Mikko Häyrynen

Performance analysis of obfuscated JavaScript

Master's Thesis in Information Technology

May 15, 2022

University of Jyväskylä

Faculty of Information Technology

Author: Mikko Häyrynen

Contact information: `mianhayr@student.jyu.fi`

Supervisor: Antti-Jussi Lakanen

Title: Performance analysis of obfuscated JavaScript

Työn nimi: Obfuskoidun JavaScriptin suorituskykyanalyysi

Project: Master's Thesis

Study line: Ohjelmisto- ja tietoliikennetekniikka

Page count: 42+3

Abstract: Obfuscation aims to transform source code into an illegible format while preserving its semantics. This thesis explores the different obfuscation techniques used in the JavaScript programming language. An experiment is conducted to measure the impacts of several different techniques on the performance of three example programs. The findings suggest that code obfuscation can incur severe performance drawbacks, although the extent to which this would hinder real production programs remains unclear.

Keywords: obfuscation, web, javascript, performance

Suomenkielinen tiivistelmä: Obfuskaatiolla pyritään muuntamaan lähdekoodi lukukelvottomaan muotoon siten, että sen semantiikka säilyy muuttumattomana. Tämä tutkielma kartoittaa erilaisia JavaScript-ohjelmointikielessä käytettyjä obfuskaatiomenetelmiä. Käytännön tutkimuksessa menetelmiä vertaillaan keskenään mittaamalla niiden vaikutuksia kolmen esimerkkiohjelman suorituskykyyn. Tulokset osoittavat, että obfuskaatio voi heikentää suorituskykyä merkittävästi, mutta eivät välttämättä yleisty todellisiin tuotanto-ohjelmiin.

Avainsanat: obfuskaatio, web, javascript, suorituskyky

Glossary

Obfuscation	The transformation of source code into an illegible format while preserving its semantics.
Deobfuscation	The reverse operation of obfuscation, aims to recover the original source code from obfuscated code.
Minification	The transformation of source code by removing unnecessary characters to make files as small as possible.
Whitespace	Characters which occupy space in text, but are not visible.
Benchmark	Running a program in order to evaluate its performance.
API	Application programming interface, a connection between computer programs or the constituents of a program.
AST	Abstract syntax tree, a representation of program structure.
DOM	Document Object Model, an interface which represents an XML or HTML document as a tree structure.
LT	Literal transformations, a data obfuscation technique where the value contents of variables are obfuscated.
CFF	Control flow flattening, a structural obfuscation technique where control structures are obfuscated.
DCI	Dead code injection, a structural obfuscation technique where unnecessary code is added to complicate analysis.

List of Figures

Figure 1. Calculator program. 17
Figure 2. Wordle solver program. 18
Figure 3. Turtle graphics program. 20

List of Tables

Table 1. Research questions. 13
Table 2. External dependencies. 16
Table 3. File size measurement results. 21
Table 4. File size multipliers. 21
Table 5. Performance measurements for P1. 22
Table 6. Performance measurements for P2. 23
Table 7. Performance measurements for P3. 23

Contents

1 INTRODUCTION 1

2 OBFUSCATION 3
- 2.1 Defining obfuscation 3
- 2.2 Benign motives 4
- 2.3 Malignant motives 5
- 2.4 Obfuscation versus minification 6
- 2.5 Classifying obfuscation techniques 7
- 2.6 Related research and previously developed tools 9
- 2.7 Deobfuscation techniques and tools 11
- 2.8 Motivations for the study 12

3 PERFORMANCE ANALYSIS EXPERIMENT 13
- 3.1 Research questions 13
- 3.2 Experimental setup 13
- 3.3 Experimental programs 15
  - 3.3.1 Calculator (P1) 17
  - 3.3.2 Wordle solver (P2) 17
  - 3.3.3 Graphical demo (P3) 19

4 PERFORMING ANALYSIS 21
- 4.1 File size measurements 21
- 4.2 Performance measurements 22

5 DISCUSSION 24
- 5.1 File size impacts 24
- 5.2 Performance impacts 25
- 5.3 Threats to validity 27

6 CONCLUSIONS 29
- 6.1 Overview of contributions 29
- 6.2 Future work 29

BIBLIOGRAPHY 31

APPENDICES 38
- A Baseline options for JavaScript Obfuscator 38
- B LT options for JavaScript Obfuscator 39
- C CFF options for JavaScript Obfuscator 40
- D DCI options for JavaScript Obfuscator 40

1 Introduction

JavaScript is the dominant client-side programming language in web development, used by over 97% of all websites (W3Techs 2022). It is currently supported by all major browsers, and can also be used to program server-side applications using runtime systems such as Node.js (OpenJS Foundation 2009). Both client and server-side implementations of the language conform to the ECMAScript specification, which has been published since 1997 and is currently on its thirteenth iteration, called ECMAScript 2022 (Ecma International 2022). The reader is expected to be generally familiar with the JavaScript language.

Obfuscation is defined as transforming source code into an illegible format (Rajba and Mazurczyk 2021), while preserving its semantics (Roeder and Schneider 2010). Obfuscation is widely applied in modern web development to support both benign and malicious goals (Skolka, Staicu, and Pradel 2019). Malicious uses center around hiding attacks in obfuscated code (Xu, Zhang, and Zhu 2012), whereas benign application mainly aims to protect distributable code from theft (Collberg, Thomborson, and Low 1997). Obfuscation techniques have also been shown to have an effect on the performance of programs (Skolka, Staicu, and Pradel 2019), but research focusing on obfuscated JavaScript code performance is lacking. The aim of this thesis is to explore the numerous different obfuscation methods available and compare a subset of them between each other in terms of performance impacts. Because the performance and responsiveness of a website directly affect user experience (Selakovic and Pradel 2016), results on the impacts of source code obfuscation can offer meaningful insight into web design as a whole.

The research experiment is conducted by developing multiple example programs which utilize commonly used third party libraries as dependencies. The code of the example programs is then obfuscated, and benchmarking tools are used to measure the impacts of obfuscation on the performance of the programs when compared to their baseline implementation. Several different obfuscation methods are applied independently and in tandem to measure the impact of each method and their combined effect on different kinds of programs.

While obfuscation methods in web development mainly target JavaScript code, they can also

perform transformations between JavaScript and HTML, or obfuscate CSS definitions. Although these languages are often closely intertwined, this thesis will limit its scope to analyze the effects of pure JavaScript transformations from a performance-centric perspective.

The rest of this thesis is organized as follows. Chapter 2 explores the current body of work surrounding obfuscation techniques in web development and presents motivations for the experimental study. Chapter 3 describes the research method of the experimental study and the programs developed for it. Chapter 4 presents the results of the study. Chapter 5 discusses the results and their implications, as well as threats to validity. Chapter 6 collects the main contributions of the thesis and discusses motivations for future work.

2 Obfuscation

2.1 Defining obfuscation

The term **obfuscation** is generally used to denote methods whose main purpose is to hinder code analysis (Moog et al. 2021), by performing transformations that preserve the functional semantics of the original code (Hammad, Garcia, and Malek 2018). In other words, a program is equivalent with its obfuscated counterpart if they have the same observed behavior Collberg, Thomborson, and Low (1997). Yet another definition would be to say obfuscation means the behavior of a script cannot be fully reasoned about until its execution (Sarker, Jueckstock, and Kapravelos 2020). The reverse operation of obfuscation is called **deobfuscation**, and it covers methods which attempt to recover the original source code from an obfuscated script. Obfuscation is used for both benign and malignant purposes, the most common of which are discussed in the following sections. Several different techniques to achieve obfuscation are also presented.

