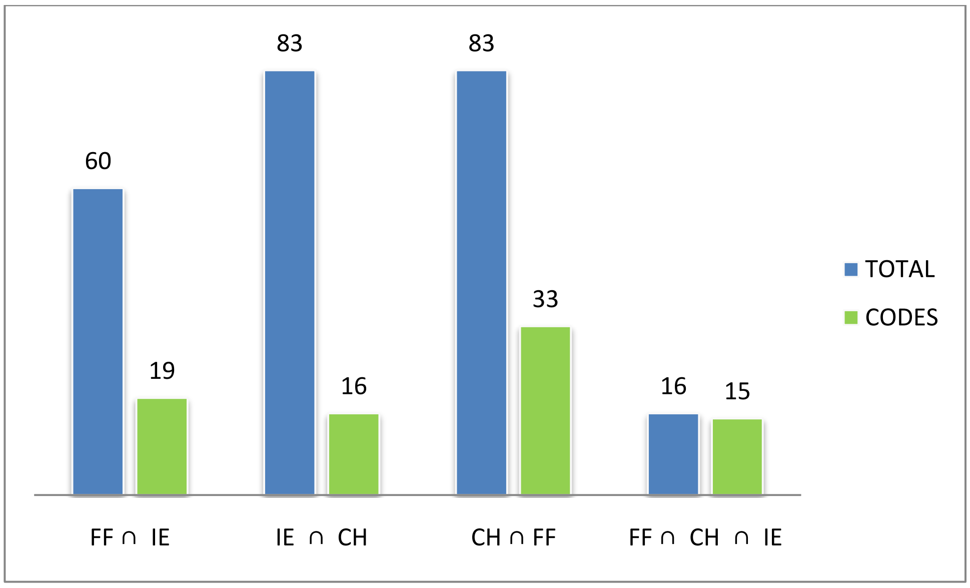


Figure 7: Table 8 code 3 result intersection sets for every browser combination

4.2.7. List of New Web Sites Detected

We took the list of URLs from Konami Code Sites[37] as a reference of sites known to contain Konami Code and compared it to our findings. The sites that contain Konami Code appear to fall into different categories with Web design, video game and programming related Web pages standing out the most. We plan to send our complementary list to the owners of the site.

Table 16: New Web sites detected automatically when waiting 25 seconds after page load

Web sites detected automatically		
http://github.com	http://bannersnack.com	http://archiduchesse.com
http://amobil.no	http://itler.net	http://seilmagasinet.no
http://dlink.de	http://the-big-bang-theory.com	http://akam.no
http://godsgirls.com	http://snacktools.com	http://amirite.net
http://nsmb.com	http://gamestar.de	http://pixel2life.com
http://duelinganalogs.com	http://txstate.edu	http://dslvalley.com
http://bttradespace.com	http://bordom.net	http://episerver.com
http://iconarchive.com	http://smbc-comics.com	http://sparkfun.com
http://paulirish.com	http://teknofil.no	http://thedoghousediaries.com
http://instantshift.com	http://wearehunted.com	http://konigi.com
http://iapps.im	http://n-styles.com	http://mister-auto.com

Table 16 contains 33 automatically detected new Web sites with Konami code. 27 new Web sites found using manual inspection are represented in Table 17.

Table 17: New Web sites detected by manually examining URLs with code 4 in Table 8

Web sites detected using manual inspection (does not contain results already in Table 16)		
http://mozilla.org	http://add.io	http://funnyordie.com
http://glassdoor.com	http://mochimedia.com	http://soundclick.com
http://earticlesonline.com	http://tupalo.com	http://oakley.com
http://diskusjon.no	http://mochiads.com	http://gonintendo.com
http://jonraasch.com	http://evo.com	http://nvidia.it
http://voddler.com	http://absoluteradio.co.uk	http://purepwnage.com
http://splitbrain.org	http://nvidia.fr	http://bigspaceship.com
http://ideaonline.co.id	http://mpsaz.org	http://rifftrax.com
http://comviq.se	http://contagiousmagazine.com	http://texastribune.org

4.2.8. Evaluation of the Research Questions

Based on the results of our experiment we can answer the **RQ1** and say that 38 (a union set of unique URLs from the reports of all Web browsers) Web sites with Konami Code were found automatically from the 100 000 world's most popular Web sites. Additionally 27 Web sites were found semi-automatically after the manual inspection of code 3 results in Table 8.

