THE UNIVERSITY of York

Submitted in part fulfilment for the degree of MSc in Cyber Security.
Department of Computer Science at the University of York.

Using Code Coverage to optimise Web Browser Fuzzing

James Fell

30th August 2017

Supervisor: Professor Howard Chivers

Word Count: 19882 as counted by the *Libre Office 5.2* word count.

This includes the body of the report excluding the references in Appendix 1 and source code in Appendix 2.

Abstract

Identifying vulnerabilities in software products has been an important area of security research for many years. One of the most successful approaches to this problem so far has been that of fuzz testing or fuzzing. This process involves repeatedly passing invalid inputs into an application at runtime and monitoring it with a debugger or other instrumentation for evidence of potential vulnerabilities being triggered, such as observing a segmentation fault or memory corruption. One of the main ways of measuring how thoroughly the fuzzing process has tested a given piece of software is by measuring code coverage throughout the process. If part of the target program has not been executed during fuzzing then any vulnerabilities in that part obviously cannot be detected. This project investigates the development of a suite of software tools that enable the fuzzing of a web browser such that the maximum code coverage is obtained during the testing. The prototype system is evaluated through a number of experiments and is finally used to fuzz a selection of web browsers. Multiple previously unknown bugs are detected in several different web browsers including Edge, Internet Explorer and Firefox.

Acknowledgements

I would like to thank my supervisor, Professor Howard Chivers, for his valuable guidance and feedback during the completion of this project. I would also like to thank Berend-Jan Wever (aka Skylined) for engaging in long technical discussions with me about many aspects of fuzzing web browsers, as well as for sharing his BugId crash analysis tool with me. Similarly, conversations with Mateusz Jurczyk (aka j00ru) of Google, Jeremy Brown of NVIDIA, Atte Kettunen of F-Secure and Rosario Valotta all provided me with useful advice and helped to accelerate my progress.

An extended version of the literature review from this report was published in PenTest Magazine on 10th March 2017. In addition, the precis paper that accompanies this report was submitted to Elsevier Computers & Security on 20th August 2017. At the time of writing, it is undergoing peer review.

Ethics Statement

Potential ethical issues were considered before and during carrying out the work described in this report. The project did not involve experiments that would affect any third party networks, computer systems, data or people. No privacy or data protection concerns existed. The main ethical concern was the potential for misuse of any resulting tool or technique that aids in the discovery of software vulnerabilities. However, fuzzing is known to already be an important part of the ongoing development process for the Google Chrome, Microsoft Edge and Mozilla Firefox teams. Any contribution to this field is therefore likely to improve security for everyone rather than harm it. It was also agreed that any vulnerabilities discovered during the experiments would be reported to the appropriate software vendors for fixing prior to eventually being made public.

Table of Contents

Table of Contents

Chapter 1: Introduction

Identifying vulnerabilities in software has long been an important research problem in the field of information security. Over the last decade, improvements have been made to programming languages, compilers and software engineering methods aimed at reducing the number of vulnerabilities in software [26]. In addition, exploit mitigation features such as Data Execution Prevention (DEP) [65] and Address Space Layout Randomisation (ASLR) [66] have been added to operating systems aimed at making it more difficult to exploit the vulnerabilities that do exist. Nevertheless, it is fair to say that all software applications of any significant size and complexity are still likely to contain undetected vulnerabilities and it is also frequently possible for skilled attackers to bypass any defences that are implemented at the operating system level [6, 7, 12, 66].

There are many classes of vulnerabilities that occur in software [64]. Ultimately they are all caused by mistakes that are made by programmers. Some of the most common vulnerabilities in binaries are stack based and heap based buffer overflows, integer overflows, format string bugs, off-by-one vulnerabilities, double free and use-after-free bugs [5]. These can all lead to an attacker hijacking the path of execution and causing his own arbitrary code to be executed by the victim.

Techniques for identifying software vulnerabilities can be divided firstly into two different approaches; static analysis and dynamic analysis. Static analysis of software involves ways of examining the source code or compiled binary without executing it. Dynamic analysis involves examining the software at runtime, typically after attaching some kind of debugger. Both of these approaches have their relative advantages and disadvantages [69].

Static analysis, although useful, often produces many false positives that cannot be exploited in practice and requires a lot of manual verification work to identify which issues are genuine vulnerabilities [69]. It does, however, allow for complete code coverage with the entire application being inspected.

For dynamic analysis the most common automated technique for finding vulnerabilities is the process of fuzz testing or fuzzing. In essence this consists of repeatedly giving an application invalid input and monitoring for any sign of this triggering a bug, such as the application crashing or hanging [67]. There are many advantages to this approach such as the ease of automation and the ability to test even very large applications where code review would be too time consuming. It is also the case that each bug discovered by fuzzing automatically comes with its own 'proof of concept' test case proving that the bug can definitely be triggered by a user. In practice many serious and high profile memory corruption vulnerabilities today are discovered through fuzzing.

In order to fully test an application using fuzzing, it is important to somehow ensure that the combination of test cases being used execute as large a proportion of the application as possible. That is to say, it is necessary to maximise code coverage [71]. If large sections of code are not executed during the fuzzing process, it is certain that any vulnerabilities that do exist in those areas will not be found. It is this problem that we are concerned with in this report.

It is possible to fuzz any application that accepts input from the user, whether this is in the form of opening a file, reading from a network socket, reading environment variables or any other form of input that can be modified. However, the system developed in this project focuses exclusively on fuzzing web browsers. The web browser is an ubiquitous piece of software used by everyone who has a computer. It is also routinely used to open untrusted input every time it is used to search for information and visit new web sites. Plenty of examples from recent years can be found of web browser vulnerabilities being used for everything from

stealing banking information to state sponsored espionage campaigns [46].

This report describes the development of a new suite of browser fuzzing tools that can be used to perform the following tasks:

- HTMLscan - Analyse an existing corpus of web pages and advise on how it could be improved to get better code coverage.
- HTMLscrape - Create a new corpus of web pages by random web crawling.
- HTMLgen - Create a new corpus of web pages by generating it from a custom grammar that we construct as part of this project. The resulting corpus is ideal for use in browser fuzzing sessions due to the superior code coverage that it provides.
- HTMLharness – Test harness to read a corpus of web pages from disk and sequentially deliver each test case to a web browser in order to fuzz it.
- HTMLfuzz – Continuously generate test cases using the same grammar as HTMLgen, deliver them to a target web browser and then apply a collection of fuzzing strategies to the DOM tree using JavaScript in order to trigger memory corruption bugs.

The remainder of this report is structured as follows.

Chapter 2 presents the current state of the field of fuzzing. This is examined in detail and previous work by the security research community is evaluated, including that which focused on attacking web browsers. Gaps in the literature are identified.

Chapter 3 describes the aims and objectives of the project and what was set out to be achieved.

Chapter 4 documents the design of the proposed system and compares it to the alternatives that were considered. A suitable set up for conducting experiments is also described. All design decisions are justified.

Chapter 5 discusses the implementation of the suite of software tools and the challenges that were overcome.

Chapter 6 describes experiments that were done to evaluate the developed system. The results are presented, analysed and discussed.

Chapter 7 covers the conclusions that can be drawn from the research that was done and also the future work that could be carried out.

Chapter 2: Background

In this chapter the fuzzing method of detecting vulnerabilities is introduced. The current state of fuzzing technology is examined and the strengths and weaknesses of different approaches are compared. Finally, previous work on the fuzzing of web browsers is reviewed and opportunities for original research are identified.

2.1 Fuzzing

The term fuzzing was first used in 1988 by Professor Barton Miller at the University of Wisconsin. Whilst connected to a University terminal via a dial-up connection during a storm, Miller noticed that line noise was causing extra, random characters to be added to the commands that he was sending and this often caused the various UNIX utilities that he was using to crash [9].

This experience motivated Miller and other researchers at the University to create a tool for generating random strings and then pipe its output into the input of a range of UNIX utilities. The group tested almost ninety different UNIX utilities this way on seven different UNIX distributions and found that they were able to crash over 24% of them. These experiments were described in detail in [9] and gave birth to a relatively simple way to supplement the formal verification and software testing techniques that already existed.

Since Miller's early work, fuzzing has become steadily both more popular and more sophisticated. One important step forwards for the popularity of fuzzing was the 2001 public release of the SPIKE fuzzing tool by Dave Aitel followed by his presentation of it at the Black Hat USA conference in 2002 [10]. The SPIKE framework was an API and set of tools for network protocol fuzzing. It allowed the user to create a model of any network protocol and then use this model to create and send traffic that almost complied with the specification, but not quite.

A successful fuzzing framework consists of more than just a way to repeatedly create invalid inputs and pass them to the target application. Some way of instrumenting the target is required in order to monitor what is happening internally as each test case is processed. Crashes and other misbehaviour must be detected and also logged and reported to the user along with as much data as would be necessary for further manual investigation. There must also be a test harness that can automatically restart the application as necessary so that the fuzzing campaign can run unsupervised. Better frameworks will also handle issues such as crash deduplication [97], crash triage based on suspected exploitability [39], and test case minimisation [38].

In [40] and [102] we see two different studies in which a number of popular free fuzzing tools are evaluated and compared to each other. This work illustrates well that there are many different kinds of fuzzers and they are suitable for different situations. There is no single tool that provides the best solution in every situation.

2.2 Instrumentation

Fuzzing cannot be successful without some form of instrumentation being applied to the target application. At the very least, there must be some way of detecting when the program crashes and logging which test case caused the crash. However, far more than this can be achieved using more advanced instrumentation. It is possible to detect subtle forms of memory corruption and other errors even when they do not cause the program to crash outright [68]. This allows the detection of vulnerabilities that would otherwise be missed. It is also possible to measure code coverage and therefore know which parts of the target application have been tested so far and which have not [71]. This can be simply for informational purposes for the user or can actually be acted upon by the fuzzer itself when creating new test cases.

Instrumentation can be provided directly by the fuzzing framework or by using a separate third party tool. When no source code is available it is common to use dynamic analysis tools such as Valgrind [36], DynamoRIO [75] or PaiMei [37]. It would also be possible to simply attach a debugger such as IDA Pro [79] to detect when the target crashes.

Valgrind runs on Linux and contains its own memory error detector and two thread error detectors [76]. This allows it to be used during fuzzing to detect memory corruption even when it does not cause the program to crash. DynamoRIO can run on Windows as well as Linux and can be used to extract code coverage information among many other things [75]. PaiMei is a reverse engineering framework that can also be used for monitoring code coverage of Windows applications through its Pstalker module [37] and Flayer is a tool built on top of Valgrind which allows taint analysis to be performed at runtime [77]. All of these tools can be made use of for instrumentation needs when fuzzing a target for which no source code is available.

If source code is available then instrumentation can be added at compile time. For example, on Linux systems AddressSanitizer allows instrumentation for detecting memory errors such as buffer overflows and use-after-free to be added as the application is compiled [68].

Some fuzzers, notably AFL (American Fuzzy Lop), add their own custom instrumentation at compile time [41]. This allows the fuzzer to get precise feedback from the target application during the execution of each test case. The creation of future test cases can then be based on this feedback. Such evolutionary based fuzzing is explained in more depth in section 2.3.

It should be noted that all forms of instrumentation slow the target program down, often significantly [78]. It is therefore important to consider what is needed for each specific fuzzing scenario as there is a trade-off between more speed and better instrumentation.

2.2.1 Code Coverage

As mentioned, code coverage refers to tracking which parts of a program have been executed. This makes it possible to see how much of the target has actually been tested. Vulnerabilities cannot be found in a section of a program if that section does not get executed by any of the test cases.

In [94] research was carried out showing that measuring the code coverage of an initial corpus and adding new test cases to it until no further coverage can be gained is highly beneficial and increases the number of vulnerabilities discovered. This was confirmed in [98]. Further, specific to browser fuzzing it was documented in both [61] and [100] that code coverage for an initial web browser test corpus can be increased by including existing conformance test suites from the browser vendors as these are designed to test as many parts of the browser as possible.

There are three common approaches to measuring code coverage; line, branch and path coverage [71].

Line coverage keeps track of which individual lines of source code have been executed. This is the most basic kind of coverage information that can be collected. The shortcoming is that for example a conditional statement can be marked as executed as soon as the condition is tested, whether or not it evaluated to true, if it is all on one line.

Branch coverage deals with this by keeping track of which branches have been taken for each conditional jump in the program.

Path coverage is even more detailed and keeps track of which different paths have been taken through the program, meaning which sequences of lines and branches have been executed and in which order.

It obviously takes more test cases to achieve full path coverage than full branch coverage, and more test

cases to achieve full branch coverage than full line coverage. “In general, a program with n reachable branches will require 2n test cases for branch coverage and 2^n test cases for path coverage” [71].

The gcov tool on Linux is a popular tool for adding code coverage instrumentation when the target's source code is available. It is now built in to the gcc compiler and can be activated by using the “-fprofile-arcs -ftest-coverage” option when compiling [107]. Using gcov provides line, function and branch coverage data but not path coverage.

As mentioned in Section 2.2 various tools such as Valgrind can be used to get coverage data when the source code is not available. In that case we would be reasoning about lines of assembly language from a disassembled binary rather than lines of source code.

We will return to the subject of code coverage when we examine evolutionary fuzzing in section 2.3 but it is clear that we can only find vulnerabilities in the segments of code that we actually execute during our fuzzing and this depends entirely on the test cases that we make use of.

2.3 Synthesis of Test Cases

Although issues such as instrumentation and efficiency are important, probably the most crucial aspect of a successful fuzzing campaign is creating good test cases that will interact with the target in a way likely to expose any defects. There are three main approaches to creating the test cases or inputs for use in a fuzzing campaign; mutation, generation and evolution [71, 99].

Mutation fuzzing works by starting with a corpus of one or more valid input samples. For example, when fuzzing an image viewer one might start with a number of valid JPEG files. These are then repeatedly modified or corrupted to produce new test cases that are each given as input to the software being tested. This is sometimes referred to as 'dumb fuzzing' as the fuzzer has no specific knowledge of the program being fuzzed or the correct format or syntax to use for inputs.

There are many ways in which the test cases can be mutated. Walking bit flipping and byte flipping involves inverting sequences of bits at different places in the test case [81]. Deleting segments of the test case and also splicing in sections from other test cases are common mutation approaches [80]. Another approach is to select sections of the test case and repeat them by inserting copies at random positions [100]. A recent example of a mutation fuzzer is Zulu from NCC Group [81].

Generation fuzzing works by creating test cases based only on some kind of model that describes a valid input, such as a grammar of a programming language. Continuing with the image viewer example, with a specification for the PNG image format it would be possible to repeatedly generate slightly invalid PNG files to use as test cases without needing to use any samples of existing files. This is sometimes referred to as 'smart fuzzing' as the fuzzer is applying detailed knowledge of the format or protocol being fuzzed. This generally results in better code coverage and deeper testing of the target program [2]. An example of a generation fuzzing framework is Sulley [84].

Evolutionary fuzzing creates new test cases based on the response of the target program to previous test cases [103, 105]. This can be an extension of either mutation or generation fuzzing, as either technique can be guided to create the new test cases. In either case, the difference with evolutionary fuzzing is the way in which test cases are created based on feedback from the instrumentation so that they evolve towards a specific goal [99]. Based on that goal and also the precise way in which it is achieved, evolutionary fuzzing can itself be subdivided into three common approaches; coverage-based , taint-based and symbolic-assisted fuzzing.