The theoretical basis of obfuscation has been studied by Collberg, Thomborson, and Low (1997), who propose obfuscation as an effective approach to combat reverse engineering of code. In this context, obfuscation is defined as the transformation of a program into a form where anything one can compute from it could also be computed from the input and output behavior of the program (Barak et al. 2001). By defining a family of unobfuscatable programs, Barak et al. (2001) have shown that obfuscators in this strict sense are impossible to develop. An approach based on relaxed requirements, called *best-possible obfuscation*, has since been proposed by Goldwasser and Rothblum (2007). Using the relaxed approach, obfuscation is shown to be achievable. This thesis is not concerned with formalized obfuscation in the context of information theory, but rather focuses on the practical applications and implications of obfuscation in software development. Theoretical results based on representing programs as Boolean circuits are far distanced from real-life scenarios in modern web development, where obfuscation is widely applied.

2.2 Benign motives

The JavaScript code of a website is distributed to any visitor of the website, which makes it susceptible to intellectual property theft. On major websites, obfuscation is mainly used to combat such theft, and to protect the company's copyrighted code and trade secrets. Although intellectual property is generally protected under copyright law, and theft can be addressed through litigation, developers fear the reverse engineering and plagiarism of proprietary algorithms and data structures (Collberg, Thomborson, and Low 1997). Some companies also obfuscate the source code of their websites to prevent ad-blocking extensions from working properly, in order to increase advertisement revenue (Hieu, Athina, and Zubair 2021). A less creditable, yet arguably still benign motive would be to obscure potential security vulnerabilities in the application from attackers using obfuscation. Paid programs can also be watermarked in a way that uniquely identifies the person the program is sold to, and obfuscation makes the difficult to remove such watermarks (Barak et al. 2001).

Whether the benign uses of obfuscation are applied to enhance secrecy or security, they are essentially an instance of security through obscurity – a notion which has been discussed and rejected by many (Swire 2004). Some also argue that revealing implementation details will increase security and software quality, as it will expose the code to peer review, and hence let the developers know about potential vulnerabilities. Regardless, many companies still choose to obfuscate their code and increasingly advanced obfuscation techniques continue to be developed, which are then met with increasingly advanced detection methods.

The ever-increasing popularity of the JavaScript language has generated a desire for performance optimizations. Such optimizations have been studied in depth by (Selakovic and Pradel 2016), who found that simple changes in the program structure can incur significant changes in performance. Some of the code transformations discussed in their research, such as inefficient reimplementation of standard API functionality or inefficient iteration, can also also used as obfuscation techniques. This implies that certain obfuscation techniques are likely to have some performance drawbacks. Obfuscation can also complicate development processes and cause additional workload for developers, as supplemental steps need to be taken to obfuscate any produced source code. Furthermore, abandoned projects will be extremely hard for future developers to reason about, if the original untransformed source code

is not available. End users will also have difficulties understanding obfuscated code, although this can often be an intended effect, if obfuscation is applied to protect said code.

Obfuscation may also have an unintended effect on the functionality of the code. Skolka, Staicu, and Pradel (2019) tested the correctness of obfuscated libraries using the original test suites of the libraries, and found that less than half of the obfuscated scripts preserved their original semantics. This is mainly due to implementation-level bugs in the obfuscation tools used, as well as oversights of certain corner-cases in the JavaScript language. Issues could be guarded against using automated tools, such as running a test suite for obfuscated code in a deployment pipeline to verify it still retains the original functionality.

2.3 Malignant motives

Common attack vectors in browsers include cross-site scripting (XSS) (Lekies, Stock, and Johns 2013), cross-site request forgery (CSRF) (Jovanovic, Kirda, and Kruegel 2006) and drive-by downloads to install and run malware on the user's machine (Provos et al. 2008). Several vulnerable web APIs, such as `postMessage` (Son and Shmatikov 2013), WebRTC (Reiter and Marsalek 2017) or even the `canvas` element (Mowery and Shacham 2012) can also be utilized for browser based attacks. Attacks based on fingerprinting are often accomplished through vulnerable APIs, but even simple functions can be used, such as `performance.now()` to deliver timing attacks (Skolka, Staicu, and Pradel 2019). All of these examples benefit from obfuscation, as their underlying attack vectors can be obscured by transforming the source code, and this transformation will complicate both manual and automatic analysis when the code is analyzed for malware (Skolka, Staicu, and Pradel 2019). Obfuscation is particularly effective against simplistic detection methods, such as regex-based text analysis on a script (Likarish, Jung, and Jo 2009).

A study on Android applications by Hammad, Garcia, and Malek (2018) found that even trivial obfuscation methods can have a severe impact on the detection accuracy of anti-malware products. They also note that combined transformations do not provide significant benefit over the application of a singular obfuscation method. However, these findings do not necessarily apply to JavaScript obfuscation and should be taken in their own context. Regardless,

obfuscation methods are generally found to decrease the detection rate of antivirus software, and the topic is an area of active research. The techniques used in malicious scripts tend to be more complex than those used in benign ones, and are discussed in more depth in Section 2.5. Furthermore, methods which are generally considered bad programming practice, such as dynamic code generation, are often associated with malicious obfuscation (Skolka, Staicu, and Pradel 2019).

Xu, Zhang, and Zhu (2012) found that 71% of malicious JavaScript files utilize some obfuscation techniques, and even trivial methods, such as string concatenation, can fool more than half of the top antivirus software. More advanced methods, such as encoding obfuscation, avoided detection from all of the 20 tested state-of-the-art antivirus software. They argue that this is due to a static signature-based detection approach used in the antivirus software, which is not effective in analyzing obfuscated code. These findings, however, may be outdated by now and should be interpreted with that caveat in mind.

According to Feinstein and Peck (2007), most security devices have set upper bounds for the time and space complexity of analysis. These can be abused by repeated applications of obfuscation techniques, which will limit the software's ability to find any hidden attack vectors. Conversely, as obfuscation methods are often used to hide malicious code, certain antivirus programs may trigger false positive warnings when a website with obfuscated source code is visited. False positives are particularly prominent when analysis is conducted using simplistic and heuristic detection methods (Murad et al. 2010). Browser manufacturers have also taken action against hidden attacks by disallowing extensions which contain obfuscated code. Google Chrome has instituted such a ban in recent years, supported by their statement that 70% of malicious extensions are obfuscated (Claburn 2018). Mozilla Firefox followed suit soon after with a similar ban (Cimpanu 2019), but neither manufacturer has extended the restriction to minified scripts.

2.4 Obfuscation versus minification

Another means of transformation adjacent to obfuscation is **minification**, which is purely concerned with reducing code size, and by extension its transmission time. The two trans-

formation methods should not be confused, since minification is nearly always benign and used in pursuit of performance improvements, unlike obfuscation. This is supported by the research of Skolka, Staicu, and Pradel 2019, who showed that minified code generally improves the performance of a website, whereas obfuscation reduces it. However, a study by Król and Zdonek (2020) has shown that the performance improvements of minification are not necessarily significant, and additional techniques, such as image compression should also be applied. Minification can even be seen as a useful, simplified, systematic and automated form of code golf, a recreational activity in which the aim is to compress source code into as few characters as possible, while preserving its original semantics.