To answer **RQ2** and calculate the precision and recall of our experiment we compared URLs from Table 13 and Table 14. The precision and recall formulas are given as follows:

Precision = $\frac{tp}{tp+fp}$ Recall = $\frac{tp}{tp+fn}$ F-measure = 2*$\frac{precision*recall}{precision+recall}$

We also decided to calculate the F-measure as both precision and recall measures can have their weak points. The tp in formulas stands for count of true positives, fp for false positives and fn for false negatives.The precision and recall is given for each browser separately in Table 18. When comparing the precision results to the recall values then it can be seen that our method is aimed to be more precise and reduce the number of false positives. High precision is achieved at the expense of recall, which is lower than the precision because of the false negatives that were discarded using the 25 second wait period. We would not have much use for high recall when the precision would be low since finding Konami Codes from a set that contains a lot of false positives is a labor intensive task.

Table 18: Precision, recall and F-measure results

Measure	Mozilla Firefox	Internet Explorer	Google Chrome
Precision	0.78	0.67	0.92
Recall	0.53	0.56	0.66
F-measure	0.63	0.61	0.77

RQ3 can be answered based on the results of Table 15. In Google Chrome the Konami Code was found in all 60 new Web pages with Konami Code. In Mozilla Firefox the code was found in 59 (98.3%) URLs. Internet Explorer had the lowest CBC with 37 Web sites (61.7%). The intersection of the three Web browsers is 37 URLs. This means that 61.7% of the Konami Codes worked properly on all three Web browsers.

4.3. Threats to Validity

Our implementation has a several limitations, some of them have been fixed already, but were present in the version that was used when conducting the experiment.

- Web pages which DOM level content is modified without any user interaction are currently ignored by our approach. For example:
  - Web pages with splash screens that automatically switch content after having displayed an intro animation clip like this page[45].
  - Pages which execute JavaScript code when the `onLoad` event is triggered
  - Pages with dynamically changing advertisements
  - Pages with social networking site toolbars that are loaded after the browser's `onLoad` event
- Due to a WebDriver limitation[46] we are unable to detect the Konami Code in Web pages which will display a JavaScript alert window after the code input – an example Web site with such behavior is *Absolute Bica*[45].
- Web pages which do not load completely (the body `onLoad` event is not triggered) in two minutes are not checked and the next candidate URL is launched to avoid pages which load endlessly and to keep the program working. Two minutes was chosen based on the results of the initial test experiment of 64 URLs that is described in paragraph 4.2.1.
- In several Web pages the element that is found to send the Konami Code input to is not found by using our XPath expression.

  ```
  WebElement element = driver.findElement(By.xpath("*"))
  ```

  This happens when the element found by this XPath expression is not visible. This problem could be avoided by first checking if the candidate element is visible and if it is not, then another element should be selected.
- We do not know if *http://www.* would have been a better choice with regard to the percentage of Web pages loaded compared to the *http://* prefix that we used. The program code could be improved so that if a Web page is not found, the program would try to open the URL using the *www* prefix as well. Currently the number of pages not loaded due to this limitation could be figured out by performing a static analysis on the DOMs to find out the URLs that did not resolve to any Web page.

[45] http://www.absolutebica.com/

This retrieved list could then be run again, but now with *http://www.* prefix instead of running with *http://*.

- Some Web sites might be programmed to display Konami Code only on a specific browser intentionally, but in our work we consider a code that works on one Web browser and not on another to be a CBC issue.