Coverage-based evolutionary fuzzing attempts to evolve test cases that result in new paths being executed in

the target. Test cases that exercise new code are favoured over those that do not. An example of a popular coverage-based evolutionary fuzzer is AFL (American Fuzzy Lop) [43]. AFL carries out file format mutation fuzzing and so is initially seeded with examples of valid input files. It then measures code coverage for each test case and uses Genetic Algorithms [108] to evolve a population of test cases that cover the maximum amount of code. This approach has proved extremely successful in real world fuzzing campaigns with AFL discovering many serious vulnerabilities in many popular applications [41]. In addition, an experiment was carried out that involved fuzzing the djpeg utility on Linux with AFL using only a simple text file containing the word “hello” as its initial corpus. This text file was seen to evolve into a valid JPEG file over a couple of days, simply by AFL observing how different mutations to the original test case resulted in different code paths being executed in the target [42]. In effect, the fuzzer was able to learn the JPEG file format from scratch.

Taint-based evolutionary fuzzing attempts to evolve test cases that exercise specific target code paths. Instead of simply aiming for maximum code coverage, the aim is to evolve test cases that will propagate user input into specific target functions within the program [96]. The taint analysis can either be performed beforehand as a static technique or can be performed dynamically during the fuzzing.

Lanzi et al created a prototype system that uses prior static analysis to identify specific paths of execution to target and then guides the fuzzing with this [3]. Sofia Bekrar et al worked on a system that uses dynamic taint analysis during fuzzing [4, 85]. The idea behind both is to allow a mutation fuzzer to focus its mutations on the areas of the test cases that are most likely to expose a vulnerability. These areas can be identified by tainting all user input and then tracing which inputs reach target sinks. The work carried out in [3] and [4] is therefore quite similar in concept, the main difference being that the former applied the taint analysis to the binary without executing it whilst the latter carried out the analysis on the binary at runtime.

Symbolic-assisted evolutionary fuzzing combines symbolic execution [82] with fuzzing in an attempt to enjoy the strengths of both. Test cases are modified using a dynamic symbolic execution engine to make changes that are known to result in the execution of specific segments of previously uncovered code [83]. An example of a symbolic-assisted fuzzer is SAGE (Scalable Automated Guided Execution) which is a proprietary tool that Microsoft uses in-house to search for vulnerabilities in their own software [1].

There are strengths and weaknesses to be considered for all of the approaches that we have examined. Creating a mutation fuzzer is generally quicker and easier than creating a generation fuzzer as far less knowledge is needed about the target application. Additionally, this means that one mutation fuzzer can often be used to fuzz a range of different target applications without modification [99]. A generation fuzzer however will either need modifying for each target or at least a new input model or grammar will need to be specified each time. It is also the case that as long as a sample of valid input can be obtained, even if the specification is proprietary and unavailable it is possible to carry out mutation fuzzing, whereas it may be impossible (or at least require a lot of reverse engineering skills) to carry out generation fuzzing.

The mutation approach however can run into problems with things like checksums where invalid inputs can be quickly rejected by the program under test without getting processed fully. In these cases a common approach is to comment out the sanity checks in the target if the source code is available. This will make it accept and process inputs even when items like checksums or the file's 'magic number' are invalid. If the source code of the target is not available then a specific test harness may need to be written to process the output of the fuzzer and fix checksums before forwarding it to the target. This is not a problem when using generation fuzzing as of course the fuzzer itself will be able to ensure that each test case is at least valid enough to pass any such tests.

Mutation fuzzers are also heavily dependent on the quality of the initial corpus of test cases used to seed them. Without an excellent corpus to start from they are unlikely to test as deeply as generation fuzzers. For

example, if a protocol being fuzzed consists of 100 different commands, but the sample we give to the mutation fuzzer only contains 5 of them, we will not have a very thorough fuzzing campaign. There may be certain parts of a file format or a protocol that are valid but are rarely seen in practice and so may be absent from a typical corpus. With a generation fuzzer we would supply a detailed specification of the protocol and hence the full set of commands would be included in test cases. Developing this model of the file format or network protocol can be very time consuming though [40] and it is also possible to miss out undocumented features if the model is not created using reverse engineering. In [94] undocumented HTTP responses were discovered in a corpus that was created by web crawling showing that sometimes mutation fuzzing can provide coverage that would actually be missed by generation fuzzing.

In [2] experiments were performed to compare the code coverage obtained from mutation and generation fuzzing in the context of targeting a PNG image file reader. It was found that several parts of the PNG format's specification were not in common use and therefore not present in any of the PNG files downloaded from the web. This means that significant parts of a typical PNG image viewer would never be executed when loading readily available images. The authors concluded that “applications often contain large sections of code that will only execute with uncommon inputs”. In this situation the experiments found that generating test cases resulted in 76% more code coverage than mutation of PNG files downloaded from the Internet [2].

No academic literature was found researching if the experimental results seen in [2] hold true for HTML files with respect to code coverage in web browsers. It may be that the common practice of downloading a selection of random web pages from the Internet to form an initial corpus for browser fuzzing [94] is actually missing out some valid HTML tags and attributes and therefore not getting the best code coverage possible.

2.4 Fuzzing Web Browsers

When fuzzing web browsers to find vulnerabilities it is common to focus on the rendering engine [44, 57]. This part of the browser is responsible for taking the HTML, CSS and images that make up a web page and turning them into what is displayed to the user. Popular rendering engines in use today include EdgeHTML, Trident, Gecko, WebKit and Blink (itself a fork of WebKit) [47]. The rendering engine is a common location to find exploitable UAF (Use-after-free) vulnerabilities [56].

There are two main approaches to fuzzing a web browser's rendering engine; file format fuzzing and DOM fuzzing. The file format fuzzing approach consists of simply applying the mutation or generation techniques that we have already examined to create malformed web pages and providing these to the browser as input. This is therefore applying a generic fuzzing approach and using the same methods that one might use to fuzz a PDF reader or a video player.

The DOM fuzzing approach applies techniques that are specific to web browsers [44, 59] and the actions are carried out by the browser itself through JavaScript. There are many variations but a typical approach is to crawl the DOM tree of a test case document and rearrange the nodes in various pseudo-random ways. Garbage collection is then triggered somehow and the process is repeated [59, 60]. This can result in the discovery and triggering of various memory corruption bugs.

DOM fuzzing tools in general have a problem with producing test cases to efficiently reproduce crashes when they are discovered. A typical DOM fuzzing tool builds up the DOM tree by randomly selecting HTML tags from an array and calling the document.createElement() method with each. The built-in PRNG in JavaScript (Math.random) is often used which cannot be explicitly seeded with a known value. The script then does some mutations to the DOM tree (again often involving the use of Math.random) which may cause a crash, and if the browser does not crash the script reloads the page and starts over. This continues for perhaps several hours until eventually the browser crashes. When the browser does crash there is not a

specific test case available with which to reproduce the crash.

Nothing can be written to disk from JavaScript within a web page for obvious security reasons. Most DOM fuzzers instead use the console.log() method from JavaScript to log messages to the browser's console [133]. This information can then be used to try to investigate what happened and reconstruct a test case. This is far from ideal compared to traditional fuzzing where any time a crash occurs the specific test case that caused it is available right away. The consequence of this has been that DOM fuzzers have been good at triggering bugs in browsers but investigating the bug to determine the root cause and fix it is often challenging.

2.4.1 Existing Browser Fuzzing Tools

Several tools exist that were created specifically to fuzz web browsers. We now examine some of the more successful ones and contrast their differing approaches.

The earliest browser fuzzing tool of note was Mangleme which was released in 2004 by Michal Zalewski [89]. This tool was a generation fuzzer that produced malformed HTML pages. It was implemented in C and deployed as a compiled CGI program that would run on a web server. By navigating to the CGI program's URL in the web browser to be tested, the page would repeatedly refresh and a different HTML test case would be processed each time. The test case generation worked by referencing a two dimensional array containing all the possible HTML tags and all the attributes of each tag. Malformed HTML pages were constructed by selecting random tags and adding random attributes from the array, while also scattering some random characters and bytes throughout. The whole tool consisted of about 200 lines of C. Despite the simplicity of this tool it found serious vulnerabilities in all the main web browsers of the time.

Radamsa is a general purpose mutation fuzzer, developed by researchers at the Oulu University Secure Programming Group (OUSPG). Although it is a general purpose fuzzer, it has been used effectively to target web browsers as documented in [100]. Radamsa uses model-inference assisted fuzzing. It attempts to build its own model of the input based on the valid inputs in the initial corpus that it is given. It then uses this model to apply mutations that are likely to be valid enough to be processed properly by the target rather than being immediately rejected as invalid. This allows the fuzzer to increase the depth of testing it is capable of whilst still not requiring the user to provide anything more than some valid input samples.

Radamsa was applied to web browser fuzzing in 2010 and 2011 as described in [100]. It was used to mutate HTML, CSS and JavaScript as well as PDF files, Flash and also image formats such as PNG and JPEG. All of these file formats were used as web browser test cases. The test cases were obtained first by random web crawling and then by downloading various conformance test suites. From this it was assumed that code coverage would be quite high, although nothing was done to actually measure this or try to increase it even further. It is possible that if this had been done the work would have been even more successful. Approximately 60 new bugs were found in a range of web browsers.

LangFuzz is a 2012 tool for fuzzing interpreters that takes a context free grammar [93] as input along with a test suite of valid sample progams and an additional corpus of programs that were seen to previously cause errors in the target application. The tool then combines aspects of mutation and generation fuzzing. It uses the grammar to parse the test suite and previous crashes and then recombines fragments of crash code into the test suite to create new programs in an attempt to find new issues [58]. For example, the tool was used to fuzz the Mozilla JavaScript engine by providing it with a context free grammar describing JavaScript, a JavaScript test suite and also a collection of JavaScript programs that had previously caused errors in the Mozilla engine. The newly generated JavaScript programs that LangFuzz created from this resulted in 105 new vulnerabilities being discovered in 3 months [58].

The success of the LangFuzz approach in [58] suggests that when forming an initial corpus for general

browser fuzzing it may help to include previous crashing test cases that are already known as well as public browser exploits and proof of concepts. By doing this it may be that new browser vulnerabilities which happen to be similar to old, fixed vulnerabilities can be revealed. The literature review did not reveal evidence that this idea has already been evaluated.

Grinder, released in 2012, is a distributed framework to support fuzzing Windows web browsers on a large scale [95]. It consists of a web application running on a centralised Grinder Server and any number of Grinder Nodes that carry out the actual fuzzing. As a framework, the aim is to provide an easy solution to problems such as documenting crashes, providing instrumentation, crash deduplication, restarting the browser as needed and related features rather than the fuzzing itself. The user must provide their own fuzzing program that creates the test cases.

GramFuzz was developed in 2013 by researchers in China [8]. Their work examined combining techniques from generation and mutation fuzzing while using HTML, CSS and JavaScript test cases to fuzz web browsers. The tool obtains a corpus of web pages from the Internet using web crawling and then builds a grammar tree from them. The grammar is then used to create mutations to the corpus that are likely to be correct enough to be accepted by the browser rather than being rejected immediately as may be the case with a pure mutation fuzzing approach. This is quite similar to the approach taken by Radamsa that we saw earlier. During testing GramFuzz found 36 new severe vulnerabilities in Firefox and Internet Explorer.

DOMfuzz [51] and Jsfunfuzz [52] are two fuzzers released by Mozilla at the end of 2013 and beginning of 2014. DOMfuzz employs a mix of mutation and generation techniques to carry out DOM fuzzing. It takes an initial corpus of pages and then applies techniques such as rearranging the DOM tree by moving nodes around. Jsfunfuzz is a generation fuzzer for testing JavaScript engines. It works by generating and executing random JavaScript function bodies. Unlike the JavaScript fuzzing that was performed with LangFuzz it therefore does not take an initial corpus but rather relies entirely on generation.

NodeFuzz is a modular cross platform browser fuzzing harness released in 2014 [53, 56]. It solves a lot of the same problems as Grinder but it is written in Node.js and is platform independent. The user is expected to supply their own test case creation modules and also their own instrumentation modules. These can therefore be specific to the exact browser being tested if necessary.

Finally, ShakeIt is a mutation fuzzer published in 2015 for use within a browser fuzzing framework [57]. The program takes a corpus of valid web pages and extracts tokens from them. It then swaps random tokens with each other within these pages to create new test cases. Memory corruption bugs can be exposed in this way as the rendering engine attempts to make sense of seeing valid tokens in invalid locations or in an invalid order.

This is by no means a comprehensive list of all existing browser fuzzing tools, but it does serve to highlight the typical functionality and approaches that have been tried over the years. It can be seen that while fuzzing research in general has given considerable attention to measuring and increasing code coverage, the work that has been done on browser specific fuzzing tools does not appear to have made this a priority. It can also be seen that there has been a trend to focus on generation fuzzing when attacking web browsers, one notable exception being the work done applying Radamsa to browsers [100]. The work in [100] proved that mutation of an existing corpus of web pages can be a successful approach to browser fuzzing, and had work been done to ensure the maximum code coverage of the initial corpus it could perhaps have been even more successful.

2.5 Conclusions

In this chapter we examined one of the main approaches that security researchers use to find software vulnerabilities. Competing approaches to fuzzing were examined, from simple random inputs all the way to

using genetic algorithms and taint analysis. The importance of measuring code coverage to evaluate the completeness of a fuzzing campaign was examined. Previous work on fuzz testing of web browsers was reviewed. Gaps in the literature were identified that provide opportunities for original research that can be carried out to form this project.

It was established for PNG files that samples typically obtained by web crawling will not contain all of the valid content that is found in the format's specification. In this situation generating test cases from a specification that includes every valid part of a PNG file was found to result in 76% more code coverage than mutation of a corpus of PNG files downloaded from the Internet [2].

It seems quite likely that the same situation exists for HTML files, although this has not been established from the literature review. It would therefore be useful to develop a tool that scans an existing corpus of web pages and determines if any HTML tags or attributes are not present. If experiments using this tool find that a corpus of typical web pages harvested from the Internet is missing certain rarely used elements of HTML then we can conclude that using a grammar to generate an initial corpus should improve code coverage for browser fuzzing sessions. Developing such a grammar based corpus generation tool that makes full use of the HTML specification would then be another of the project goals.

Just as increasing code coverage increases fuzzing success, it was also seen from the literature review that experimenting with new approaches to creating invalid inputs also tends to lead to the discovery of new bugs. This has especially been true of DOM fuzzing. Whenever a researcher pioneers a new technique for fuzzing the DOM tree that has not been used before in existing fuzzing tools it often triggers a number of previously undetected bugs in all web browsers. It would therefore be useful to develop a DOM fuzzer that corrupts the DOM in some ways that were not seen in the literature.

DOM fuzzers have traditionally had problems with not allowing crashes to be easily reproduced. This is therefore a problem area that could also be researched as part of this project.

Chapter 3: Aims and Objectives

This chapter states the overall aims of the project and also breaks these down into a list of nine objectives that need to be accomplished by the work in order to achieve these aims.

In Chapter 2 we identified gaps in the literature that suggest themselves as useful avenues of research. We examined how code coverage is currently used in general fuzzing and we also looked at tools specifically designed to fuzz web browsers. It was found that little work has been done on maximising code coverage when specifically fuzzing web browsers.

It was established that code coverage is extremely important when fuzzing any application. If a fuzzer creates test cases that only exercise a small percentage of the target's code base, it has no chance of detecting vulnerabilities in the unexecuted code. When fuzzing a web browser it is therefore important to use test cases that exercise as much functionality as possible and therefore give the best chance of exposing vulnerabilities. When using HTML test cases, it can be expected that the more HTML tags and attributes are present in the corpus, the more functionality will be exercised in the tested web browser when processing it.

We identified the need to run an experiment in order to determine whether a significant number of HTML tags and attributes are rarely used in practice and so will not appear in a typical corpus obtained by web crawling. If as expected this proves to be the case, there is also a need to develop a tool to create a corpus of web pages that would use all existing HTML tags and attributes and thus give optimal code coverage when later used with a mutation fuzzer.