The techniques used in minification are fairly simple, and usually include the omission of comments, redundant whitespace and indentation in the script. These features were originally implemented by the JSMin tool (Crockford 2001) and complemented further with the abbreviation of local variable names by the YUI compressor (Yahoo! 2007). More advanced methods can perform logical optimization on the code, such as omitting redundant variable declarations or unreachable code. Minified JavaScript files are often combined with a source map (Lenz and Fitzgerald 2011) to associate their definitions with the original untransformed source code, which is essentially a form of deobfuscation. A study by Sakamoto et al. (2015) found that 87% of Alexa Top 500 websites could gain further benefit from minification, and the total file sizes of those sites could be reduced by 39% using the authors' techniques. According to Skolka, Staicu, and Pradel (2019), minification affects more than a third of all scripts on the web, and is commonly included as a step in deployment pipelines.

2.5 Classifying obfuscation techniques

Obfuscation techniques can be categorized based on the transformations they perform on the code. Numerous techniques have been identified by previous research, and more advanced methods continue to be developed to combat increasingly efficient detection systems.

This section presents the most prominent JavaScript obsfuscation methods in current use, and builds on the work of Feinstein and Peck (2007), Jscrambler (2017), Xu, Zhang, and Zhu (2012), Skolka, Staicu, and Pradel (2019) and Moog et al. (2021).

- **Randomization** operates by inserting random segments into a script, such as redundant comments or whitespace. It also encompasses identifier obfuscation, which is the garbling of variable or function names to make them illegible.
- **Data obfuscation** refers to transformations on literal values, such as strings or numbers. Methods include string splitting and concatenation, string replacements based on regular expressions, relocating literal values into a global array, changes to string encoding, and using arithmetic operators or bit shifting to obscure numerical values.
- **Structural modification** can insert dead code in the form of redundant conditional branches or loops, which have no effect on the functionality of the code. A method called control flow flattening can also be applied to transform nested control structures into `switch` statements, which are placed inside a single infinite loop to make the program flow significantly harder to follow, while preserving its semantics.
- **Definition overwriting** can replace existing properties in the global object, such as function definitions or global variables. This makes it extremely challenging to determine if any definition conforms to the assumed JavaScript language specification, as references can be swapped around or altered to point elsewhere.
- **Dynamic code generation** creates new JavaScript segments during runtime, using global functions such as `eval()`, the function constructor `Function()`, or inserting new script tags into the HTML using `document.createElement('script')`. Scripts can also be partitioned to be defined over multiple locations in the code, in order to make them harder to piece together.
- **Encoding or encryption** can transform a code block into a string which is no longer valid JavaScript. The original code can then be recovered from the transformed string via decoding or decryption, and evaluated using dynamic code generation methods.

Obfuscation tools often combine several of the methods above in order to maximize obscurity. Some tools, such as the JavaScript Obfuscator by Serafim (2016), also support extensive customization of the transformation options, allowing the user to pick how heavily obfuscated they want the end product to be. They may also offer features that hinder the use of inspection tools, such as overwriting methods of the `console` object made for logging purposes, or limiting use of the `debugger` statement. Some malicious scripts have also been found to have applied obfuscation methods recursively, which each iteration adding another

layer of obfuscation using the same method (AbdelKhalek and Shosha 2017).

2.6 Related research and previously developed tools

The detection, analysis and deobfuscation of transformed code has been studied extensively in the last decade. The solutions developed for obfuscation detection utilize static source code analysis, as well as dynamic approaches, which analyze code generation and execution during runtime. Modern tools also include machine learning algorithms for scalable analysis (Fang et al. 2022).

Xu, Zhang, and Zhu (2012) have performed a comparison of obfuscation methods to reveal malicious code by categorizing and scanning them using antivirus software. Encoding-based approaches were found to be more effective for malicious purposes than data obfuscation or randomization. The authors remark on the benefits and drawbacks of both static and dynamic approaches for obfuscation detection, observing that static analysis suffers from poor detection rate, while dynamic methods can incur a performance penalty. They note that a hybrid approach could be used to improve accuracy and performance. Xu, Zhang, and Zhu (2013) have also developed a mostly static detection solution based on function call invocation, called JSTill. According to their test results, this solution incurs negligible performance overhead, but produces good detection accuracy and low numbers of false positives.

Kim, Im, and Jung (2011) have developed their own detection system and second the previous notion about the effectiveness of hybrid approaches. Their hybrid platform produced better results than previously developed alternatives in terms of both detection rate and analysis time. Al-Taharwa et al. (2014) propose an alternative, completely static solution that focuses specifically on readably obfuscated scripts and has demonstrably superior performance. Their solution is agnostic to the malignancy of the script, and works by constructing an AST of the program code. Sarker, Jueckstock, and Kapravelos (2020) have compared static and dynamic approaches and combined them into a hybrid analysis platform to detect obfuscation based on browser API usage patterns. Their approach is based on the observation that if the results of a script's static and dynamic analysis differ in terms of browser API usage, the script contains obfuscated behavior. They also note that even though the `eval()`

function is infamous for dynamic code generation (Richards et al. 2011), it is no longer a prevalent obfuscation method, and has given way to more novel approaches.

A massive study conducted by Skolka, Staicu, and Pradel (2019) analyzed nearly a million scripts from Alexa Top 100 000 websites using a neural network based classifier and found transformations affecting 38% of all scripts. However, a vast majority of these transformations were mere minification, with more advanced obfuscation methods present in less than 1% of the scripts. It was also found that nearly 90% of all obfuscated scripts were the result of a single obfuscation tool, called DaftLogic Obfuscator (Daft Logic 2008). Third party scripts were nearly twice as likely to be transformed than scripts from the visited website itself, and obfuscation was also found more prevalent on certain types of websites, such as those serving adult content. For identification, the authors created a novel approach using enriched AST representations, which incorporate information about whitespace and variable name length into the standard JavaScript AST.

A recent study by Moog et al. (2021) propose yet another AST-based static solution extended with control and data flow analysis, which also detects other code transformations such as minification. The approach is based on the premise that the AST of a transformed script will have a different structure than the original script. Their research found that 90% of Alexa Top 10k websites contain script transformations, which indicates that transformations do not imply maliciousness. It was also found that the more popular a website is, the more likely it is to use advanced transformation methods in addition to simple ones.

Recently, machine learning methods have also been applied to improve obfuscation detection accuracy. Research by Ndichu, Kim, and Ozawa (2020) performs deobfuscation, unpacking and decoding (DUD) in order to train a classifier model, which determines whether or not a script is malign or benign. The model training works by applying a natural language processing (NLP) algorithm using the vector representations of words found in a script. Such vector representations have previously been used by the authors to detect maliciousness in a script, but the models perform suboptimally when analyzing obfuscated scripts (Ndichu et al. 2018; Ndichu et al. 2019). Using a sample of over 200 000 scripts preprocessed using the DUD technique, the model was trained and tested, and found to show improvements in both accuracy and performance when compared to previous approaches.