5. Conclusions and Future Work

In this thesis we have shown that Konami Code is a perfect example of modern Web content because the user would never find an Easter egg by simply following the Web sites classic URL navigation model. Therefore it would also be difficult to detect unexpected behavior of modern Web pages when testing with insufficient methods and outdated tools. Users expect identical browsing experience on all modern Web browsers and the functional CBC issues make the testing of RIA even more daunting. We proposed an automatic method and were able to discover 60 new Web sites with Konami Code from the world's top 100 000 Web pages. 33 of them were found automatically and 27 semi-automatically by manually examining a set of initially reported URLs with each Web browser. Our method achieved a high precision and a medium recall rate. We also tried to detect Konami Codes using a CBC approach, which returned correct results with high accuracy (96.3%). To evaluate the CBC of Web sites with Konami Code we conducted a manual inspection and found that the code does not work on one Web page with Mozilla Firefox and several Web pages with Internet Explorer. This may indicate possible similar functional CBC issues with other modern Web sites.

For future work we would like to solve some of our program's limitations. Our current approach is only capable of searching for the Konami Code from a predefined URL, which in our case was the site's home page. However some Web sites may contain the Konami Code in different subdomains and different states of the Web site as well. For example in case of *speccedforawesome.com* the Konami code is in the forum section which can be accessed from the site's home page. To test for Konami Code in many different states of a given starting URL, a Web crawler could be used together with our program's plugin-like implementation for Crawljax.

The list of *alexa.com*'s top 1 million Web sites contains an additional 900 000 URLs which we have not checked so far. It is very likely that more Web sites with Konami Code could be detected given that we discovered 60 new URLs from a list of 100 000, compared to the 91 Web sites currently known to contain the Konami Code on Konami Code Sites[37].

6. Konami koodiga veebilehtede automaatiseeritud leidmine vaadeldes nende brauseriteüleseid erinevusi

Bakalaureusetöö (6 EAP)

Risko Ruus

Resümee

Käesolev bakalaureusetöö sisaldab endas automatiseeritud lahendust veebilehtedest Konami koodi leidmiseks. Lisaks vaadeldakse tuvastatud lehekülgede ühilduvust erinevate veebilehitsejatega.

Konami kood on sisendkombinatsioon (↑ ↑ ↓ ↓ ← → ← → B A), mis pärineb jaapani mängutootja Konami 1986 aasta videomängust *Gradius*. Tänapäeval on internetis palju veebilehti, mis sisaldavad endas samuti Konami koodi. Lehekülje külastajal on võimalik sisestada kombinatsioon oma arvuti klaviatuurilt, mille tulemusena kuvatakse arvutiekraanile lehe varjatud sisu. Tihtilugu on sellised veebilehed arendatud erinevate tehnoloogiate kogumit AJAX kasutades, mis võimaldab muuta lehekülje dokumendi-mudelit (DOM) ilma lehte uuesti laadimata. Veebilehtedel esineb aga tihti veebibrauseriteüleseid ühilduvusprobleeme ning käesolevas töös uurimegi kui palju automaatselt tuvastatud Konami koodiga veebilehtedest töötab nii *Mozilla Firefox, Internet Explorer* kui ka *Google Chrome* brauseritel.

Töös viiakse läbi eksperiment *alexa.com* portaali andmetel põhineva 100 000 maailma populaarseima veebilehe uurimiseks. Selleks oleme ehitanud rakenduse kasutades *Java* programmeerimiskeelt ning veebilehtede testimiseks loodud raamistikku *WebDriver*. Oleme seadnud üheks eesmärgiks leida nende 100 000 veebisaidi seast võimalikult palju uusi Konami koodiga lehekülgi

Eksperimendi tulemusena leidsime automaatselt 60 uut Konami koodiga lehekülge. Nendest 33 leidsime automaatselt ning 27 pool-automaatselt eksperimendi vahetulemuste käsitsi läbivaatamise käigus. Antud tulemuste uurimise järel selgus, et veebilehitsejates *Mozilla Firefox* ning *Google Chrome* töötavad Konami koodiga veebilehed võrdväärselt hästi, kuid *Internet Explorer* ei suuda kuvada Konami koodiga varjatud saladust peaaegu pooltelt tuvastatud veebilehtedelt.