It was established from examining historical fuzzing research that in this field trying new approaches often results in new bugs being found. Therefore in addition to our goal of increasing code coverage, we also have the goal of finding effective new ways of performing DOM fuzzing. Combining these two goals would be expected to result in new browser bugs being detected.

Therefore the aims of this project are the following:

Firstly to increase the code coverage obtainable during fuzzing of web browsers by developing a novel method of generating a corpus of web pages. Secondly to develop novel new methods of DOM fuzzing and apply these to the test cases we have generated that give good code coverage. The research will aim to evaluate the ultimate effectiveness of this by fuzzing a selection of real web browsers and documenting any crashes that result.

The objectives of the proposed work are to:

1. Identify and develop a method of parsing a corpus of web pages and identifying which HTML tags and attributes are not present in it.
2. Develop a software tool called HTMLscan to automate this method.
3. Apply the HTMLscan tool to a corpus of web pages obtained by web crawling. We expect this to show the inadequacy of such a corpus and therefore show the need to create better corpora.
4. Identify and develop a method that can be used to generate a corpus of web pages that will make use of all possible HTML tags and attributes, and thus cause as much code to be executed as possible when it is processed by a web browser.
5. Develop a software tool called HTMLgen to automate this method.
6. Evaluate the HTMLgen tool firstly by creating a new corpus with it and then analysing this with

HTMLscan to show that no tags or attributes are missing. Additionally, measuring actual code coverage obtained when processing the entire generated corpus in a suitably instrumented web browser would be helpful.

7. Identify and develop new methods that can be used to fuzz the DOM tree after a browser has parsed a web page, using one or more mutation strategies that have not been used before.
8. Develop a software tool called HTMLfuzz to apply these methods to the high coverage test case corpora generated by the HTMLgen tool. Methods of making it easier to reproduce crashes as compared to traditional DOM fuzzing tools should also be considered here.
9. Evaluate the HTMLfuzz tool simply by fuzzing a selection of web browsers with it and observing if any crashes are caused and any new bugs discovered.

Objectives 1, 4 and 7 are achieved in Chapter 4 which describes the proposed methods of the three tools in detail. The technical details of how to evaluate an existing corpus, generate a better corpus and also how to fuzz the DOM tree with the better corpus are documented in full.

This leads to the necessity of developing a suite of software tools to implement the methods as stated in objectives 2, 5 and 8. A high level software approach is briefly described in Chapter 4 while the implementation of this suite of tools along with the challenges faced are described in Chapter 5.

Objectives 3, 6 and 9 involve performing experiments to gather empirical evidence of the suite's ability to solve the stated research problems and provide gains in code coverage while also applying new DOM fuzzing methods. The experimental design is covered in Chapter 4 while the performing of the experiments is documented in Chapter 6 and the results are evaluated in Chapter 7.

Chapter 4: Design and Methodology

This chapter examines the proposed approach for solving the research problems and meeting the nine objectives that were described in Chapter 3. Different approaches to the problems are considered and design decisions are justified.

4.1 Proposed Methods

It was shown in Chapter 2 that there is a need to develop a method of evaluating an existing corpus of web pages and identifying missing HTML. This is Objective 1 from those that we listed in Chapter 3. It is therefore necessary to execute the following three steps.

1. Specify a list of all HTML tags and attributes that exist. It is important to note that since the ultimate goal is to help improve code coverage, we are interested in every HTML tag or attribute that triggers distinct code to be executed in any web browser. We must therefore go beyond simply using the latest HTML 5 specification. It is necessary to research and include old, obsolete HTML. A good corpus would include all of these tags and attributes.
2. Read and parse each web page from the corpus being analysed. Build a set of all the HTML tags and attributes that appear in the corpus.
3. Subtract the set of HTML tags and attributes that were found in the corpus from the set of all HTML tags and attributes that make up our specification. The remaining HTML tags and attributes are of course those that we would like to see in our corpus but which are not present.

While the common approach to creating a HTML corpus for fuzzing web browsers is to use web crawling to download a random selection of web pages from the Internet [94], it is likely that using the above method to evaluate such a corpus will prove this to be highly inadequate. This therefore leads us to Objective 4 from those we listed in Chapter 3.

Objective 4 is to “identify and develop a method that can be used to generate a corpus of web pages that will make use of all possible HTML tags and attributes, and thus cause as much code to be executed as possible when it is processed by a web browser”. In order to achieve this it is necessary to execute the following steps.

1. Select tags and attributes from our full HTML specification in a pseudo-random way and combine them to create a fairly short HTML web page. Repeat this step many times in order to ensure that all tags and attributes appear at least once in the total collection of pages, and in order to provide plenty of variety in the corpus.
2. Write each of the generated HTML pages to disk in order to create a high quality corpus that can be used to seed a third party browser fuzzing tool.

This method differs from the work done by Charlie Miller in [2] regarding fuzzing using PNG files. In that work, after it was seen that PNG files harvested from the web were incomplete compared to the PNG specification, mutation fuzzing was abandoned in favour of generation fuzzing based on the official PNG specification. The proposed approach that we are applying to HTML test cases instead consists of using generation to create a syntactically valid corpus before proceeding with mutation fuzzing.

The proposed approach has two advantages. Firstly, by combining both generation and mutation we get the strengths of both. We can generate an initial corpus of valid test cases that makes full use of the HTML

specification and gives superior code coverage while still having the simplicity of then applying a mutation engine to carry out the actual fuzzing, including using the generated corpus to seed a third party mutation fuzzer such as Radamsa [129].

Secondly, by our generation tool using not just the latest, official HTML 5 specification but also the old, undocumented and proprietary HTML tags uncovered by our research we benefit from combining the strengths of generating a corpus from a specification with the strengths of harvesting one from the web. The problem with generating a corpus based on only the official, documented specification is that certain undocumented features may be missed out. This results in missed opportunities and less code coverage.

As a historic example of this principle, in [94] it was documented that crawling the Internet to build a corpus of HTTP responses from public web servers resulted in the discovery of an undocumented HTTP response header coming from a particular IIS server. When fuzzing various web browsers by mutating the HTTP protocol it was seen that including this undocumented header in the corpus increased code coverage in the Internet Explorer browser. In turn, mutating this header led to the discovery of a UAF vulnerability in Internet Explorer. This vulnerability had been missed by previous fuzzing campaigns simply because the HTTP response header in question was undocumented and so did not appear in any generated corpus. Only when web crawling was used to collect samples of HTTP responses from the wild was it discovered. Our approach applies this lesson by making an effort to research undocumented, rare and obsolete HTML tags and including these in our generation engine.

Objective 7 is to “identify and develop a method that can be used to fuzz the DOM tree after a browser has parsed a web page, using one or more mutation strategies that have not been used before”. In order to achieve this it is necessary to execute the following steps.

1. Generate a pseudo-random, syntactically valid HTML test case using the same generation engine that was created for our corpus generation tool. This guarantees good code coverage and variety of test cases.
2. Add a JavaScript to the test case that implements our proposed DOM fuzzing strategies. The intention is that once the HTML test case has been parsed by the web browser and a DOM tree has been constructed, this JavaScript will be executed and will use DOM API calls to manipulate the tree in a way calculated to trigger bugs.
3. Deliver the finished test case to the web browser that is being tested. Allow the JavaScript to finish executing.
4. Return to Step 1 and keep looping like this until the web browser crashes.

This method allows the generation engine described earlier to be used not only to generate a corpus on disk to use with a third party mutation fuzzer, but also to provide its code coverage benefits to the process of DOM fuzzing. The other key aspect to making this succeed is of course to use novel DOM fuzzing strategies rather than applying the exact same techniques that already appear in the literature and have been used already to fuzz the popular web browsers. The proposed DOM fuzzing strategies are documented later in this Chapter. By combining novel DOM fuzzing strategies with our high coverage test case generation engine, we hope to discover web browser bugs that have been missed by the previous efforts of security researchers.

4.1.1 HTML Specification

The first step in the proposed methodology is to acquire or create a full specification of the HTML language. This is necessary in order to be able to identify which parts of the markup language are absent from a given HTML corpus, as well as to know what should appear in an ideal corpus as we generate it. As a browser's

rendering engine will contain separate functionality for handling each HTML tag and attribute that it parses it is necessary to make full use of all available HTML tags when creating a corpus in order to get better code coverage. A web page that contains 25 different HTML tags can be expected to result in more code coverage when parsed by a web browser than one containing only 5 of those HTML tags.

As of 28th October 2014 the current version of the language is HTML 5.0 [109] as published by the W3C (World Wide Web Consortium). An official list of all the HTML5 tags and their attributes is available online [110]. This can be used as the starting point for our specification.

As mentioned in Section 4.1 it is not enough to only rely on the HTML 5 specification though. As the HTML language has evolved since its beginnings in the early 1990s various tags have been removed from the specification. Many web browsers have not removed the corresponding rendering functionality that handles these obsolete HTML tags though. It is therefore possible that vulnerabilities exist in some browsers in functions that deal with old HTML tags that no longer appear in the official specification. In order to maximise code coverage and increase the effectiveness of a fuzzing campaign it is therefore important that all these old tags are present in the corpus.

Research into this topic online turned up a number of sources of information about obsolete HTML tags [122, 123, 124]. As well as old tags that have been removed from the specification this also revealed the existence of proprietary tags created by Microsoft and Netscape that were actually never in the official HTML specification. This research resulted in the identification of the following 44 HTML tags that are not in the HTML 5 specification but which nonetheless can be expected to trigger unique code paths in at least some web browsers: *acronym, app, applet, basefont, bgsound, big, blink, center, command, comment, details, dialog, dir, frame, frameset, hgroup, hp0, hp1, hp2, hp3, hype, ilayer, image, isindex, key, layer, listing, marquee, menu, menuitem, multicol, nextid, nobr, noembed, noframes, plaintext, server, sound, spacer, strike, summary, tt, typewriter, xmp*.

An ideal HTML corpus would therefore contain all 44 of those obsolete tags as well as the 108 tags that appear in the HTML 5 specification. In addition to this a total of 147 HTML attributes were identified from both the HTML 5 specification and the available details of the obsolete tags.

We therefore have a list of 152 HTML tags and a list of 147 HTML attributes that can be used as a HTML specification for evaluating a corpus.

Ideally when fuzzing web browsers we would consider not just the HTML content of the corpus but also CSS (Cascading Style Sheets) as this is also parsed by the browser's rendering engine and can be used to expose vulnerabilities [44, 57]. JavaScript could also be included as this can be used to fuzz the browser's JavaScript interpreter. There are also several different formats of font, document, image, audio, video and video subtitle files that can be referenced from HTML and then processed by the web browser [125]. These are all beyond the scope of this project though. A future version of the tool suite could include some kind of specification of all these other data sources and apply the same kind of evaluation of them as we are attempting for the HTML. Due to time constraints and to avoid overcomplicating things it is sensible to deal only with HTML for this project and the first version of the tool suite though and prove through experiments that the overall concept is sound before applying it to other sources of browser input than HTML.

4.1.2 Evaluating a Corpus

Given an existing corpus of HTML files that we wish to evaluate against our HTML specification, it is necessary to first read each file and parse it.

After opening each web page file and reading its content, it is necessary to carry out the process of lexing or lexical analysis. This involves using regular expressions to group individual ASCII characters together into

discrete tokens [134]. Once each token has been identified and classified, the process of parsing can be carried out. This builds up a parse tree data structure based on the specific tokens that appear in the input and their order [135]. Once this has been done we are no longer simply looking at the content of a text file, but rather a data structure that can be iterated over with the HTML tags and attributes clearly identified as such.

After reading and parsing each file in the corpus we can build a list of all the HTML tags and attributes that are present in it. It should then be a fairly simple matter to see which tags and attributes appear in our specification but do not appear in the corpus. These missing HTML tags and attributes should be reported to the user and it is recommended that the corpus be expanded by adding this missing HTML.

4.1.3 Generating a Corpus

Objective 4 is to “identify and develop a method that can be used to generate a corpus of web pages that will make use of all possible HTML tags and attributes, and thus cause as much code to be executed as possible when it is processed by a web browser”.

There are two obvious approaches that can be used for generating the new HTML files from our specification. The simpler option is to take an approach similar to Mangleme [89] where all the HTML tags and their attributes are stored in a two dimensional array and then picked from it in a random order and placed in the pages. This is relatively easy to implement but will not generate web pages that are syntactically correct. This is because in HTML it is not valid to just use tags in absolutely any order [109]. As a simple example, <*head*> should be used in the page before <*body*> is and not the other way round.

There is, however, an argument in favour of using this method. Rendering engines within web browsers are not strict about HTML correctness and will do their best to render anything even close to correct [114]. Therefore even if the web pages generated by our tool have tags appearing out of order, all major web browsers will still try to render them rather than simply refusing to process them.

The second approach is to formulate and use a grammar to generate HTML that is syntactically correct and valid. A context-free grammar (CFG) [93] for HTML 2.0 in Backus–Naur Form (BNF) is available online [111]. This could be used as a starting point and be adapted and updated by appropriately adding in production rules for the other tags and attributes from our specification in Section 4.1.1. By incorporating a grammar into our tool and following its rules we can be sure to make use of tags in the correct order and hence generate new web pages that are valid HTML.

Even though the first approach would produce HTML that the browser's rendering engine would attempt to process the second approach is clearly better for the following reason. It is true that whichever mutation fuzzer is used will generate malformed HTML from the corpus by the very nature of fuzzing. However, the initial corpus that our own tool generates should still be made up of samples of valid input rather than HTML that is already malformed and incorrect. The mutation engine that is applied to the final corpus should be entirely responsible for the introduction of incorrect syntax.

We will therefore use the more challenging of the two possible approaches and construct our own grammar that includes all of the HTML tags and attributes that appear in our specification. The HTMLgen tool will use this grammar and will follow its production rules, selecting pseudo-random branches whenever multiple options are available for expanding a non-terminal symbol. This will allow the generation of rich, varied corpora that make use of the full range of available HTML tags and attributes. Such corpora can be expected to be ideal for fuzzing and to result in excellent code coverage.

4.1.4 DOM Fuzzing Strategies

Objective 7 is to “identify and develop new methods that can be used to fuzz the DOM tree after a browser has parsed a web page, using one or more mutation strategies that have not been used before”. In this section we describe the series of DOM fuzzing strategies that we intend to implement in our tool and apply to each test case. Some are novel, others are 'tried and tested' strategies that are known to reveal bugs and hence should be included as well.

The HTMLfuzz tool will apply the following steps after loading each HTML test case.

1. Use NodeIterator and TreeWalker to create logical views of the DOM tree. As we make changes to the underlying DOM tree in the following steps, the browser will dynamically update the logical views to keep them consistent with the underlying DOM tree. This can involve a lot of pointer arithmetic and increases the chances of memory corruption bugs being revealed.
2. Crawl the DOM tree and select four random element nodes. For each node set all of its attributes to have values randomly selected from a list of values likely to trigger issues. This list will contain items such as a very long string that could trigger buffer overflows, as well as specific positive and negative integers that are chosen for their ability to trigger integer overflows.
3. Select a random element node. Create two references to that node. Use one of the references to remove the node from the DOM tree. Trigger garbage collection and then attempt to use the other reference to update the node. This has the possibility of triggering use-after-free vulnerabilities.
4. Select two random element nodes. Attempt to swap them with each other.
5. Build an array of all the element attributes that exist in the entire tree, along with their values. Select a random element node and attempt to add the entire array of attributes to it.
6. Build an array of references to each element node. Randomly shuffle the array. Iterate through the array cloning each element node and attempting to add each cloned node as a child to a random element node on the DOM tree.
7. Select a random element node. Add 20,000 new text nodes to it as children. These text nodes should alternate between containing a string and being empty. Once this is completed, call the document.normalize() method.
8. Select two random element nodes. To the first element, attach a MutationObserver that when triggered will try to remove the second element from the DOM tree and trigger garbage collection. To the second element, attach a MutationObserver that when triggered will try to remove the first element from the DOM tree and trigger garbage collection. At this point we can make any change to one element immediately followed by the other in order to potentially trigger use-after-free vulnerabilities.