2.7 Deobfuscation techniques and tools

Deobfuscation is the reverse operation of obfuscation, and its aim is to recover the original source code by performing reversed transformations. Automated deobfuscation systems have also been developed to aid in the prevention of malicious code. Early solutions include The Ultimate Deobfuscator by Chenette (2008) and jsunpack by Harstein (2009), which implement simple deobfuscation techniques. Caffeine Monkey by Feinstein and Peck (2007) is a detection framework, which builds on top of the SpiderMonkey JavaScript implementation and includes a deobfuscator as a part of the system. However, the approaches used in these tools mostly center on preventing the execution of malicious scripts, instead of restoring the original semantics of the code. A commonly applied prevention method is to replace invocations of potentially malicious functions with `alert()`, which can be used to halt execution and simultaneously output the code to the user. These tools are also dated, and may not account for many recent developments in the field of obfuscation.

Research on an automated deobfuscation system called JSDES has been conducted by AbdelKhalek and Shosha (2017). Their system includes a runtime analysis component, which will capture dynamically generated JavaScript code and sequence it to restore the original code. This is done by identifying all the different methods used in dynamic code generation, and necessitates the execution of said code in a protected environment. The code execution is then tracked by the system to reconstruct the original code in correct order. Although the system exhibits a high success rate for deobfuscation, its main goal is to reveal maliciousness in a script, rather than fully restore the human-readable semantics of the original script.

Numerous modern and actively maintained web-based solutions exist to aid in making JavaScript code more easily legible, such as Dirty Markup (10 Best Design 2009) and Online JavaScript Beautifier (Lielmanis and Newman 2013). However, such tools perform mostly simple transformations such as whitespace formatting, and do not even claim to achieve extensive deobfuscation. More thorough approaches exist in the form of JS NICE (Langdon 2014) and JavaScript Deobfuscator (ben-sb 2021), which are able to revert some of the more advanced techniques, such as the use of proxy functions, global string arrays and encoded identifiers. Still, it is trivial to see that certain obfuscation methods can never be fully reverted, and features such as meaningful variable names can be irreversibly wiped out during obfuscation.

2.8 Motivations for the study

JavaScript performance has become an increasingly interesting subject as the popularity of the language has increased, especially since it can be now used to develop server-side applications which might serve thousands of requests per second. Performance on the client-side is equally important, as unresponsive websites can cause users to prefer a solution from a competitor instead. These observations are noted by Selakovic and Pradel (2016), who have studied the performance pitfalls of the language in depth. They have found that JavaScript offers several APIs to perform a single task, and these are often used in suboptimal ways by developers. This observation is significant, since obfuscation methods can perform transformations between different APIs. However, the authors also note that most of the optimization methods do not provide consistently improved performance across multiple JavaScript engines. Their findings support the previously studied notion that most optimizations can be achieved with relatively simple changes, which conversely supports the presentiment that simple obfuscation methods may have significant performance impacts.

The performance of both obfuscated and minified code has been studied previously by Skolka, Staicu, and Pradel (2019). Their research on several minification and obfuscation tools showed that runtime performance can be slightly improved with minification, but obfuscation generally increases execution time by 16% to 37%. However, Selakovic and Pradel (2016) have noted that the performance impact of different optimization methods can vary significantly between browsers, depending on the underlying JavaScript engine used. They note that less than half of the tested optimization methods consistently improve performance in different versions of V8 and SpiderMonkey, which are the JavaScript engines of Google Chrome and Mozilla Firefox, respectively.

It is therefore clear that obfuscation is likely to incur losses in performance, but the extent to which each individual obfuscation method contributes to this is unknown. This thesis will make its contribution here, by measuring, analyzing and comparing the effects of several distinct obfuscation techniques. Furthermore, as most of the related research focuses on malicious code detection and attack prevention, the study conducted in this thesis will concentrate on the more benign aspects of obfuscation and its unintended side effects.

3 Performance analysis experiment

3.1 Research questions

The study aims to answer the following research questions:

RQ1	**What are the measurable impacts of different obfuscation techniques in terms of performance?**
RQ1.1	How does code obfuscation affect file size?
RQ1.2	How does code obfuscation affect execution time?

Table 1: Research questions.

3.2 Experimental setup

As noted by Selakovic and Pradel (2016), it is challenging to reliably measure JavaScript performance, mainly due to JIT compilation, garbage collection, and differences between operating systems and browsers. This experiment will therefore follow an approach similar to theirs, where the code examples are executed repeatedly, this time using an automated JavaScript benchmarking library, in order to minimize the effect of these challenging factors. The code will be executed in two popular JavaScript engines, V8 and SpiderMonkey, in order to observe their potential differences.

The data set selection proved to be challenging. An ideal setup would consist of a large body of real world scripts, scraped from some of the most widely used websites or open source repositories. Although these kinds of data sets already exist, such as the 150k JavaScript Dataset provided by the research of Raychev et al. (2016), this approach poses several problems. A large portion of scripts gathered using automated methods are likely to have already been transformed, as shown by Skolka, Staicu, and Pradel (2019). In order to alleviate the effects of previously transformed scripts on the results of this study, such scripts would have to be identified and omitted, likely using a machine learning based approach, which is beyond

the scope of this study. Additionally, most real world scripts include dynamically invoked portions, such as event listeners. Such code makes it challenging to judge whether or not the execution of an automatically invoked script corresponds to its real world usage. If not, the analysis of such a script would not provide meaningful results. Furthermore, external dependencies and the platform-agnostic nature JavaScript makes the correct execution of arbitrary scripts in a controlled environment near impossible. The data set selection is therefore conducted manually, and will comprise of several original example programs.

It should be noted that some additional preliminary experiments were also conducted by obfuscating some commonly used open source libraries with extensive test coverage, and executing the code via the test suites of the libraries. This approach was inspired by the previous work of Skolka, Staicu, and Pradel (2019). It was quickly apparent that the testing frameworks themselves added enough overheard for the performance effects of obfuscation to be completely insignificant, and thus these experiments were not found to produce meaningful results, nor are they discussed further.

The obfuscation techniques chosen for the analysis are the following:

- **Literal transformations (LT)**: multiple data obfuscation methods are applied on literal values in the code. String literals are split into small portions and moved into a separate array. Calls to this array are obfuscated, and the values within are shuffled and encoded using the RC4 stream cipher. An arbitrary static seed is provided for the random number generator used in the obfuscation process to ensure consistent reproduction of results. The indices of the global string array are also obfuscated, shifted and rotated. Numerical literals are also transformed into computable expressions. The settings applied in this setup are presented in Section B.
- **Control flow flattening (CFF)**: structural transformation is applied to flatten the conditional branching into a singular centralized control structure. The technique is applied at the highest available threshold, which will apply flattening to every node of the parsed program. The settings applied in this setup are presented in Section C.
- **Dead code injection (DCI)**: sections of random unnecessary code are inserted into the program as a structural transformation. The technique is applied at the highest available threshold, which will insert a block of dead code within every node of the

parsed program. The settings applied in this setup are presented in Section D.

These techniques were selected for their ability to affect the program structure significantly. Trivial obfuscation techniques have no effect on program execution (Hammad, Garcia, and Malek 2018), and as such are of no interest for this study. For the sake of brevity, in the following sections the selected obfuscation techniques are referred to by their abbreviated identifiers. Each technique will be applied separately and the resulting performance be compared to that of the baseline program. Finally, the techniques will all be applied in tandem to measure their combined impact.