Meie töö väärtuslikuks avastuseks võib lugeda automatiseeritud lahenduse loomist Konami Koodide leidmiseks ning 60 uue Konami koodi sisaldava veebilehe leidmist. Varem teadaolevad 91 URLi olid seni avastatud käsitsi ning üldsusele teada vaid 91. Lisaks näeme neid tuvastatud veebilehti võimaliku materjalina brauseriteüleste funktsionaalsete erinevuste põhjalikumaks uurimiseks.

Autor avaldab tänu oma juhendajale, kelle ideed ning soovitused on olnud tõeliseks abiks käesoleva töö kirjutamisel. Samuti soovib autor tänada Oskar Grossi, kes pühendas oma aega ja energiat töö läbivaatamisele ning parandusettepanekute tegemisele. Töö valmimisse on panustanud hilisõhtuseid tunde ka Virge Terav, kes aitas koos autoriga käsitsi kontrollida üle 4000 veebilehe, mille eest autor on talle südamest tänulik.

7. References

Allaire J Macromedia Flash MX—A next-generation rich client [Report]. - 2002.

Andrews Anneliese A., Offutt Jeff and Alexander Roger T. Testing web applications by modeling with fsms [Journal] // Software and Systems Modeling. - 2005. - Vol. 4. - pp. 326-345.

Artzi Shay [et al.] Finding bugs in dynamic web applications [Conference]. - [s.l.] : ACM, 2008. - pp. 261-272.

Benedikt Michael, Freire Juliana and Godefroid Patrice VeriWeb: Automatically Testing Dynamic Web Sites [Conference]. - 2002.

Garrett Jesse James Ajax: A new approach to web applications. Adaptive Path // Ajax: A new approach to web applications. Adaptive Path. - 2005.

Marchetto Alessandro, Ricca Filippo and Tonella Paolo A case study-based comparison of web testing techniques applied to AJAX web applications [Journal] // Int. J. Softw. Tools Technol. Transf.. - [s.l.] : Springer-Verlag, 2008. - Vol. 10. - pp. 477-492.

Marchetto Alessandro, Ricca Filippo and Tonella Paolo An Empirical Validation of a Web Fault Taxonomy and its Usage for Web Testing [Journal] // J. Web Eng.. - 2009. - 4 : Vol. 8. - pp. 316-345.

Marchetto Alessandro, Tonella Paolo and Ricca Filippo State-Based Testing of Ajax Web Applications [Conference]. - [s.l.] : IEEE Computer Society, 2008. - pp. 121-130.

Mesbah Ali and Deursen Arie van A component- and push-based architectural style for ajax applications [Journal] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2008. - Vol. 81. - pp. 2194-2209.

Mesbah Ali and Prasad Mukul Automated Cross-Browser Compatibility Testing [Conference]. - [s.l.] : ACM, 2011.

Mesbah Ali, Bozdag Engin and Deursen Arie van Crawling AJAX by Inferring User Interface State Changes [Conference]. - [s.l.] : IEEE Computer Society, 2008. - pp. 122-134.

O'Reilly Tim O'Reilly -- What Is Web 2.0 [Journal]. - 2005. - 31 August 2008.

Ricca Filippo and Tonella Paolo Analysis and testing of Web applications [Conference]. - [s.l.] : IEEE Computer Society, 2001. - pp. 25-34.

Ricca Filippo and Tonella Paolo Web Testing: a Roadmap for the Empirical Research [Conference]. - [s.l.] : IEEE Computer Society, 2005. - pp. 63-70.

Rode Jochen, Rosson Mary Beth and Pérez-Quiñones Manuel A. The Challenges of Web Engineering and Requirements for Better Tool Support [Report] / Virginia Polytechnic Institute and State University Center for Human-Computer Interaction. - 2002.

Torchiano Marco, Ricca Filippo and Marchetto Alessandro Defect location in traditional vs. Web applications - an empirical investigation [Conference]. - 2009. - pp. 121-129.

Wang Minghui [et al.] A Static Analysis Approach for Automatic Generating Test Cases for Web Applications [Conference]. - [s.l.] : IEEE Computer Society, 2008. - pp. 751-754.