These steps are all designed to cause the browser to carry out actions that could expose underlying memory corruption bugs. They have the best chance of success when applied to a large and varied corpus of test cases that makes full use of all available HTML tags and attributes. Combining it with the grammar based generation engine discussed in section 4.1.3 will therefore be quite suitable.

A note is necessary here regarding garbage collection, as some of the DOM fuzzing strategies described above are more likely to trigger bugs when the browser carries this out. Most web browsers have their own custom heap implementation and their own memory manager rather than using the operating system's own functionality. When targeting Internet Explorer and Edge there is a JavaScript function called CollectGarbage() that can be called in order to trigger garbage collection. This function is not implemented

on other browsers such as Chrome or Firefox though and so we need an alternative method of triggering garbage collection. The method we will use is essentially the same as heap spraying, although in this case we are not using it to exploit a vulnerability. If we build up a very long string in JavaScript, this will cause more and more memory to be allocated on the heap. This can be expected to trigger the browser's custom memory management functionality to carry out garbage collection in an attempt to free up more memory.

4.2 Software Approach

It is necessary to design and develop three software tools (HTMLscan, HTMLgen, and HTMLfuzz) that will automate the proposed methods that were described in Section 4.1. Developing these tools will achieve Objectives 2, 5 and 8 as listed in Chapter 3. The implementation of these tools will be described in detail in Chapter 5 but here we explain and justify the high level approach.

The tools should all be command line only tools without a GUI (Graphical User Interface). User interaction will be relatively simple with only the need to specify various options such as a TCP port to listen on and/or input or output directories. Adding a GUI to such a tool would be to add unnecessary complication and overhead. The user can provide the necessary options on the command line quite easily.

The tools should all be cross-platform as fuzzing web browsers is a security research task commonly carried out on all operating systems and not restricted only to one [40]. Suitable programming languages for developing cross-platform software include Java, Python and Ruby.

4.3 Experimental Design

Once the suite of tools has been developed it will be necessary to run experiments in order to evaluate each tool's performance.

In order to achieve Objective 3 as listed in Chapter 3, it is necessary to use the HTMLscan tool to discover how much of the HTML specification appears in a typical corpus of web pages scraped from the Internet.

In order to achieve Objective 6 it is necessary to generate a corpus using HTMLgen and then evaluate it with HTMLscan. It will also be useful to somehow process the corpus with an instrumented web browser to record actual code coverage. Using the corpus generated by HTMLgen to seed a third party mutation fuzzer and carry out some browser fuzzing sessions would also be a worthwhile experiment.

In order to achieve Objective 9 it is also necessary to use HTMLfuzz to fuzz a selection of web browsers on both Windows and Linux, and to analyse any crashes that result.

The experiments themselves along with their results are documented in Chapter 6. This section documents the tools and the environment that should be set up in order to be able to later run relevant experiments.

4.3.1 Web Browser

In order to evaluate a given HTML corpus, it is useful to load every test case included in it into a suitably instrumented web browser and observe code coverage.

In order to be able to instrument the browser easily it is best for it to be open source. This leads to obvious options of Firefox [117] and Chromium [118], the source code for both of which is readily available for download. It is also possible to instrument web browsers for which we have no source code, such as Microsoft's Internet Explorer/Edge browser [119], but this is generally more complicated than when we can add the instrumentation at compile time.

There is also the issue of how to automate or script the instrumented web browser to make it load each page

from the corpus. In Chapter 2 we looked at Grinder [95] and NodeFuzz [53, 56], which are both browser fuzzing frameworks that would be suitable for this purpose. Both tools have the capability to handle repeated loading of test cases into a web browser.

An alternative approach is to process the corpus using a headless web browser. This is essentially a web browser without a GUI that is controlled from the command line. It would be easy to automate the repeated loading of test cases into a headless browser as it can be controlled from a scripting language such as bash. This would be more efficient than trying to automate a full web browser such as Firefox and would likely allow test cases to be processed at a higher rate. By concentrating on only the rendering engine we have the bulk of the browser's attack surface [44] without the inefficiency of the GUI and other features.

Based on this reasoning, PhantomJS [70] was selected as the target program that we will use in this project for measuring code coverage of our various test corpora. It is an open source, headless, scriptable version of the WebKit rendering engine [120]. It should therefore be possible to compile it to include the necessary instrumentation and then script it to process multiple web pages rapidly without manual intervention.

The WebKit rendering engine is used in Apple Safari (the default browser on the iPhone) [120] and a fork of it known as Blink is used in Google Chrome [121]. Creating a test corpus that provides good code coverage of WebKit is therefore a very useful first step towards a fuzzing campaign that would reveal vulnerabilities in popular modern web browsers.

Other options that were considered as alternatives to PhantomJS include TrifleJS [115] which is a headless version of Internet Exporer's Trident rendering engine, and SlimerJS [116] which provides an API to the Gecko rendering engine from Firefox. TrifleJS was rejected due to not being open source. SlimerJS was rejected due to not being truly headless. It actually needs Firefox itself to be installed and graphical windows are displayed which makes it a less efficient option than PhantomJS.

4.3.2 Measuring Code Coverage

It is necessary to instrument the selected web browser so that code coverage can be recorded. As we have selected an open source browser that we will be compiling on Linux we can make use of gcov. As explained in Section 2.2.1 gcov is activated simply by using the “-fprofile-arcs -ftest-coverage” option when compiling an application with gcc [107].

Using this method it should be possible to compile a version of PhantomJS that records code coverage whenever executed. The resulting report after each execution will show which lines and branches were executed in that session [107].

4.3.3 Fuzzing Environment

Once the HTMLgen and HTMLfuzz tools have been developed it will be necessary to carry out some actual fuzzing campaigns.

The proposed environment for this consists of two separate virtual machines as shown below.

Debian Linux 8 VM

- Popular web browsers: Firefox and Chromium, both built with ASan instrumentation
- Niche web browsers: Netsurf, Xombrero, Midori, Opera
- Headless web browser: PhantomJS [70] built with gcov instrumentation
- Third party fuzzing tools: Radamsa [129]

- Debuggers: GNU debugger (gdb) with the 'exploitable' plugin [39]

Windows 10 VM

- Popular web browsers: Firefox, Chrome, IE11, Edge
- Niche web browsers: Midori, Opera, K-Meleon, Pale Moon, Avant, Maxthon
- Debuggers: Microsoft Debugging Tools, Immunity Debugger, OllyDbg
- Crash analysis tool: BugID [136]

One important point to note is that when fuzzing campaigns are running, the VM should not have access to the Internet. The reason for this is that many web browsers, upon experiencing some kind of error state, attempt to automatically submit a report back to the software vendor. Such a report would usually include enough debug information for the vendor to investigate what happened and possibly fix the bug.

A second important note, specific to fuzzing on the Windows 10 VM, is that it is good practice to enable PageHeap [137] for each specific browser executable before it is started. This can be done by typing the following command in an Administrator command prompt.

```
gflags /p /enable c:\path\to\browser.exe /full
```

This acts a little bit like Asan on Linux by causing an exception to be raised if any heap memory corruption occurs. Memory is poisoned at the boundaries of each heap allocation so that any attempt to access memory beyond the bounds is immediately detected. Unfortunately some browsers implement their own memory management instead of using the operating system and so PageHeap has no effect on them. If in doubt, it is best to always enable PageHeap anyway since it cannot do any harm and sometimes it will aid in detecting bugs.

4.4 Conclusions

This chapter detailed how all of the objectives from Chapter 3 will be achieved. The proposed methodology was explained as well as a high level approach to developing the software to automate the methods. An experimental design was then proposed that will allow the software tools to be evaluated.

The next chapter documents the implementation of the software as well as the main problems that were encountered and how they were overcome.

Chapter 5: Implementation

This chapter documents the implementation of the suite of tools described in Chapters 3 and 4. Rather than describe every part of every tool in detail, we focus upon the most interesting aspects of the software as well as the main challenges that occurred and how they were overcome.

All of the software was developed in the Python programming language using version 2.7 of the Python interpreter. Using Python allowed for rapid development of applications due to the excellent number and variety of libraries available to assist in almost any task. It also allowed for the development of tools that can run on either Linux or Windows without any modifications.

5.1 HTMLscan

The HTMLscan tool reads and parses an existing HTML corpus and generates a report showing which tags and attributes are missing from it.

As described in Section 4.1.1 a list of 152 HTML tags was first created along with a list of 147 HTML attributes. These were stored in the Python code as separate lists.

In order to parse each page in the corpus, the BeautifulSoup Python library [126] was used. This should be installed before attempting to run the HTMLscan tool. This can be achieved by simply running the command 'pip install beautifulsoup4'.

BeautifulSoup is a Python library “for navigating, searching, and modifying a parse tree” [126] which allows the developer to manipulate various kinds of documents. For handling HTML documents it uses the built-in html5lib Python library, which handles the actual parsing.

The following snippet of Python code taken from the HTMLscan tool shows how this library was used to achieve our goal. The first line reads a single HTML file from disk (pointed to by subdir, file) and parses it into a tree. The for loop then adds all the HTML tags appearing in the document to a list called corpusTagList and all the attributes to a list called corpusAttributeList.

```
# Parse the current HTML file
soup = BeautifulSoup(open(os.path.join(subdir, file)), 'html.parser')

# Add the tags and attributes from this HTML file to the appropriate lists
for tag in soup.findAll():
        corpusTagList.append(tag.name)
        corpusAttributeList.append(tag.attrs.keys())
```

After this routine has been executed for every file in the corpus it is a simple matter to compare corpusTagList to the list of 152 HTML tags, and corpusAttributeList to the list of 147 attributes that are hard-coded into the tool. A brief report is produced listing which HTML tags and attributes are not present in the corpus.

The HTMLscan tool takes a single argument on the command line; the directory containing the corpus to read in.

A sample output is shown in the screenshot in Figure 1.

```
Number of web pages in corpus: 30

Number of HTML5 tags in specification: 108
Number of obsolete HTML tags in specification: 44
Total Number of HTML tags in specification: 152

Number of HTML5 tags found in corpus: 98
Number of HTML5 tags missing from corpus: 10
Missing HTML5 tags: set(['samp', 'span', 'bdi', 'kbd', 'dfn', 'u', 'sup', 'small', 'strong', 'cite'])

Number of obsolete HTML tags found in corpus: 35
Number of obsolete tags missing from corpus: 9
Missing obsolete HTML tags: set(['multicol', 'big', 'image', 'hp0', 'blink', 'menuitem', 'spacer', 'key', 'strike'])

Number of HTML attributes in specification: 147
Number of HTML attributes found in corpus: 144
Number of HTML attributes missing from corpus: 3
Missing HTML attributes: set(['async', 'radiogroup', 'defer'])
```

Figure 1: HTMLscan results

This report that is given as HTMLscan's output tells the user which HTML tags and attributes should be added to the corpus in order to increase code coverage.

5.2 HTMLscrape

Once HTMLscan had been implemented it was necessary to use it to evaluate a corpus that had been obtained by web crawling. The aim was to find out if typical web pages that can be found online are usually missing various HTML tags and attributes and if therefore a corpus obtained by web crawling would likely lead to big gaps in code coverage.

It was found that manually building a corpus of random web pages by downloading them from the Internet using a web browser was very labour intensive and quite tedious. Therefore, although not planned in Chapter 4, it was determined that quickly developing an extra tool that automates this process was worthwhile.

The HTMLscrape tool takes two arguments on the command line; a directory to save the corpus to and the number of pages that are desired for the corpus.

The tool makes use of the Google library [127] which allows a Python script to run searches on Google and receive the search result URLs. The tool has a hard-coded list of 10 unrelated keywords that will be searched for. The user is advised to add more of their own. The idea is simply that by running multiple searches on Google for a selection of unrelated keywords the resulting list of URLs appearing in the search results should be quite diverse. Different types of web sites from different industries and sectors should appear in the results. It is expected that this will give a good deal of variety in the downloaded corpus as the pages will have been created by different people for different purposes.

Once the Google searches have been executed and a list of the desired number of URLs has been obtained from the search results, the wget library [132] is used to download each URL. Each web page is saved in the output directory with a filename of the form page1.html where the 1 is a counter that is simply incremented for each downloaded page.

5.3 HTMLgen

The main feature of HTMLgen is a Context-Free Grammar [93] representing the HTML language and including all of the tags and attributes that appear in the HTMLscan tool's lists.

This grammar is implemented in two stages. Firstly, there is a class that represents a Context-Free Grammar, and that provides methods for both adding production rules and generating random sentences. Secondly, once the class is instantiated to create a new object there is a large list of production rules that are added to it.

The grammar class is based on Python code from Eli Bendersky [128] and has two methods. The first is for

adding a new production rule to the grammar as shown in the code snippet below.

```
# Add a new rule to the grammar
def newRule(self, nonTerminal, rightHandSide):
        prodRules = rightHandSide.split('|')
        for prodRule in prodRules:
                self.productionRules[nonTerminal].append(tuple(prodRule.split()))
```

It can be seen that the class has a property called productionRules which is a dictionary whose keys are the non-terminal symbols of the grammar.

The second method that the class has generates a random sentence from any given starting symbol by using recursion and following production rules until no non-terminal symbols are left. Whenever a production rule offers multiple choices for expanding a non-terminal symbol, one is picked at random. This method is shown in the code snippet below.

```
# Generate web page by recursively expanding non-terminal symbols using random choices until there are none left
def generateWebpage(self, symbol):

        randProdRule = random.choice(self.productionRules[symbol])
        webpage = ''

        for gramSymbol in randProdRule:
                if gramSymbol in self.productionRules:
                        webpage += self.generateWebpage(gramSymbol)
                else:
                        webpage += gramSymbol + ' '
        return webpage
```

With this class we have everything necessary to build up a grammar and then use it to generate random web pages.

The production rules were created firstly by taking the Backus–Naur Form grammar of HTML 2.0 that was already obtained [111] as a starting point. This grammar did not include any attributes, only the HTML tags. Therefore the documentation for each tag was referred to and used to add all possible attributes along with sensible values.

For each tag listed in the HTMLscan tool but not already appearing in the grammar, the documentation of that tag was also referred to and an appropriate production rule was added. This again included making full use of all possible attributes and setting them to have sensible values. This task was quite time consuming, but once finished it gave the capability of generating random, syntactically valid web pages that use HTML drawn from the entire list of tags and attributes that have ever existed.

The HTMLgen tool takes two arguments on the command line; a directory to save the corpus to and the number of pages that are desired for the corpus. Using it is therefore exactly the same as using HTMLscrape, except now pages are generated from an internal grammar instead of being harvested from the web.

When the tool is executed, it calls the method to generate a random sentence with **`html_document`** as the starting symbol. This is the symbol in the grammar that represents the root of a HTML document (essentially the <html> tag) and therefore using that as the starting symbol results in a sentence that is a full, syntactically correct HTML web page. This page is written to the output directory in the same way as HTMLscrape. The web page is saved in the output directory with a filename of the form page1.html where the 1 is a counter that is simply incremented for each generated page. The process is repeated until the requested number of HTML pages has been generated.