The entire experiment will be conducted in a JavaScript environment, utilizing open source libraries. Obfuscation techniques will be applied using the widely used JavaScript Obfuscator library by Kachalov (2016), which has an online version available, developed by Serafim (2016). The performance of the obfuscated scripts will be measured using the Benchmark.js library (Vukušić 2011), which also has an online version available (Vukušić 2017). These tools were selected based on their popularity and the availability of a wide array of configurable options, which will aid in isolating the effects of different obfuscation methods. It should be noted that the obfuscation techniques will not the applied to the obfuscation or benchmarking libraries themselves, or to the third party libraries which the tools depend on.

The obfuscation library used in the experiment forcibly applies some obfuscation techniques, such as the abbreviation of variable names and the removal of code comments. The issue is alleviated by selecting a baseline set of options presented in Section A, which minimize the impact of any techniques not selected for the study. Furthermore, performance is measured after applying the baseline set of options to show that the forcibly applied obfuscation techniques generally have a negligible impact on performance. Common minification techniques, such as compacting the source code by omitting whitespace, are not applied. The settings for the studied obfuscation techniques replace any conflicting baseline settings when applied.

3.3 Experimental programs

Three original browser-based JavaScript programs are constructed to carry out the performance analysis. Each programs utilizes a major third party library as a dependency to better

mimic the implementation of real world web applications. The external dependencies and their versions used are listed in Table 2, and each library is also obfuscated along with the proprietary program code prior to running the benchmark. The external libraries are sourced without any transformations, as the commonly used minified versions would be counterproductively more heavily affected by the application of the baseline obfuscation settings.

Program	Dependency	Version
P1	Vue.js	3.2.33
P2	jQuery	3.6.0
P3	Babylon.js	5.4.0

Table 2: External dependencies used in the example programs.

The programs are designed to have a higher focus on pure JavaScript execution than most contemporary real world web applications, in order to better isolate the effects of JavaScript code obfuscation. Furthermore, in order to facilitate the performance measurements and limit the effect of outside factors, the programs are designed to execute fully autonomously from start to finish, without the need for any intermediate user inputs. All of the programs also work entirely within the web browser, without the need for any backend services or other such external environments. This is a conscious choice, as the execution of an asynchronous HTTP request or any similar network-based query is orders of magnitude slower than locally executable JavaScript code, which would obscure the performance impacts of the applied obfuscation techniques.

When executed using the benchmarking library, the programs will be altered to contain intrinsic iteration, i.e. the programs will perform their work multiple times. This is done to limit the effect of environmental interference on the results, as well as to scale the execution times to a more readily perceivable interval. A single execution is thus expected to take approximately one to ten seconds, and a single benchmark one to ten minutes.

3.3.1 Calculator (P1)

The first program is a simple mock-up of a pocket calculator, which can perform basic arithmetic operations (addition, subtraction, multiplication and division) on numerical literals. The program's template is rendered dynamically using the Vue.js framework (You 2014). Native DOM API methods are used to simulate usage of the calculator by selecting elements and invoking their event listeners. Each iteration of the benchmark performs a hundred calculations, each operating on a thousand random numbers and operators. An image of the calculator can be seen in Figure 1.

The main objective of this program is to mimic the operation of the front end of a common web application, where event listeners are invoked and computations are performed as a user clicks elements on the interface.

Figure 1: Calculator program.

3.3.2 Wordle solver (P2)

The second program implements a solver for the popular web-based guessing game Wordle, originally developed by Josh Wardle and released in 2021 (New York Times 2022c). The game has since been acquired by The New York Times (New York Times 2022a, 2022b). The objective of the game is to guess a random five-letter word using a maximum of six

guesses. Each guess will reveal information about the correct answer by coloring letters in the guessed word; green letters are in the correct position, yellow letters appear in the answer in a different position, and grey letters do not appear in the answer. Wordle's popularity has grown rapidly since its release, and academic authors have also taken an interest in the game, as studies on optimal strategies have recently started appearing (Anderson and Meyer 2022).

The program implemented in this study utilizes the same word lists for allowed guesses and potential answers as the actual Wordle game. Each iteration of the program picks a random word as an answer, after which a maximum of six random guesses are made while eliminating all inapplicable words from future guesses based on the game mechanics. Without applying any statistics on the incidence of letters, this simple approach produces a win rate of approximately 67%. An image of the program completing a game can be seen in Figure 2.

The main objective of this program is to perform extensive operations on a large assortment of strings, and it is hypothesized to be heavily affected by the obfuscation of literal values. It also performs extensive DOM manipulation, which is common in dynamic web applications. When run in the benchmark, each iteration of the program will guess a thousand words.

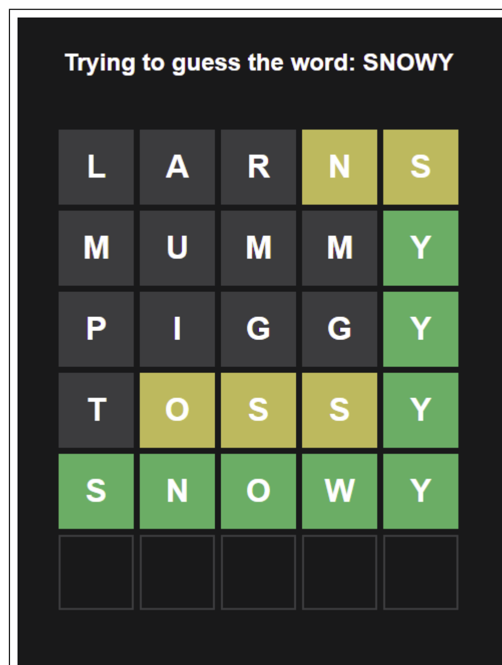

Figure 2: Wordle solver program.

3.3.3 Graphical demo (P3)

The third and final program composes a graphical demo in a three dimensional space. The demo is drawn using the notion of turtle graphics, in which a cursor is moved on a plane using simple `walk` and `turn` commands. As the cursor moves, it draws a line behind itself. Turtle graphics were notably implemented into the Logo programming language by Seymour Papert in the 1960s, and have been used to study teaching programming to children and students (Solomon and Papert 1976; Caspersen and Christensen 2000). Since then, the idea has also been implemented in many modern programming languages, such as Python (Python Software Foundation 2022). This demo program includes an original implementation of turtle graphics in a three dimensional space, using the theoretical foundations presented by Verhoeff (2010). Rendering in the browser is implemented using the open source WebGL-based library Babylon.js (Catuhe 2013).

The program draws a three dimensional interpretation of a space filling curve, first described by Hilbert (1891). As an instance of a self-similar fractal, the Hilbert curve can be constructed in two dimensions using an L-system based on rewrite rules as described by Lindenmayer (1968). Extension into the third dimension is implemented using a simplified version of the L-system proposed by Prusinkiewicz and Lindenmayer (1990). The motion of the turtle is represented in real time as the cursor draws, and some superfluous visual effects are added to entertain the author. An image of the curve in progress can be seen in Figure 3.

This program is designed to be computationally expensive, as it performs extensive computer graphics calculations to achieve real-time rendering in a three dimensional space.