As long as a sufficient number of pages were generated (the definition of “sufficient number” in this context will be investigated in Chapter 6) the resulting corpus should be very rich and ideal for use as input to a

mutation fuzzing tool, due to providing high code coverage of any HTML parsing and rendering functionality in the web browser.

5.4 HTMLharness

Once HTMLgen had been implemented, it was possible to pass the output corpus of this tool to a third party mutation fuzzer in order to create a second corpus of mutated HTML files ready to fuzz a web browser with. Two examples of suitable tools for mutating the output of HTMLgen are Radamsa [129] and ShakeIt [57].

In order to use the resulting mutated corpus, it is of course necessary to have some kind of fuzzing framework or test harness to feed the corpus to the web browser. Grinder [95] and NodeFuzz [53, 56] were mentioned in Section 2.4.1 and these were looked at again. It was felt that a simpler, light-weight solution would be a useful part of our tool suite, and this led to the rapid development of another Python tool.

HTMLharness uses the Twisted library [130] to create a HTTP server on localhost on a TCP port chosen by the user. The browser to be tested is then pointed at this HTTP server. Each GET request from the browser results in another HTML page from the corpus being read in from the filesystem and returned to the browser in a HTTP response.

The harness adds two meta tags to each web page test case before passing it to the browser. The first causes the web browser to reload the page after one second, which in turn causes the web server to read and deliver the next test case, and causes the process to continue until all files in the corpus have been loaded by the browser.

```
<meta http-equiv="refresh" content="1">
```

The second is a meta tag to instruct the web browser to disable caching.

```
<meta http-equiv="cache-control" content="no-cache">
```

This is used simply to make sure that when the browser reloads the page it does not use any cached content (such as the previous test case) but instead requests it all again from the web server, and hence receives the next test case.

The HTMLharness tool takes two arguments on the command line; the directory to read the corpus from and the TCP port to bind the HTTP server to. This simple tool thus provides an easy way of making a web browser load and process every page in a corpus of any size.

The HTMLharness tool keeps a log file recording each test case that is loaded and the order in which they are loaded. When the web browser crashes, this log file can be used to ascertain which test case was last to be loaded before the crash.

5.5 HTMLfuzz

The HTMLfuzz tool combines aspects of HTMLgen with aspects of HTMLharness, and also adds a JavaScript implementation of the DOM fuzzing strategies that were described in Section 4.1.4. The resulting tool therefore binds a HTTP server to a TCP port on localhost for the browser to connect to, then serves up randomly generated HTML test cases along with a fixed JavaScript. The web browser parses the HTML test case and builds the DOM tree, then executes the JavaScript which manipulates the DOM tree in ways designed to trigger bugs. The final thing that the JavaScript does is reload the page to cause the next test case to be generated. This process therefore continues until either it is terminated by the user, or the web browser crashes.

The HTMLfuzz tool takes a single argument on the command line; the TCP port to bind the HTTP server to.

To implement HTMLfuzz a copy was first taken of HTMLharness to use as the starting point and to modify as needed. The web server part of HTMLharness was kept, but the render_GET method, which is executed whenever a client sends a GET request to the server, was modified. Instead of reading the next web page from disk upon receiving a GET request, the requirement was to generate the next web page using the grammar functionality from HTMLgen.

The grammar from HTMLgen was therefore added to HTMLfuzz simply by copying over the htmlGrammar class and all the production rules. The DOM fuzzing JavaScript was also added to the grammar as a single production rule that is always included in every generated test case. It is the first thing to be inserted immediately after the <body> tag in each test case.

Since the web server functionality and the HTML grammar had both already been developed by this point and were simply copied over from HTMLharness and HTMLgen with minimal alteration, the main work for this tool consisted of developing the JavaScript to implement the DOM fuzzing functionality. Work was also done to attempt to overcome the shortcomings of previous DOM fuzzing tools regarding reproducibility of crashes.

5.5.1 Reproducing Crashes

As explained in Section 2.4 there has been a long term issue with DOM fuzzing regarding reproducing crashes when they occur [133]. The HTMLfuzz tool represents a step forward over previous DOM fuzzers in terms of reproducibility of crashes.

Existing DOM fuzzers typically consist of a single page of JavaScript that the user loads into the browser to be tested. The JavaScript itself builds a random DOM tree to act as a test case by repeatedly selecting random HTML tags from an array and using the document.createElement method. The JavaScript then applies various mutations to the DOM tree in an attempt to trigger bugs and eventually reloads the page to start this whole process again from the beginning. For security reasons, JavaScript in a web page running inside a web browser cannot write files to disk. The typical DOM fuzzer therefore cannot write static test cases to disk to allow crashes to be easily reproduced. The best it can do is send messages to the browser's console using the console.log() method [133]. When the DOM fuzzer has been running for a while and the browser eventually crashes, no unique test case is available to trigger the exact same browser state that led to the crash. All that is available is information about the fuzzer's state that was logged to the browser's console, which is less than ideal.

If the DOM fuzzer is being used with the Grinder framework [95] and has been integrated properly to use Grinder's logging facility, there is a Ruby script available as part of the framework that attempts to reconstruct a test case from this log. This only tries to reproduce the last test case before the crash though, rather than a sequence of test cases, which is sometimes necessary.

When implementing the HTMLfuzz tool, thought was given to how to improve this situation. Because we are generating the HTML test cases from a grammar in our Python, instead of picking random HTML tags using JavaScript within the browser process, it is of course possible to write the current HTML test case to disk immediately before serving it to the web browser.

Each time HTMLfuzz is started up, the Python PRNG (pseudo-random number generator) is seeded with the current time and this same seed is used as the directory name for storing test cases in. This is shown in the code snippet below.

```
randomSeed = time.time()
random.seed(randomSeed)
os.mkdir(str(randomSeed))
```

This ensures that the PRNG seed for each fuzzing session is stored in case the stream of test cases needs to be repeated. By simply hard-coding the value of randomSeed to be a seed from a previous fuzzing session, the exact same sequence of HTML test cases will be generated by HTMLfuzz from the grammar.

Inside this directory, the last 20 test cases are stored. Since the web browser is not restarted between each test case (this would simply be too slow) it is possible for a crash to be caused by the cumulative effect of a series of test cases. Having the last 20 test cases on disk is therefore useful, and having the PRNG seed also means that we can go even further back in the sequence than this if necessary when investigating a crash.

This handles crash reproduction in terms of the HTML test case. The next part to consider is reproducing what actions the JavaScript took when mutating the DOM for that specific test case.

5.5.2 JavaScript Pseudo-Random Number Generation

The built-in JavaScript PRNG, Math.random(), cannot be seeded by the programmer with a specific value of his own choosing. Therefore, even with the HTML test case being saved to disk, without a way to seed the JavaScript PRNG a test case could crash the browser the first time it is loaded, but the crash could not be reproduced as the next time that same test case is loaded the DOM tree would not get mutated in exactly the same way.

A custom PRNG function implemented in JavaScript was therefore needed that would allow for it to be seeded. To solve this problem, the contents of the small xor4096.min.js script was used which is a JavaScript implementation of Richard Brent's xor4096 algorithm taken from David Bau's seedrandom library [131]. This function can be seeded with a custom value that HTMLfuzz stores inside the JavaScript of each test case. The same seed that is used for the Python PRNG is also used for the JavaScript PRNG.

To summarise, each test case is saved to the filesystem before it is given to the browser, and each test case contains fixed HTML along with a JavaScript containing a fixed PRNG seed. This means that any given test case can be loaded again from disk and should perform the exact same actions each time. There is therefore a good chance of being able to reproduce crashes quite easily without the complexity normally experienced with DOM fuzzing.

5.5.3 Implementing the DOM Fuzzing Strategies

Once a seedable PRNG had been obtained and placed in the JavaScript, each of the DOM fuzzing steps given in Section 4.1.4 were implemented and tested in order.

A helper function was first created to crawl the DOM tree and return a reference to a random element node. This functionality is required by many of the different DOM fuzzing steps and the JavaScript code is shown below.

```
function selectRandomElement() {

        var rootnode=document.childNodes[0];
        var walker=document.createTreeWalker(rootnode, NodeFilter.SHOW_ELEMENT, null, false);
        var elementsArray = [];

        while (walker.nextNode()) {
                if (walker.currentNode.id != "nomutate") {
                        elementsArray.push(walker.currentNode);
                }

        }

        randomElement = elementsArray[Math.floor(prng()*elementsArray.length)];
        return randomElement;
}
```

A TreeWalker logical view of the DOM tree is created that filters out all nodes other than element nodes. This tree is then crawled by repeatedly calling the nextNode() method. At each node, a reference to the node is added to an array. Once the whole tree is crawled, an element is picked at random from this array and returned.

Other helper functions that were created include garbageCollection() to trigger garbage collection, shuffle(array) to randomly shuffle an array, and fuzzAttrValues(node) to mutate all of the attributes of a given element node.

A master function called domFuzz() was created that carries out each of the DOM fuzzing steps from 4.1.4 in order and calls the helper functions as needed in order to achieve this. The body tag of each HTML test case contains **`onload="domFuzz();"`** so that this function is called once the page finishes loading. The final line of the domFuzz() function uses window.location.href to reload the page and cause the process to start again with the next HTML test case.

5.6 Conclusions

This chapter described the implementation of a suite of five tools all for use in web browser fuzzing. The operation of each tool was described as well as the most important aspects of how it was developed. It was explained how the grammar inside HTMLgen and HTMLfuzz allows our tools to get better code coverage of HTML parsing functionality than traditional web browser fuzzing. It was also explained how HTMLfuzz provides better chances of reproducing crashes than traditional DOM fuzzing tools.

The next chapter describes a number of experiments that were carried out in order to evaluate the tools and observe how effective they are for real world fuzzing campaigns against modern web browsers.

Chapter 6: Experiments and Evaluation

The previous chapter documented the implementation of a suite of five new browser fuzzing tools. It is now necessary to conduct a number of experiments in order to evaluate the performance of this suite of tools. This chapter first presents the experiments and how they are to be carried out, then it presents the results.

6.1 Experiment Methods

This section presents step by step instructions for carrying out five experiments aimed at evaluating the suite of fuzzing tools that was implemented in Chapter 5. Conducting these experiments as described below will achieve Objectives 3, 6 and 9 as listed in Chapter 3. It will also provide empirical data to show whether the tools that we developed contribute anything useful to the field of web browser fuzzing.

6.1.1 Experiment One: Evaluate a Harvested Corpus

This experiment aims to determine if a typical corpus created by random web crawling can be expected to contain all existing HTML tags and attributes.

1. Use HTMLscrape to build a corpus of 50 web pages taken from the Internet. This number is arbitrary, as long as the same number is used for experiment two below to make a fair comparison.
2. Use HTMLscan to evaluate this corpus and identify any missing HTML tags and attributes.

6.1.2 Experiment Two: Evaluate a Generated Corpus

This experiment aims to determine whether a corpus generated from our grammar by the HTMLgen tool, that is the same size as the corpus from Experiment One, is better or worse than the harvested corpus in terms of the number of HTML tags and attributes that are present in it.

1. Use HTMLgen to build a corpus of 50 web pages generated from the grammar.
2. Use HTMLscan to evaluate this corpus and identify any missing HTML tags and attributes.
3. Compare the result of this to the result of Experiment One. Draw a conclusion as to the relative quality of the two methods of obtaining a corpus for use in web browser fuzzing.

6.1.3 Experiment Three: Identify Optimum Corpus Size

This experiment aims to determine the ideal size of a corpus generated from our grammar by the HTMLgen tool. When using this tool to generate a corpus to seed a third party fuzzing tool with, it is useful to know how big it needs to be in order to ensure that all HTML tags and attributes are present and maximum code coverage is achieved. A corpus smaller than this would miss out on code coverage and a corpus bigger than this would have unnecessary redundancy which would slow down the fuzzing while not providing coverage gains.

1. Use HTMLgen to generate several different corpora of varying sizes; starting with 10 pages, then 25 pages, followed by increments of 25 up to the largest corpus of 500 pages.
2. Evaluate each corpus with HTMLscan and identify the point at which the corpus is big enough to have used all the HTML tags and attributes in the grammar.

3. Evaluate each corpus with a gcov instrumented web browser using the HTMLharness tool. Identify the point at which increasing the corpus size provides little or no increase in code coverage.
4. Draw a conclusion from this data as to the best size to use when generating a corpus with the HTMLgen tool to be used in a fuzzing session.

6.1.4 Experiment Four: Seed Third Party Fuzzing Tools

This experiment aims to determine whether a corpus generated from our grammar by the HTMLgen tool is suitable for seeding third party mutation fuzzers with.

1. Use HTMLgen to generate a corpus. The size of this corpus should be whatever Experiment Three determined to be optimal.
2. Use the third party Radamsa tool to parse this corpus and generate 50,000 mutated test cases. This forms a second corpus.
3. Use HTMLharness to deliver each of the test cases in the second corpus to a selection of web browsers.
4. Analyse and report on any crashes that result from this experiment.

6.1.5 Experiment Five: Test DOM Fuzzing Strategies

This experiment aims to determine whether the HTMLfuzz tool is effective at triggering bugs in web browsers.

1. Run HTMLfuzz on each of the VMs and fuzz a selection of web browsers. Each browser can be fuzzed for 24 hours.
2. Analyse and report on any crashes that result from this experiment.

6.2 Experiment Results and Evaluation

This section presents and evaluates the results of running each of the experiments.

6.2.1 Experiment One: Evaluate a Harvested Corpus

HTMLscrape was used to build a corpus of 50 web pages. The keywords used were the defaults that can be seen by examining the submitted source code. This corpus was then evaluated using HTMLscan. The results are shown below.

- HTML5 tags missing: 35 out of 108.
- Obsolete tags missing: 43 out of 44.
- Attributes missing: 94 out of 147.

This shows that a corpus of 50 web pages taken randomly from the Internet is missing a large number of HTML tags and attributes. With 78 missing tags and 94 missing attributes a significant portion of HTML parsing and rendering functionality in the target web browser would not be exercised using this corpus.

6.2.2 Experiment Two: Evaluate a Generated Corpus

HTMLgen was used to build a corpus of 50 web pages. This corpus was then evaluated using HTMLscan.

The results are shown below.

- HTML5 tags missing: 1 out of 108.
- Obsolete tags missing: 5 out of 44.
- Attributes missing: 0 out of 147.

This shows that a corpus of 50 web pages generated by HTMLgen is missing a much smaller number of HTML tags and attributes compared to the scraped corpus. This generated corpus would therefore be more likely to trigger bugs when used in a fuzzing campaign as more functionality would be exercised in the target.

6.2.3 Experiment Three: Identify Optimum Corpus Size

A number of different sized corpora were generated using HTMLgen. Each corpus was evaluated by HTMLscan. The results of this are shown in Table 1 below.

Corpus Size	**Missing HTML Tags**	**Missing HTML Attributes**
10	44	31
25	22	9
50	6	0
75	3	0
100	0	0
125	1	0
150	0	0
175	0	0
200	0	0
225	0	0
250	0	0
275	0	0
300	0	0
325	0	0
350	0	0
375	0	0
400	0	0
425	0	0
450	0	0
475	0	0
500	0	0

Table 1: How HTML content varies with corpus size

It can be seen from this data that if the user generates 100 pages or more there is a good probability that every HTML tag and attribute will appear in the corpus at least once.

Next it was necessary to evaluate code coverage for these corpora. In Section 4.3.2 it was stated that the PhantomJS headless browser would be compiled with gcov instrumentation and this would be used

whenever code coverage needed to be recorded. In practice there were problems with this and it was not possible to make PhantomJS compile successfully when gcov was enabled. It was however possible to compile the Netsurf web browser with gcov instrumentation.

The results of using HTMLharness and the gcov instrumented build of the Netsurf web browser to record code coverage data for the corpora are shown in Table 2 below. The first row, for a corpus of zero size, documents the code coverage from simply opening the web browser and then closing it again without processing any input at all.

Corpus Size	**Line Coverage**	**Function Coverage**
0	32.1% (8452 lines)	33.2% (936 functions)
10	37.3% (9803 lines)	38.6% (1089 functions)
25	39.1% (10,299 lines)	40.0% (1129 functions)
50	40.3% (10,591 lines)	41.4% (1168 functions)
75	41.2% (10,850 lines)	42.0% (1186 functions)
100	41.0% (10,789 lines)	41.9% (1183 functions)
125	41.1% (10,802 lines)	42.0% (1184 functions)
150	41.3% (10,857 lines)	42.0% (1186 functions)
175	41.1% (10,803 lines)	41.9% (1183 functions)
200	41.4% (10,881 lines)	42.1% (1188 functions)
225	40.8% (10,735 lines)	41.7% (1176 functions)
250	41.4% (10,893 lines)	42.1% (1188 functions)
275	41.1% (10,800 lines)	41.9% (1183 functions)
300	41.4% (10,899 lines)	42.1% (1188 functions)
325	41.2% (10,846 lines)	42.0% (1185 functions)
350	41.4% (10,902 lines)	42.1% (1188 functions)
375	41.4% (10,897 lines)	42.1% (1188 functions)
400	41.5% (10,907 lines)	42.1% (1188 functions)
425	41.4% (10,894 lines)	42.1% (1188 functions)
450	41.1% (10,802 lines)	41.9% (1183 functions)
475	41.4% (10,893 lines)	42.1% (1188 functions)
500	41.4% (10,888 lines)	42.1% (1188 functions)

Table 2: How code coverage varies with corpus size

It can be seen from this data that if the user generates 200 pages there is a good probability that the resulting code coverage will be about as good as it can be from this tool. Generating a bigger corpus than this appears to be unlikely to give significant coverage gains. The 200 page corpus and the 500 page corpus both cover 1188 functions and the larger corpus only gets an extra 7 lines of coverage.

At this point it is important to address the percentage coverage figures observed above. The reader may be questioning why the total percentage of lines and functions covered never goes much above 40% as this may seem like a low figure, especially given that our goal is maximising code coverage. It is important to remember that we are currently only considering corpora consisting of HTML content. Modern web browsers contain a lot of functionality, and can parse many different input types. As the current work does not make use of the several different formats of video, audio, images and fonts that browsers can process and

also excludes CSS and JavaScript, it is not actually surprising to see a lot of functions remaining unexecuted. The important point is to show that we can maximise code coverage for HTML-only corpora, and that this opens the possibility of applying the same techniques to the other formats of input that web browsers can take.

Both tables show a rapid increase in quality of corpus as its size grows from 10 to 100. From this point, no extra HTML tags or attributes appear in the corpus and the resulting code coverage only goes up slightly. This confirms the fact that using more tags and attributes results in more code coverage, and therefore confirms that assessing a corpus with HTMLscan before using it to fuzz a web browser is useful.

By considering the data in both tables as well as the statements made above, it can be concluded that a HTMLgen corpus of size 200 gives a good trade-off between corpus size, variety of HTML and code coverage. Generating a much bigger corpus than this is likely to be counterproductive as fuzzing will take longer and no code coverage gains will be seen.

6.2.4 Experiment Four: Seed Third Party Fuzzing Tools

The existing corpus from HTMLgen that consists of 200 files was used as the starting point for this experiment since this had been determined as an optimal size. This corpus was used to seed Radamsa to create a second mutated corpus of 50,000 pages.

The HTMLharness tool was used to deliver the final corpus to a selection of web browsers on both Windows and Linux. The crashes that were observed are summarised in Table 3 below.

Fuzzing VM OS	**Web Browser**	**Number of Crashes**
Windows 10	Microsoft Edge	2
Windows 10	Microsoft Internet Explorer 11	1
Windows 10	Mozilla Firefox	1
Debian 8	Netsurf 3.2	1
Debian 8	Xombrero	1

Table 3: Summary of crashes triggered in experiment four

We now look at each crash, grouped by which web browser they occur in.

6.2.4.1 Firefox

The only crash in Experiment Four that appeared to be a security vulnerability was the one affecting Mozilla Firefox. This was observed as an access violation while attempting to execute unallocated memory.

```
BugId AVE:Unallocated 679.f6f @ firefox.exe!xul.dll!NS_LogCOMPtrAddRef summary

BugId:           AVE:Unallocated 679.f6f
Location:        firefox.exe!xul.dll!NS_LogCOMPtrAddRef
Description:     Access violation while executing unallocated memory at 0x3A656764.
Version:         firefox.exe: 50.0.2.6177 (x86)
                 xul.dll: 50.0.2.6177 (x86)
Security impact: Potentially exploitable security issue, if the attacker can control the address or the memory at the address.
Command line:    ['C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe', '--no-remote', '-profile', 'C:\\Users\\user\\AppData
                 \\Local\\Temp\\Firefox-profile', 'http://127.0.0.1:8000/']
```

Figure 2: BugId analysis of Firefox crash

As can be seen from the BugId [136] summary in Figure 2, the test case caused Firefox to place an address in the EIP register that points to memory that is simply not allocated, rather than to valid instructions. If it is possible to alter the test case to place an address of our choosing in EIP then exploitation is almost certainly

possible. Alternatively, even if we cannot put our own value in EIP, by using a technique such as heap spraying we can make it likely that the address pointed to will actually be allocated and will contain a NOP sled followed by our shellcode.

6.2.4.2 Edge

Both of the bugs in Edge were NULL pointer dereferences in edgehtml.dll!GetWebPlatformObject. These are almost certainly not exploitable to get remote code execution and can only be used to close the user's browser as a denial of service attack. The BugId summaries are shown below in Figures 3 and 4.

```
BugId AVR:NULL+0x28 b91.b91 @ microsoftedgecp.exe!edgehtml.dll!GetWebPlatformObject

BugId:           AVR:NULL+0x28 b91.b91
Location:        microsoftedgecp.exe!edgehtml.dll!GetWebPlatformObject
Description:     Access violation while reading memory at 0x28 using a NULL ptr.
Version:         microsoftedgecp.exe: 11.0.14393.82 (x64)
                 edgehtml.dll: 11.0.14393.576 (x64)
Security impact: Denial of Service
```

Figure 3: BugId analysis of Edge crash

```
BugId AVR:NULL+0x658 b91.b08 @ microsoftedgecp.exe!edgehtml.dll!GetWebPlatformObject

BugId:           AVR:NULL+0x658 b91.b08
Location:        microsoftedgecp.exe!edgehtml.dll!GetWebPlatformObject
Description:     Access violation while reading memory at 0x658 using a NULL ptr.
Version:         microsoftedgecp.exe: 11.0.14393.82 (x64)
                 edgehtml.dll: 11.0.14393.576 (x64)
Security impact: Denial of Service
```

Figure 4: BugId analysis of Edge crash

6.2.4.3 Internet Explorer 11

The bug that was triggered in Internet Explorer 11 involved causing functions to call each other recursively until no more return addresses or function arguments could be pushed onto the process's stack. This again can only be used to close the user's browser as a denial of service attack, as shown in Figure 5.

```
BugId RecursiveCall 5b4.f40 @ iexplore.exe!mshtml.dll!Ordinal107 summary

BugId:           RecursiveCall 5b4.f40
Location:        iexplore.exe!mshtml.dll!Ordinal107
Description:     A recursive function call involving 3 functions exhausted available stack memory
Version:         iexplore.exe: 11.0.14393.0 (x86)
                 MSHTML.dll: 11.0.14393.576 (x86)
Security impact: Denial of Service
Command line:    ['C:\\Program Files\\Internet Explorer\\iexplore.exe', 'http://127.0.0.1/']
```

Figure 5: BugId analysis of IE 11 crash

6.2.4.4 Netsurf

The Netsurf web browser was fuzzed on Debian Linux with the gdb debugger attached to it. The output of gdb when the crash occurred is shown below.

```
(20.136923) content/content.c content__reformat 362: 0xff1cc0
netsurf: render/layout.c:3342: layout_minmax_line: Assertion '*line_min <= *line_max' fai
led.

Program received signal SIGABRT, Aborted.
0x00007ffff43e5067 in __GI_raise (sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/rai
se.c:56
56      ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) exploitable
Description: Abort signal
Short description: AbortSignal (20/22)
Hash: 7c589ae4380a0503676b328197c43b6b.dbbf7cbcd421f9621db3b4a9aee22650
Exploitability Classification: UNKNOWN
Explanation: The target is stopped on a SIGABRT. SIGABRTs are often generated by libc and
 compiled check-code to indicate potentially exploitable conditions. Unfortunately this c
ommand does not yet further analyze these crashes.
```

It can be seen that this crash was due to an assertion failure.

6.2.4.5 Xombrero

The Xombrero web browser was fuzzed on Debian Linux with the gdb debugger attached to it. The output of gdb when the crash occurred is shown below.

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff63cfa0a in ?? () from /usr/lib/x86_64-linux-gnu/libwebkitgtk-3.0.so.0
(gdb) exploitable
Description: Access violation near NULL on source operand
Short description: SourceAvNearNull (16/22)
Hash: 02b36002160ccd318c7755dde5bb0a40.22a64551386a0429ae4de28d565f8da2
Exploitability Classification: PROBABLY_NOT_EXPLOITABLE
```

It can be seen that this bug was another NULL pointer dereference as the segmentation fault occurred when the process tried to read from an address that was close to NULL.

6.2.5 Experiment Five: Test DOM Fuzzing Strategies

The HTMLfuzz tool was used to attack a selection of web browsers on both Windows and Linux. The crashes that were observed are summarised in Table 4 below.

Fuzzing VM OS	**Web Browser**	**Number of Crashes**
Windows 10	Microsoft Edge	1
Windows 10	Microsoft Internet Explorer 11	3
Windows 10	Midori	2

Table 4: Summary of crashes triggered in experiment five

We now look at each crash, grouped by which web browser they occur in.

6.2.5.1 Edge

The first crash from Experiment Five that appeared to be a security vulnerability was the one affecting Microsoft Edge.

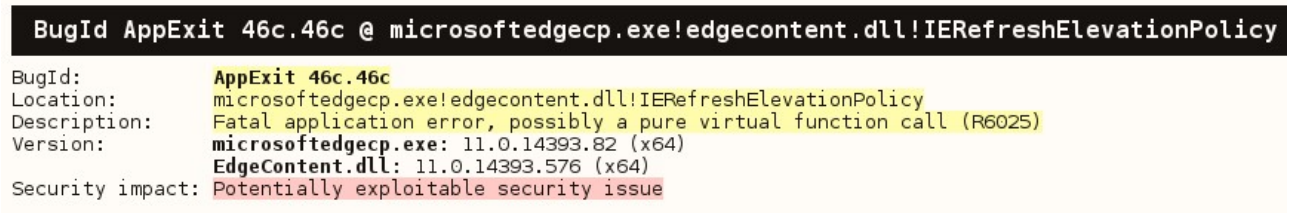

Figure 6: BugId analysis of Edge crash

6.2.5.2 Midori

The other crash that appeared to be a security vulnerability was one of those affecting the Midori web browser. The bug is in the third party Cairo graphics library [138]. The test case causes this to attempt to read from unallocated memory.

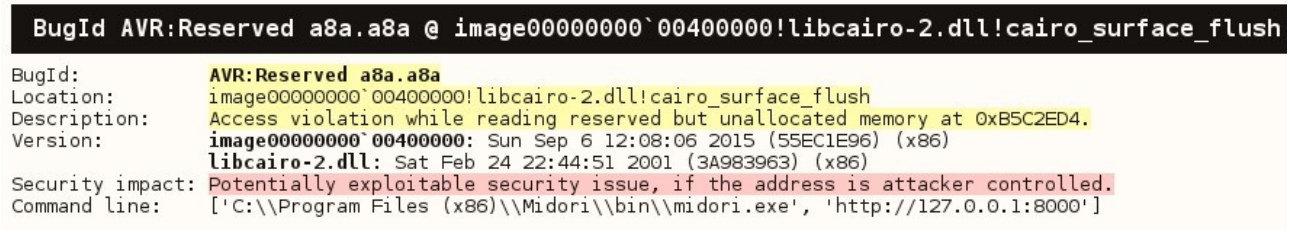

Figure 7: BugId analysis of Midori crash

The second Midori crash is a NULL pointer dereference in libwebkitgtk-3.0-0.dll which is observed as an access violation while attempting to read memory at an address close to NULL.

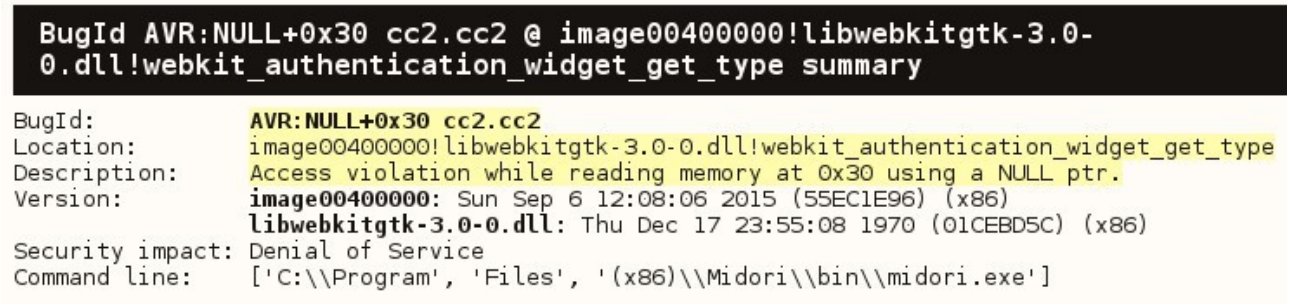

Figure 8: BugId analysis of Midori crash

6.2.5.3 Internet Explorer 11

Three bugs were found in Internet Explorer 11 running on Windows 10.

The first bug that was triggered involved causing functions to call each other recursively until no more return addresses or function arguments could be pushed onto the process's stack. This can only be used to close the user's browser as a denial of service attack, as shown in Figure 9 below.

```
BugId RecursiveCall fef.6d3 @ iexplore.exe!mshtml.dll!CDispContainer::RecalcChildren

BugId:           RecursiveCall fef.6d3
Location:        iexplore.exe!mshtml.dll!CDispContainer::RecalcChildren
Description:     A recursive function call involving 2 functions exhausted available stack memory
Version:         iexplore.exe: 11.0.14393.0 (x86)
                 MSHTML.dll: 11.0.14393.576 (x86)
Security impact: Denial of Service
Command line:    ['C:\\Program Files\\Internet Explorer\\iexplore.exe', 'http://127.0.0.1:8000']
```

Figure 9: BugId analysis of IE 11 crash

The other two bugs that were triggered in IE 11 were both NULL pointer dereferences. They were both observed as access violations while attempting to read memory at addresses close to NULL. The BugId summaries can be seen in Figures 10 and 11 below.