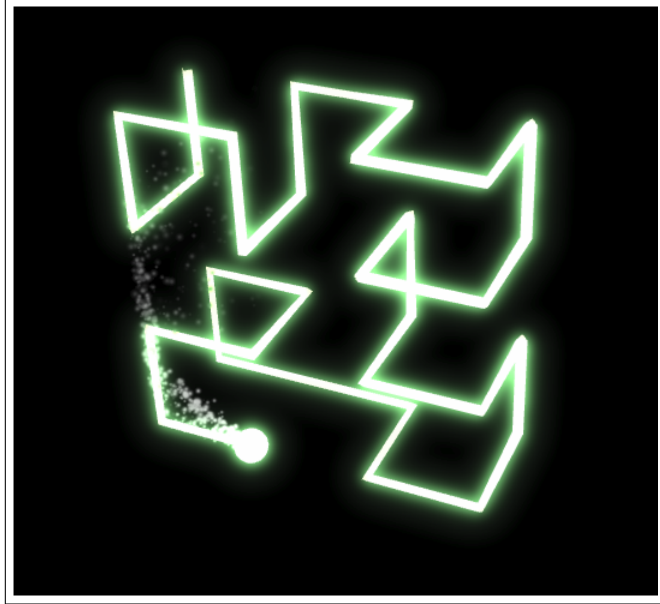

Figure 3: Turtle graphics program.

4 Performing analysis

4.1 File size measurements

Each of the example programs and dependency libraries were obfuscated using all of the selected settings. The resulting raw file sizes for each of the example programs and their corresponding dependency libraries are presented in Table 3. For the sake of clarity, file sizes are also presented as multipliers when compared to the size of the unaltered files in Table 4. Since all of the obfuscation setups used in this study are completely deterministic, these results are reproducible using the same settings.

	P1	Vue.js	P2	jQuery	P3	Babylon.js
Unaltered	1.387 KB	636.0 KB	119.5 KB	288.6 KB	7.231 KB	13.19 MB
Base	1.769 KB	570.5 KB	172.1 KB	297.3 KB	9.204 KB	12.75 MB
LT	15.43 KB	1059 KB	643.0 KB	559.1 KB	33.19 KB	38.22 MB
CFF	3.579 KB	997.3 KB	175.2 KB	592.5 KB	18.86 KB	25.59 MB
DCI	3.297 KB	1601 KB	416.2 KB	861.1 KB	26.59 KB	30.33 MB
All	21.30 KB	4899 KB	664.0 KB	2626 KB	104.7 KB	119.5 MB

Table 3: File size measurement results for all programs and dependencies.

	P1	Vue.js	P2	jQuery	P3	Babylon.js
Unaltered	–	–	–	–	–	–
Base	1.28	0.90	1.44	1.03	1.27	0.97
LT	11.1	1.66	5.38	1.94	4.59	2.90
CFF	2.58	1.57	1.47	2.05	2.61	1.94
DCI	2.38	2.52	3.48	2.98	3.68	2.30
All	15.4	7.70	5.56	9.10	14.5	9.06

Table 4: File size multipliers as compared to the size of the unaltered files.

4.2 Performance measurements

The code was run in Google Chrome version 101.0.4951 and Mozilla Firefox version 100.0, both of which are the latest available at the time of writing. The underlying hardware consisted of an AMD Ryzen 1700 processor running at 3.85 GHz, 16 GB of RAM and an NVIDIA GeForce GTX 1070 GPU. Hardware acceleration was enabled in both browsers. Each benchmark was executed a minimum of 50 times to ensure statistically significant results. The benchmarking library would occasionally perform some additional iterations, likely due to the timeout limit not having passed since the last iteration. These were included as part of the sample. Performance measurements for all transformed program variants are presented in Table 5, Table 6 and Table 7.

All of the values are presented as reported by the Benckmark.js library, with the exception of measurements for the LT, CFF, DCI and All transformations for P3 in the Google Chrome browser. These transformations were found to produce significantly higher execution times during the initial few iterations of the benchmark, which were removed from the sample as outliers. It is nevertheless worth a mention that such a browser-specific initiation overhead was observed, and it was consistently produced multiple times.

	P1 in Google Chrome			**P1 in Mozilla Firefox**		
	n	Mean (sec)	SD	*n*	Mean (sec)	SD
Unaltered	52	1.689	0.041	53	1.638	0.020
Base	52	1.980	0.034	52	1.652	0.025
LT	52	2.183	0.054	53	1.652	0.016
CFF	52	2.216	0.034	51	1.727	0.013
DCI	52	2.186	0.079	52	1.806	0.028
All	51	3.199	0.032	52	2.078	0.023

Table 5: Performance measurements for P1. Reported are the number of iterations run (n), average execution time per iteration and standard deviation.

	P2 in Google Chrome			P2 in Mozilla Firefox		
	n	Mean (sec)	SD	n	Mean (sec)	SD
Unaltered	51	4.146	0.030	51	4.075	0.040
Base	51	4.216	0.033	51	4.098	0.033
LT	51	4.637	0.032	51	4.013	0.039
CFF	51	4.431	0.032	51	4.205	0.049
DCI	51	4.426	0.037	51	4.670	0.053
All	50	5.549	0.045	51	4.151	0.047

Table 6: Performance measurements for P2. Reported are the number of iterations run (n), average execution time per iteration and standard deviation.

	P3 in Google Chrome			P3 in Mozilla Firefox		
	n	Mean (sec)	SD	n	Mean (sec)	SD
Unaltered	51	2.888	0.020	50	8.885	0.015
Base	51	2.875	0.015	50	8.875	0.060
LT	49	3.168	0.014	50	8.885	0.015
CFF	47	4.146	0.059	51	9.327	0.244
DCI	49	2.892	0.016	51	8.888	0.042
All	45	9.892	0.078	51	9.294	0.558

Table 7: Performance measurements for P3. Reported are the number of iterations run (n), average execution time per iteration and standard deviation.

5 Discussion

5.1 File size impacts

File sizes were found to drastically increase when any obfuscation method was applied. The only exception to this was applying the baseline settings to the Vue.js and Babylon.js dependency libraries, which slightly reduced the size of the libraries. This likely stems from the removal of comments and changes to whitespace, as non-minified versions of all dependencies were used. Since the minified versions are significantly smaller yet semantically identical to the originals, obfuscating them with the baseline settings would likely produce the same results and thus lead to even larger increases in file size.

Literal transformations affected the file sizes the most. This is expected, since the LT settings contain many transformations which heavily increase the character count in the code. This is in line with the documentation of the obfuscation library (Kachalov 2016), which explicitly states that heavy application of some of the string obfuscation techniques can produce very large file sizes. This is notably caused by escape sequence encoding and string splitting at a high intensity. Another interesting observation is that literal transformations produce larger relative increases in smaller input files than in larger ones. This implies there is some initial overhead brought about by the application of literal transformations, the impact of which dissipates as file sizes increase.

The CFF and DCI settings also increased file sizes significantly, generally by a factor of two or three. For DCI, these findings are in line with the documentation (Kachalov 2016), which warns users of the fact that large increases in file size should be expected. For CFF however, no such caveat is issued in the documentation. Since the techniques were applied at the same intensity for all programs, the variations in the size increase between files are likely explained by differences in program structure, which when parsed results in a different number of nodes to be altered by the techniques. Another interesting observation is that applying all of the obfuscation techniques in tandem sometimes increased the file size more than the sum of the constituents, and sometimes less. The reason for this is unclear.

5.2 Performance impacts

Significant performance degradation was observed as a result of obfuscation. This is in line with the findings of Skolka, Staicu, and Pradel (2019), who report that obfuscated code is measurably slower than regular code. Skolka, Staicu, and Pradel (2019) do not perform a separate warm-up phase prior to running their benchmarks, whereas in this study the initial iterations were removed as outliers if their performance is significantly worse than following iterations. The authors note that warm up affects user experience on websites and this study acknowledges that notion, but consistent iterations with low variation in performance were favored over single run results. Initialization overhead is reported as an additional result.