```
BugId AVR:NULL+8 a4f.aa1 @ iexplore.exe!mshtml.dll!Tree::ComputedBlock::Style

BugId:           AVR:NULL+8 a4f.aa1
Location:        iexplore.exe!mshtml.dll!Tree::ComputedBlock::Style
Description:     Access violation while reading memory at 0x8 using a NULL ptr.
Version:         iexplore.exe: 11.0.14393.0 (x86)
                 MSHTML.dll: 11.0.14393.576 (x86)
Security impact: Denial of Service
Command line:    ['C:\\Program Files\\Internet Explorer\\iexplore.exe', 'http://127.0.0.1:8001']
```

Figure 10: BugId analysis of IE 11 crash

```
BugId AVR:NULL+8 5b4.5b4 @ iexplore.exe!mshtml.dll!Ordinal107 summary

BugId:           AVR:NULL+8 5b4.5b4
Location:        iexplore.exe!mshtml.dll!Ordinal107
Description:     Access violation while reading memory at 0x8 using a NULL ptr.
Version:         iexplore.exe: 11.0.14393.0 (x86)
                 MSHTML.dll: 11.0.14393.576 (x86)
Security impact: Denial of Service
Command line:    ['C:\\Program Files\\Internet Explorer\\iexplore.exe', 'http://127.0.0.1/']
```

Figure 11: BugId analysis of IE 11 crash

6.3 Conclusions

It was seen that the HTMLgen tool uses all HTML tags and attributes at least once as the size of the corpus reaches approximately 100 test cases, and the code coverage obtained from the corpus levels out at approximately 200 test cases. This tells us that an ideal size for a corpus when using HTMLgen is about 200 test cases. Less than that is likely to miss out on coverage, while a larger corpus will introduce redundancy that will slow down the fuzzing without adding significant coverage gains.

It was seen that using HTMLgen to seed Radamsa successfully identified some new bugs in web browsers, as did using HTMLfuzz. The majority of bugs being discovered are not exploitable, with many being NULL pointer dereferences, which is quite normal for any kind of fuzzing. In total, 12 new unique bugs were discovered and 3 of these appear to be potentially exploitable. The three potentially exploitable bugs were in the latest versions of Edge, Firefox and Midori, all on Windows 10.

The next chapter evaluates the work of this project with respect to the aims and objectives that we began with, and then draws final conclusions.

Chapter 7: Conclusions

This chapter provides further evaluation of the work carried out by judging its success measured against the objectives that were stated in Chapter 3. Overall project conclusions are also presented in order to summarise the main things that were learned during the work. Finally, several ideas for future work are discussed.

7.1 Evaluation

The first objective was to identify and develop a method of parsing a corpus of web pages and identifying which HTML tags and attributes are not present in it. The key to achieving this successfully was researching a full set of HTML tags including historic usage rather than just relying on the current specification. In addition to the current HTML5 specification that is readily available online, the research identified 44 older HTML tags as documented in Section 4.1.1. Based on this, the simple method was proposed of parsing an input corpus, identifying all used HTML tags and attributes, then reporting a list of those that are in our custom specification but not used in the corpus.

The second objective was to develop the HTMLscan tool to automate the identification of missing HTML tags and attributes in any given corpus. This was efficiently achieved using Python and the BeautifulSoup parsing library, along with the list of HTML tags and attributes that had been researched. This is documented in Section 5.1. The tool efficiently processes an input corpus and reports to the user the list of missing HTML tags and attributes that should be added.

The third objective was to use the HTMLscan tool to assess a corpus that was obtained by web crawling. At this point a minor fault with the proposed method was revealed, as the specific way of building such a corpus had not been considered in detail. This task had been thought to be trivial. Building the corpus manually was quickly ruled out as too time consuming and tedious. Direct use of an existing Linux tool such as wget was then considered. In the end, the development of another small Python tool to solve the problem was decided on. It did not take long to develop the HTMLscrape tool to perform this function as documented in Section 5.2.

Once a corpus had been created using HTMLscrape it was simple to evaluate it with HTMLscan. The experimental results in Section 6.2.1 show that a lot of the HTML tags and attributes were missing as we had expected. The third objective was therefore met.

The fourth objective was to identify and develop a method to generate a corpus of web pages that makes full use of the HTML specification constructed earlier. It was decided in Section 4.1.3 that this would be achieved using a Context-Free Grammar consisting of production rules that use the full range of HTML tags and attributes from our specification. Other options were considered and this choice was justified.

The fifth objective was to develop the HTMLgen tool to automate the generation of HTML corpora that make full use of all existing tags and attributes. The most challenging part of this was the construction of the Context-Free Grammar. The achievement of this is described in Section 5.3.

The sixth objective was to use HTMLscan to evaluate a corpus that had been generated with HTMLgen and compare this to the earlier corpus from HTMLscrape, as well as using an instrumented web browser to measure the code coverage it produces. As detailed in Section 6.2.2, it was found that when comparing two corpora of equal size, the output of HTMLgen is far richer and more complete than the output of HTMLscrape, when evaluated based upon the number of unique HTML tags and attributes appearing in it.

It was deemed useful to determine the optimum size of a HTMLgen created corpus, based upon both the

number of HTML tags present in different sized corpora, and also the code coverage obtained from each. It was found at this point that instrumenting the PhantomJS headless browser using gcov was not possible due to compilation errors that could not be fixed in the available time. A gcov instrumented version of the Netsurf web browser was successfully compiled and used instead. As this involved a full browser complete with GUI, this also led to the development of the HTMLharness tool as documented in Section 5.4 to provide a convenient way for a corpus stored on disk to be delivered to a web browser. Experimental results in Section 6.2.3 found that a corpus of 200 web pages was around the optimal size when creating it with HTMLgen.

An extra experiment was then planned in Section 6.1.4 that had not been explicitly referred to in the objectives. It was deemed useful to use the resulting optimum sized corpus generated by HTMLgen to seed a third party mutation fuzzer and fuzz some real web browsers. The experimental results in Section 6.2.4 documented six new bugs that were discovered in five different web browsers.

The seventh objective was to identify and develop new methods of DOM fuzzing. In Section 4.1.4 a number of methods of DOM tree manipulation were proposed that have the possibility of triggering memory corruption bugs. Most of these were based around the idea of carrying out many operations that require complex pointer arithmetic as well as attempting to induce a state where actions are carried out on elements that no longer exist.

The eighth objective was to develop the HTMLfuzz tool to automate and implement the proposed DOM fuzzing methods. The problem of reproducing crashes was also to be investigated. This was all documented in Section 5.5. As well as implementing the DOM fuzzing strategies, a method of writing deterministic test cases to disk was developed. This involved generating each HTML test case from the grammar and then adding the DOM fuzzing JavaScript containing a seedable PRNG function and a fixed seed. The resulting test case could therefore be written to disk before being read by the web browser. This is an improvement over the traditional method of building the HTML randomly from JavaScript and also using the non-seedable math.random function.

The ninth objective was to carry out a DOM fuzzing campaign against a selection of web browsers and measure how HTMLfuzz performs with regards to causing crashes. Section 6.2.5 documents six new bugs that were discovered in three different web browsers using the HTMLfuzz tool.

It is clear that all nine objectives were successfully achieved through the project.

7.2 Project Conclusions

The design and development of the software suite, and then the experiments that were carried out to evaluate it, revealed a number of useful insights into web browser fuzzing. This section summarises the most important knowledge that was gained.

Firstly, a large number of HTML tags and attributes of varying age were identified that are not present in the HTML5 specification but which nonetheless are still processed by many web browsers and therefore need to be included in a corpus to increase code coverage.

It was confirmed empirically that many HTML tags and attributes (both old and modern) are rarely used and hence do not appear often in typical online web pages. This proved that corpora created solely by web harvesting are likely to miss out a lot of valid HTML. It was then proven empirically that there is a correlation between the number of different HTML tags appearing in a corpus and the code coverage obtained from it. It was seen that code coverage rises with the number of different HTML tags and attributes present in the corpus, and then flattens out shortly after the point where the corpus contains each tag at least once. The conclusion can be drawn from this that in order to obtain the best code coverage when fuzzing a browser, and hence increase the probability of finding vulnerabilities, the HTML corpus must contain all

existing tags and attributes. Since it was proven that this does not occur with harvested corpora, it can also be concluded that generating the HTML corpus from a well designed grammar is a superior method.

It was seen that seeding third party fuzzing tools such as Radamsa with a corpus of 200 pages generated by the HTMLgen tool successfully discovered new bugs in a range of web browsers. Tools such as Radamsa have been used to fuzz web browsers before, as seen in some of the academic literature that was reviewed in Chapter 2, but had not been used in conjunction with a method of maximising the variety of HTML tags and attributes present in the corpus. This approach was seen to be very promising.

The project also revealed that applying DOM fuzzing techniques to test cases containing the maximum variety of HTML tags and attributes was a successful method for revealing new web browser bugs. Previous work on DOM fuzzing has focused on the manipulation of the DOM tree without putting much emphasis on the quality of the HTML test cases this is being applied to. As with seeding third party mutation fuzzers, this work has shown that it is worth using HTML rich test cases for DOM fuzzing too.

Finally, a promising design for DOM fuzzing tools that allows full proof-of-concept test cases to be saved to disk during fuzzing was presented. This provides a contribution to the field as reproducing and investigating crashes is simpler using the developed HTMLfuzz tool than with previous DOM fuzzers in the public domain.

7.3 Future Work

Although the project has overall been quite successful, there are a number of ways that the software tools could be improved in future. As we only considered HTML content, and modern web browsers can process many more types of input than this, there is clearly a need to apply the work to other forms of input. Because most popular browsers use third party libraries to handle images, videos, audio files, and also fonts, it is not a good idea to include those in browser fuzzing. Those third party libraries can be fuzzed directly using custom wrappers, which is much faster and more efficient than fuzzing them through the web browser. However, the HTMLgen and HTMLfuzz tools should both generate test cases that include CSS rather than only HTML. We are clearly missing out on code coverage in the rendering engine without this. It would also be worth generating JavaScript within each test case to test each browser's JavaScript interpreter.

Adding CSS and JavaScript would of course involve further work on the Context-Free Grammar after first identifying all possible CSS and JS tokens, in the same way as this was achieved for HTML. It is highly likely that for both CSS and JS, this technique of first identifying all possible tokens including old, deprecated code, and then using a grammar to ensure that all of them are present in the corpus, will once again outperform random web harvesting.

The suite of fuzzing tools could also be improved by the design and development of a test case minimisation tool. Once a test case has been found to cause a crash, it is useful to repeatedly remove parts of the test case as much as is possible without it ceasing to cause the crash. This could be automated after investigating the most efficient way of carrying it out.

Finally, the HTMLharness tool currently uses a meta refresh tag to cause a new test case to be loaded every second, regardless of if the test case was processed much quicker than this, or if, as is occasionally the case, it had not yet finished being parsed. This makes testing inefficient. Instead of using a meta refresh tag the use of WebSockets should be investigated in order to increase the speed at which test cases are processed. This would allow the browser to signal to the HTMLharness web server when it is ready to receive the next test case.

Appendix 1: References

[1] – Patrice Godefroid et al. “SAGE: Whitebox Fuzzing for Security Testing” in Communications of the ACM, Vol 55, No 3. March 2012.

[2] – Charlie Miller and Zachory Peterson. “Analysis of Mutation and Generation-Based Fuzzing”, 1st March 2007.

[3] – Andrea Lanzi et al. “A Smart Fuzzer for x86 Executables” in 29th International Conference on Software Engineering Workshops, IEEE. 2007.

[4] – Sofia Bekrar et al. “A Taint Based Approach for Smart Fuzzing” in 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.

[5] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007.

[6] - Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”, University of California, San Diego.

[7] – Toby Reynolds. “Bypassing Address Space Layout Randomization”, Null Security, April 2012.

[8] - Tao Guo et al. “GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and Structural Mutation” in 2013 Second International Conference on Informatics & Applications (ICIA 2013), Lodz, Poland, 23 – 25 September 2013.

[9] – Miller, Fredriksen and So. “An Empirical Study of the Reliability of UNIX Tools” in Communications of the ACM, Volume 33 Issue 12, Dec. 1990, pp 32-44.

[10] – Dave Aitel. “Black Hat USA 2002 – An Introduction to SPIKE”. [Online] Available: https://www.youtube.com/watch?v=oGJKdGZUGOk [Accessed: 31st May 2016]

[11] – Microsoft. “Enhanced Mitigation Experience Toolkit (EMET)”. [Online]. Available: https://microsoft.com/emet/ [Accessed: 1st June 2016]

[12] – Shacham et al. "On the Effectiveness of Address-Space Randomization" in Proceedings of the 11th ACM conference on Computer and Communications Security, October 2004.

[13] – FindBugs: Find Bugs in Java Programs. [Online]. Available: http://findbugs.sourceforge.net/ [Accessed: 2nd August 2016]

[14] – VeraCode: Static Analysis Tools and Platforms. [Online]. Available: https://www.veracode.com/products/static-analysis-sast/static-analysis-tool [Accessed: 2nd August 2016]

[15] – Fortify Static Code Analyzer. [Online]. Available: http://www8.hp.com/uk/en/software-solutions/static-code-analysis-sast/ [Accessed: 2nd August 2016]

[16] – Bruce Schneier. “Computer Network Exploitation vs. Computer Network Attack.“ [Online]. Available: https://www.schneier.com/blog/archives/2014/03/computer_networ.html [Accessed: 1st June 2016]

[17] – AddressSanitizer. [Online]. Available: http://clang.llvm.org/docs/AddressSanitizer.html [Accessed: 1st June 2016]

[18] - “CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey,” 2010. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753 [Accessed: 2nd August 2016]

[19] - Will Dietz, Peng Li, John Regehr, and Vikram Adve. “Understanding Integer Overflow in C/C++” in Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, June 2012.

[20] – Common Weakness Enumeration, “CWE-416: Use After Free”. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html [Accessed: 2nd August 2016]