The effects of different obfuscation techniques on program performance were remarkably inconsistent across different programs and browsers. Literal transformations decreased performance on Google Chrome by 10–29%, but on Mozilla Firefox changes were negligible. Control flow flattening decreased performance on Google Chrome by 7–44%, and on Mozilla Firefox by 3–5%. Dead code injection decreased performance on Google Chrome by 0–29%, and on Mozilla Firefox by 0–15%. When all settings were applied in tandem, performance decreased on Google Chrome by 34–343%, and on Mozilla Firefox by 2–27%. The baseline settings generally affected performance by less than 2%. The only exception to this is P1 run in Google Chrome, were performance decreased by more than 17% when the baseline settings were applied. The reason for this is unclear, as the settings seemingly have minimal effects on the program structure and should not impact performance significantly.

When contrasted to previous research, the results appear sensible. Skolka, Staicu, and Pradel (2019) found an average increase of 16–37% in execution time when obfuscation was applied, but similarly to this study, the results were not consistent across all tested programs. Unfortunately, little other academic research has been conducted on the performance of obfuscated programs, as most performance-centric studies focus on the performance of obfuscation detection techniques.

The LT settings produced measurable losses in performance in all programs when executed in Google Chrome, but none at all in Mozilla Firefox. According to the documentation (Kachalov 2016), some losses in performance should be expected when certain LT tech-

niques are applied, but these were not consistently observed in both browsers. The documentation also states that heavy application of CFF can incur significant losses in performance. This effect was observed to some degree, and it is most apparent in the computationally expensive P3. The DCI settings also produced a measurable loss in performance, particularly for P2 in Mozilla Firefox, although the documentation makes no note of this. A strange effect was also observed in that DCI did not affect P2 performance in Mozilla Firefox when applied in tandem with all other techniques. However, this same result was produced multiple times. Initialization overheard increased consistently as more obfuscation methods were applied and the file sizes grew. This result is intuitive, as large obfuscated files will require more time to parse and evaluate when they are loaded in the browser. The most significant performance degradation was observed when all of the selected techniques were applied in tandem. All programs exhibited worse performance in both browsers when all techniques were applied, but the losses in performance varied substantially from 2% to 343%.

Significant differences in performance can be observed between the two browsers used. These measurements support the findings of Selakovic and Pradel (2016), who note that most optimizations do not consistently improve performance in both V8 and SpiderMonkey. Although the obfuscation methods used in this study would be more appropriately classified as deoptimization, the principle still applies. Many of the performance bottlenecks mentioned in their work can be produced by excessive obfuscation, such as memory bloat, redundantly repeated patterns and inefficient use of collections. Differences between the browsers were most apparent in P3, where performance was significantly lower in the Mozilla Firefox browser, regardless of the obfuscation methods applied or even the lack of any obfuscation. Further investigation revealed that WebGL performance issues have previously been reported in Mozilla Firefox, although the degraded performance could also be a result of implementational details in the application code. Furthermore, Mozilla Firefox did not suffer significant losses in P3 performance as a result of obfuscation, although the DCI setting slowed the application down slightly and increased variation between iteration execution times.

The browsers also exhibited differences in CPU and RAM load, which were not explicitly reported in the results. Mozilla Firefox struggled particularly with P1 when obfuscated using

the DCI settings and reserved significantly more memory than Google Chrome. Conversely, Google Chrome exhibited higher CPU load during the execution of P3. These observations may be due to implementational differences between the V8 and SpiderMonkey JavaScript engines, although the browsers may also contain other components which incur additional overhead on the execution. The setup used in the study aimed to keep the environment as free as possible from such impairments, but differences were observed nonetheless.

5.3 Threats to validity

Standard deviation was low across all measurements, with the exception of transformed P3 in the Google Chrome browser. Initial iterations which exhibited significantly slower performance than others were considered outliers and removed from the sample. However, in a real world scenario, the initial iterations may often be the most significant ones, as such programs are not designed to be executed multiple times in succession. It should thus be noted that for this particular program, heavy obfuscation may have a much larger impact than the reported numbers suggest. It is unclear what causes the initial overhead, but as JavaScript files need to be fully parsed before their execution can begin, it may be due to runtime optimizations performed by the engine.

Since the experiment was conducted on a home computer, the results may have been influenced by outside factors. Although the system was set up in a manner where interference from the passive load of other programs would be minimal, the operating system will nevertheless perform system interrupts and autonomous tasks beyond our control. Setting up a fully controlled environment was however not realistic within the scope of this study. A somewhat comparable performance measurement was conducted by Selakovic and Pradel (2016), where JavaScript programs were executed in a newly launched virtual machine. Following an approach similar to theirs may prove to be beneficial for future studies.

The programs constructed for the study focus purely on JavaScript code execution, which makes them somewhat elementary when compared to real world production applications. In reality, JavaScript code executed in the browser is seldom a major bottleneck, and most issues in the responsivity of a website stem from requests performed over a network con-

nection. Furthermore, the code in this study was stored and executed fully within the local environment, which means none of the large obfuscated files needed to be transferred over a network connection. Thus, the results of this study may not fully generalize to contemporary web applications, as they embody different characteristics.

Furthermore, techniques such as control flow flattening and dead code injection can be applied at an arbitrary intensity. In this study, all of the techniques were applied at the highest available threshold, which results in the highest level of obfuscation, but also produces larger files and less performant code. It is easy to see that one could inject an arbitrary amount of dead code and increase the file size infinitely, which means the results should be interpreted with the technique application intensity in mind.

Apart from a degraded performance, no changes in the functionality of the programs were observed. Skolka, Staicu, and Pradel (2019) observed an opposite impact, although their experiment was concerned with programs of a heavier focus on correctness. Furthermore, the obfuscation in this study was performed using a completely different tool than any of the ones applied in their research, and the JavaScript Obfuscator by Kachalov (2016) may well preserve the semantics of the original code better than other tools.

The random nature of some of the available techniques can also produce significant variation across multiple applications of the same settings. This study provided the obfuscator with a static seed value to combat this issue, but it should be noted that results may vary significantly if the randomization is tweaked even slightly.

6 Conclusions

6.1 Overview of contributions

This study offered an overview of the current body of research in the field of obfuscation, and performed a novel experiment to measure the impacts of different obfuscation techniques. The results were generally in line with those presented in previous work on which this study expanded, but the effects of the studied techniques were too inconsistent across different programs and browsers to draw definitive conclusions on the performance impacts of obfuscation.

Thus, this work concludes that the studied obfuscation techniques can have a clearly measurable impact on the file size and performance of programs, but further studies should be conducted to better measure the performance impact on different types of programs and browsers.

6.2 Future work

Future work could perform a similar experiment in a more controlled environment to limit the effect of random variables. Furthermore, this study focused its limited contribution to a comparison of several obfuscation techniques as implemented in a singular library. For a more complete review of the available technology, multiple obfuscation tools should be used, as they may contain significant differences even in the implementation of similar obfuscation techniques. More advanced techniques were not touched upon in this study and should be analyzed as their use becomes more prominent.