[21] – CVE Details, "CVE-2014-0322: Use-after-free vulnerability in Microsoft Internet Explorer 9 and 10". [Online]. Available: https://www.cvedetails.com/cve/CVE-2014-0322/ [Accessed: 2nd August 2016]

[22] - CVE Details, "CVE-2015-0058: Double free vulnerability in win32k.sys in the kernel-mode drivers in Microsoft Windows". [Online]. Available: https://www.cvedetails.com/cve/CVE-2015-0058/ [Accessed: 2nd August 2016]

[23] - Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007. Page 498.

[24] – NIST National Vulnerability Database, “CVE-2012-5144: Google Chrome off-by-one vulnerability”. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5144 [Accessed: 2nd August 2016]

[25] – Common Weakness Enumeration, “CWE-193: Off-by-one Error”. [Online]. Available: https://cwe.mitre.org/data/definitions/193.html [Accessed: 2nd August 2016]

[26] – Microsoft SDL (Security Development Lifecycle). [Online]. Available: https://www.microsoft.com/en-us/sdl/default.aspx [Accessed: 2nd August 2016]

[27] - Fabian Yamaguchi, Markus Lottmann and Konrad Rieck. “Generalized Vulnerability Extrapolation using Abstract Syntax Trees” in ACSAC ’12 Dec. 3-7, 2012, Orlando, Florida USA.

[28] – Tobias Klein, “A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security”, No Starch Press, 2011.

[29] - Johannes Dahse and Thorsten Holz. “Simulation of Built-in PHP Features for Precise Static Code Analysis” in NDSS ’14, 23-26 February 2014, San Diego, CA, USA.

[30] - John Viega, Tom Mutdosch, Gary McGraw and Edward W. Felten. “Statically Scanning Java Code for Security Vulnerabilities” in IEEE Software, Volume 17 Issue 5, September 2000, pp 68-74.

[31] – Flawfinder. [Online]. Available: http://www.dwheeler.com/flawfinder/ [Accessed: 2nd August 2016]

[32] - Fabian Yamaguchi, Nico Golde, Daniel Arp and Konrad Rieck. “Modeling and Discovering Vulnerabilities with Code Property Graphs” in 2014 IEEE Symposium on Security and Privacy, 18-21 May 2014, San Jose, California. Pp 590-604.

[33] – Cppcheck. [Online]. Available: http://cppcheck.sourceforge.net/ [Accessed: 2nd August 2016]

[34] - V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis” in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, 2005.

[35] – Thomas Ball, Bell Laboratories. “The Concept of Dynamic Analysis” in Proceedings of the 7th European Software Engineering Conference, 1999. Pp 216-234.

[36] – Valgrind. [Online]. Available: http://valgrind.org/ [Accessed: 2nd August 2016]

[37] – PaiMei. [Online]. Available: https://github.com/OpenRCE/paimei [Accessed: 2nd August 2016]

[38] – Art Manion and Michael Orlando, CERT. “Fuzz Testing for Dummies” at Industrial Control Systems

Joint Working Group (ICSJWG), May 2011.

[39] – GDB exploitable plugin. [Online]. Available: https://github.com/jfoote/exploitable [Accessed: 2nd August 2016]

[40] – Matthew Franz. “A Maze of Twisty Passages all Alike: A Bottom-Up Exploration of Open Source Fuzzing Tools and Frameworks” at CERT Vulnerability Discovery Workshop, February 2010.

[41] - Tobias Ospelt. “American Fuzzy Lop, A short Introduction” at Silicon Valley Fuzzers meeting, 9th March 2015.

[42] – Michal Zalewski. “Pulling JPEGs out of thin air”. [Online]. Available: https://lcamtuf.blogspot.co.uk/2014/11/pulling-jpegs-out-of-thin-air.html [Accessed: 2nd August 2016]

[43] – Michal Zalewski. “Technical whitepaper for afl-fuzz”. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt [Accessed: 2nd August 2016]

[44] – Rosario Valotta. “Browser Fuzzing in 2014: David vs Goliath”. [Online]. Available: https://www.syscan360.org/slides/2014_EN_BrowserFuzzing_RosarioValotta.pdf [Accessed: 2nd August 2016]

[45] – Zerodium. “How to Sell Your 0day Exploit to Zerodium” [Online]. Available: https://www.zerodium.com/program.html [Accessed: 2nd August 2016]

[46] – CVE-2010-0249: Operation Aurora Use-after-free vulnerability in Internet Explorer. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249 [Accessed: 2nd August 2016]

[47] – Wade Alcorn, Christian Frichot and Michele Orrù. “The Browser Hacker's Handbook”, Wiley, 2014. Pp 7-8.

[48] – CrossFuzz. [Online]. Available: http://lcamtuf.coredump.cx/cross_fuzz/ [Accessed: 2nd August 2016]

[49] – Wade Alcorn, Christian Frichot and Michele Orrù. “The Browser Hacker's Handbook”, Wiley, 2014. Pp 286-293.

[50] - Saif El-Sherei. “Wadi Fuzzer”. [Online]. Available: https://www.sensepost.com/blog/2015/wadi-fuzzer/ [Accessed: 2nd August 2016]

[51] – Mozilla Security. “DOMFuzz”. [Online]. Available: https://github.com/MozillaSecurity/funfuzz/tree/master/dom [Accessed: 2nd August 2016]

[52] – Mozilla Security. “jsfunfuzz”. [Online]. Available: https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz [Accessed: 2nd August 2016]

[53] – Atte Kettunen, University of OULU. “Test Harness for Web Browser Test Fuzzing”. Master’s Thesis, Degree Programme in Computer Science and Engineering, September 2014.

[54] – jsfuzzer. [Online]. Available: https://code.google.com/archive/p/jsfuzzer/ [Accessed: 2nd August 2016]

[55] – Jesse Ruderman. “CSS grammar fuzzer”. [Online]. Available: https://www.squarefree.com/2009/03/16/css-grammar-fuzzer/ [Accessed: 2nd August 2016]

[56] – Atte Kettunen. “Browser Bug Hunting: Memoirs of a Last Man Standing” at 44CON, 2013.

[57] – Jeremy Brown. “Browser Fuzzing with a Twist (and a Shake)” at Zero Nights, 2015.

[58] - Christian Holler, Kim Herzig and Andreas Zeller. “Fuzzing with Code Fragments”. Presented as part of the 21st USENIX Security Symposium (USENIX Security 2012).

[59] - Rosario Valotta. “Taking Browsers Fuzzing To The Next (DOM) Level” at DeepSec 2012.

[60] – Chen Zhang. “Smashing the Browser: From Vulnerability Discovery to Exploit” at HITCON, Taiwan, August 2014.

[61] – Renata Hodovan. “Fuzz Testing of Web Browsers” at 3rd User Conference on Advanced Automated Testing, Sophia Antipolis, French Riviera, 20-22 October 2015.

[62] – Aleph One. “Smashing the Stack for Fun and Profit”. Phrack, vol. 7, no. 49, 1996.

[63] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007. “Chapter 5: Introduction to Heap Overflows”, pp 89-107.

[64] – Pascal Meunier. “Classes of Vulnerabilities and Attacks”. [Online] Available: http://homes.cerias.purdue.edu/~pmeunier/aboutme/classes_vulnerabilities.pdf [Accessed: 2nd August 2016]

[65] – Microsoft. “A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003“ [Online]. Available: https://support.microsoft.com/en-us/kb/875352 [Accessed: 2nd August 2016]

[66] – Tilo Muller. “ASLR Smack & Laugh Reference: Seminar on Advanced Exploitation Techniques”, February 2008.

[67] – Gadi Evron et al. “Open Source Fuzzing Tools”, Syngress Publishing, 2007. “Chapter 2: Fuzzing – What's That?”, pp 11-26.

[68] - Konstantin Serebryany et al, Google Inc. “AddressSanitizer: A Fast Address Sanity Checker” in Proceedings of the 2012 USENIX Annual Technical Conference.

[69] - Yan Shoshitaishvili et al, UC Santa Barbara. “(State of) The Art of War: Offensive Techniques in Binary Analysis” in 2016 IEEE Symposium on Security and Privacy.

[70] – PhantomJS [Online]. Available: http://phantomjs.org/ [Accessed: 2nd August 2016]

[71] – Charlie Miller. “Fuzzing with Code Coverage by Example” at ToorCon, October 2007.

[72] - Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007. “Automated Source Code Analysis Tools”, page 484.

[73] – Jacob West. “How I Learned to Stop Fuzzing and Find More Bugs” at Defcon 15, August 2007.

[74] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007. “Chapter 16: Fault Injection”, pp 445-459.

[75] – Derek L. Bruening. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation”. PhD Thesis, Massachusetts Institute of Technology, September 2004.

[76] - Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation” in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp 89-100.

[77] – Will Drewry and Tavis Ormandy. “Flayer: Exposing Application Internals” in Proceedings of the first USENIX workshop on Offensive Technologies, 2007.

[78] – MWR Infosecurity. “15 Minute Guide to Fuzzing”. [Online]. Available:

https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/ [Accessed: 2nd August 2016]

[79] – IDA Pro. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml [Accessed: 2nd August 2016]

[80] - Michal Zalewski. "Binary fuzzing strategies: what works, what doesn't". [Online]. Available: https://lcamtuf.blogspot.co.uk/2014/08/binary-fuzzing-strategies-what-works.html [Accessed: 2nd August 2016]

[81] – Andy Davis. “Fuzzing the easy way, using Zulu”, NCC Group, 2014.

[82] - J.C. King. “Symbolic execution and program testing” in Communications of the ACM, 1976, pp. 385-394.

[83] - S. K. Cha, M. Woo, and D. Brumley. “Program-Adaptive Mutational Fuzzing” in Proceedings of IEEE Symposium on Security and Privacy, volume 2015-July, pp 725–741.

[84] – Pedram Amini and Aaron Portnoy. “Sulley: Fuzzing Framework”. [Online]. Available: http://fuzzing.org/wp-content/SulleyManual.pdf [Accessed: 2nd August 2016]

[85] – Sofia Bekrar et al. “Finding Software Vulnerabilities by Smart Fuzzing” in 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.

[86] - “LibFuzzer: A Library for Coverage-Guided Fuzz Testing” [Online]. Available: http://llvm.org/docs/LibFuzzer.html [Accessed: 2nd August 2016]

[87] – Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution” at The Network and Distributed System Security Symposium 2016, San Diego, California.

[88] - Wang, T., Wei, T., Gu, G., and Zou, W. 2011. “Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution” in ACM Trans. Inf. Syst. Secur. 14, 2, Article 15 (September 2011).

[89] – Michal Zalewski. “Mangleme”. [Online]. Available: http://lcamtuf.coredump.cx/soft/mangleme.tgz [Accessed: 2nd August 2016]

[90] – Bugzilla @ Mozilla. “Bug 581539 - (crossfuzz) Bugs found by Michal Zalewski's cross_fuzz”. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=581539 [Accessed: 2nd August 2016]

[91] – H.D. Moore, Rapid7. “Browser fuzzing for fun and profit”. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2006/03/30/browser-fuzzing-for-fun-and-profit [Accessed: 2nd August 2016]

[92] - Michael Sutton, Adam Greene, and Pedram Amini. “Fuzzing: brute force vulnerability discovery”, Pearson Education, 2007, p 275.

[93] – John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Pp 163-197.

[94] – Tavis Ormandy, Google Inc. “Making Software Dumber” at HIRBSecConf 2009.

[95] – Stephen Fewer. “Grinder – System to Automate Fuzzing of Web Browsers”. [Online]. Available: https://github.com/stephenfewer/grinder [Accessed: 2nd August 2016]

[96] - Vijay Ganesh, Tim Leek and Martin Rinard. “Taint-based Directed Whitebox Fuzzing” in Proceedings of the 31st International Conference on Software Engineering, 2009, pp 474-484.

[97] – Yang Chen et al. “Taming Compiler Fuzzers” in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013, pp 197-208.

[98] - Alexandre Rebert et al. “Optimizing Seed Selection for Fuzzing” in Proceedings of the 23rd USENIX conference on Security Symposium, 2014, pp 861-875.

[99] - Richard McNally, Ken Yiu, Duncan Grove and Damien Gerhardy. “Fuzzing: The State of the Art”. Australian Government Department of Defence, Defence Science and Technology Organisation, 2012.

[100] - Pekka Pietikäinen et al, University of Oulu. “Security Testing of Web Browsers”.

[101] - Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman. Trend Micro. "$hell on Earth: From Browser to System Compromise". [Online] Available: http://documents.trendmicro.com/assets/pdf/shell-on-earth.pdf [Accessed: 14th August 2016]

[102] - Mikko Vimpari. “An Evaluation of Free Fuzzing Tools”. Master's Thesis, University of Oulu, May 2015.

[103] - Jared D. DeMott, Richard J. Enbody and William F. Punch. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing" at Defcon 2007.

[104] - Alejandro Hernández. “ELF Parsing Bugs by Example with Melkor Fuzzer”. IOActive, 2014.

[105] - Fabien Duchene. “How I Evolved your Fuzzer: Techniques for Black-Box Evolutionary Fuzzing” at SEC-T Conference, 2014.

[106] - Steven Valdez. “Performing Binary Fuzzing using Concolic Execution”. Master's Thesis, Massachusetts Institute of Technology, June 2015.

[107] – GNU. “Gcov – Using the GNU Compiler Collection”. [Online] Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html [Accessed: 14th August 2016]

[108] – Tom Mitchell. “Machine Learning”, McGraw-Hill, 1997. “Chapter 9: Genetic Algorithms”, pp 249-273.

[109] – W3C. “HTML5”. [Online] Available: https://www.w3.org/TR/html5/ [Accessed: 5th October 2016]

[110] – List of HTML5 Elements and Attributes. [Online] Available: https://www.w3.org/TR/html5/index.html [Accessed: 5th October 2016]

[111] – HTML Grammar. [Online] Available: http://docstore.mik.ua/orelly/web/html/appa_02.html [Accessed: 18th October 2016]

[112] – Google Developers. API Client Libraries. [Online] Available: https://developers.google.com/api-client-library/ [Accessed: 18th October 2016]

[113] - [Online] Available: https://docs.python.org/2/library/sqlite3.html [Accessed: 18th October 2016]

[114] – Tali Garsiel. “How Browsers Work: Browser Error Tolerance” [Online] Available: http://taligarsiel.com/Projects/howbrowserswork1.htm#Browsers_error_tolerance [Accessed: 26th October 2016]

[115] – TrifleJS. [Online] Available: http://triflejs.org/ [Accessed: 26th October 2016]

[116] – SlimerJS. [Online] Available: https://slimerjs.org/ [Accessed: 26th October 2016]

[117] – Firefox Source Code. [Online] Available: https://archive.mozilla.org/pub/firefox/releases/ [Accessed: 26th October 2016]

[118] – Chromium Source Code. [Online] Available: https://www.chromium.org/developers/how-tos/get-the-code [Accessed: 26th October 2016]

[119] – Microsoft. “Introducing Microsoft Edge”. [Online] Available: https://www.microsoft.com/en-gb/windows/microsoft-edge [Accessed: 26th October 2016]

[120] - WebKit. “Open Source Web Browser Engine”. [Online] Available: https://webkit.org/ [Accessed: 26th October 2016]

[121] - Chromium Blog. “Blink: A Rendering Engine for the Chromium Project”. [Online] Available: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html [Accessed: 26th October 2016]

[122] – Griff Ruby. “The Lost Tags of HTML”. [Online] Available: http://www.the-pope.com/lostHTML.htm [Accessed: 26th October 2016]

[123] – CodeHelp.co.uk. “Deprecated HTML tags and alternatives”. [Online] Available: https://www.codehelp.co.uk/html/deprecated.html [Accessed: 26th October 2016]

[124] – Mozilla Developer Network, “HTML Element Reference”. [Online] Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element [Accessed: 26th October 2016]

[125] - Mark Vincent Yason, IBM X-Force Advanced Research. “Understanding the Attack Surface and Attack Resilience of Project Spartan's new EdgeHTML Rendering Engine”.

[126] – BeautifulSoup library. [Online] Available: https://www.crummy.com/software/BeautifulSoup/ [Accessed: 26th October 2016]

[127] – Google python package. [Online] Available: https://pypi.python.org/pypi/google [Accessed: 26th October 2016]

[128] – Eli Bendersky, “Generating random sentences from a context free grammar”. [Online] Available: http://eli.thegreenplace.net/2010/01/28/generating-random-sentences-from-a-context-free-grammar [Accessed: 26th October 2016]

[129] – Oulu University Secure Programming Group, “Radamsa”. [Online] Available: https://github.com/aoh/radamsa [Accessed: 26th October 2016]

[130] – Twisted Matrix Labs, “Twisted”. [Online] Available: https://twistedmatrix.com/ [Accessed: 26th October 2016]

[131] - David Bau, “seedrandom.js: seeded random number generator for Javascript”. [Online] Available: https://github.com/davidbau/seedrandom [Accessed: 26th October 2016]

[132] – Wget python package. [Online] Available: https://pypi.python.org/pypi/wget [Accessed: 26th October 2016]

[133] – Michal Zalewski, “Investigate cross_fuzz crashes”. [Online] Available: https://bugs.webkit.org/show_bug.cgi?id=42959 [Accessed: 26th October 2016]

[134] – John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Page 152.

[135] – John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Pp 222-232.

[136] – Berend-Jan Wever, “BugId: Python module to detect, analyze and id application bugs”. [Online] Available: https://github.com/SkyLined/BugId [Accessed: 26th October 2016]

[137] – Microsoft Developer Network, “Gflags and PageHeap”. [Online] Available:

https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561%28v=vs.85%29.aspx [Accessed: 26th October 2016]

[138] - “Cairo 2D graphics library”. [Online] Available: https://cairographics.org/ [Accessed: 26th October 2016]

Appendix 2: Source Code

Uploaded with this report are the following five Python scripts which make up the fuzzing tool suite that was developed as part of the project.

- HTMLscan.py
- HTMLscrape.py
- HTMLgen.py
- HTMLharness,py
- HTMLfuzz.py