It was also noted that browsers exhibit differences in performance when obfuscated code is executed, but this study limited its scope to Google Chrome and Mozilla Firefox. Popular browsers not considered in this study include Safari, Opera and Microsoft Edge, between which performance measurements may also vary significantly. Research focused on other browsers could offer a more complete view on the effects of obfuscation. Furthermore, as mobile devices continue to increase in popularity, studies should also be conducted on

browsers developed for mobile platforms as their underlying architecture is different from desktop devices and performance may vary significantly.

As obfuscation techniques and JavaScript engines continue to evolve, more studies should be conducted to measure the effects of new techniques in addition to old ones. Furthermore, research on performance impacts is still lacking, and should be focused on more as obfuscation techniques continue to be applied in web development. Thus, obfuscation remains an important and consequential field of study for the foreseeable future.

Bibliography

10 Best Design. 2009. "Dirty Markup". Visited on March 21, 2022. https://www.10bestdesi gn.com/dirtymarkup/.

AbdelKhalek, Moataz, and Ahmed Shosha. 2017. "JSDES: An Automated De-Obfuscation System for Malicious JavaScript". In *Proceedings of the 12th International Conference on Availability, Reliability and Security.* ARES '17. Reggio Calabria, Italy: Association for Computing Machinery. ISBN: 9781450352574. https://doi.org/10.1145/3098954.3107009. https://doi.org/10.1145/3098954.3107009.

Anderson, Benton J, and Jesse G Meyer. 2022. "Finding the Optimal Human Strategy for Wordle Using Maximum Correct Letter Probabilities and Reinforcement Learning". *arXiv preprint 2202.00557.*

Barak, Boaz, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. "On the (Im)possibility of Obfuscating Programs". In *Annual International Cryptology Conference,* 1–18. Springer.

ben-sb. 2021. "JavaScript Deobfuscator". Visited on March 21, 2022. https://deobfuscate.io.

Caspersen, Michael E, and Henrik Bærbak Christensen. 2000. "Here, There and Everywhere – on the Recurring Use of Turtle Graphics in CS 1". In *ACM International Conference Proceeding Series,* 8:34–40.

Catuhe, David. 2013. "Babylon.js". Visited on April 6, 2022. https://www.babylonjs.com/.

Chenette, Stephan. 2008. "The Ultimate Deobfuscator". Visited on March 21, 2022. https://www.slideshare.net/schenette/the-ultimate-deobfuscator-toorcon-san-diego-2008.

Cimpanu, Catalin. 2019. "Mozilla announces ban on Firefox extensions containing obfuscated code". ZDNet. Visited on March 9, 2022. https://www.zdnet.com/article/mozilla-announces-ban-on-firefox-extensions-containing-obfuscated-code/.

Claburn, Thomas. 2018. "Google taking action against disguised code in Chrome Web Store". The Register. Visited on March 9, 2022. https://www.theregister.com/2018/10/02/google_chrome_web_store/.

Collberg, Christian, Clark Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations.*

Crockford, Douglas. 2001. "JSMin". Visited on February 28, 2022. http://www.crockford.com/jsmin.html.

Daft Logic. 2008. "Daft Logic JavaScript Obfuscator". Visited on March 1, 2022. https://www.daftlogic.com/projects-online-javascript-obfuscator.htm.

Ecma International. 2022. "ECMAScript® 2022 Language Specification". Visited on February 25, 2022. https://tc39.es/ecma262/.

Fang, Yong, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. 2022. "JStrong: Malicious JavaScript Detection Based on Code Semantic Representation and Graph Neural Network". *Computers & Security* 118:102715. ISSN: 0167-4048. https://doi.org/https://doi.org/10.1016/j.cose.2022.102715.

Feinstein, Ben, and Daniel Peck. 2007. "Caffeine monkey: Automated Collection, Detection and Analysis of Malicious Javascript". *Black Hat USA* 2007.

Goldwasser, Shafi, and Guy N Rothblum. 2007. "On Best-Possible Obfuscation". In *Theory of Cryptography Conference,* 194–213. Springer.

Hammad, Mahmoud, Joshua Garcia, and Sam Malek. 2018. "A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products", 421–431. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery. ISBN: 9781450356381. https://doi.org/10.1145/3180155.3180228.

Harstein, Blake. 2009. "jsunpack". Visited on February 28, 2022. http://jsunpack.blogspot.com/.

Hieu, Le, Markopoulou Athina, and Shafiq Zubair. 2021. "CV-Inspector: Towards Automating Detection of Adblock Circumvention". In *Network and Distributed System Security Symposium (NDSS).*

Hilbert, David. 1891. "Ueber die stetige Abbildung einer Linie auf ein Flächenstück". *Mathematische Annalen* 38:459–460.

Jovanovic, Nenad, Engin Kirda, and Christopher Kruegel. 2006. "Preventing cross site request forgery attacks". In *2006 Securecomm and Workshops,* 1–10. IEEE.

Jscrambler. 2017. "Jscrambler 101 - Control Flow Flattening". Visited on March 9, 2022. https://blog.jscrambler.com/jscrambler-101-control-flow-flattening.

Kachalov, Timofey. 2016. "JavaScript Obfuscator". Visited on March 29, 2022. https://github.com/javascript-obfuscator/javascript-obfuscator/.

Kim, Byung-Ik, Chae-Tae Im, and Hyun-Chul Jung. 2011. "Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation". *International Journal of Advanced Science and Technology,* number 26, 19–32. Visited on February 23, 2022. https://www.earticle.net/Article/A147393.

Król, Karol, and Dariusz Zdonek. 2020. "The Impact of Code Minification on Map Application Performance". *Geomatics, Landmanagement and Landscape* 2:41–49. https://doi.org/10.15576/GLL/2020.2.41.

Langdon, Brett. 2014. "JS NICE". Visited on March 21, 2022. http://jsnice.org/.

Lekies, Sebastian, Ben Stock, and Martin Johns. 2013. "25 Million Flows Later: Large-Scale Detection of DOM-Based XSS". In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security,* 1193–1204. CCS '13. Berlin, Germany: Association for Computing Machinery. ISBN: 9781450324779. https://doi.org/10.1145/2508859.2516703.

Lenz, John, and Nick Fitzgerald. 2011. "Source Map Revision 3 Proposal". Visited on February 28, 2022. https://sourcemaps.info/spec.html.

Lielmanis, Einar, and Liam Newman. 2013. "Online JavaScript Beautifier". Visited on March 21, 2022. https://beautifier.io/.

Likarish, Peter, Eunjin Jung, and Insoon Jo. 2009. "Obfuscated Malicious JavaScript Detection Using Classification Techniques". In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE),* 47–54. IEEE.

Lindenmayer, Aristid. 1968. “Mathematical Models for Cellular Interactions in Development II. Filaments with One-Sided Inputs”. *Journal of Theoretical Biology* 18 (3): 280–299.

Moog, Marvin, Markus Demmel, Michael Backes, and Aurore Fass. 2021. “Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild”. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),* 569–580. https://doi.org/10.1109/DSN48987.2021.00065.

Mowery, Keaton, and Hovav Shacham. 2012. “Pixel perfect: Fingerprinting canvas in HTML5”. *Proceedings of W2SP* 2012.

Murad, Khurram, Syed Noor-ul-Hassan Shirazi, Yousaf Bin Zikria, and Nassar Ikram. 2010. “Evading virus detection using code obfuscation”. In *International Conference on Future Generation Information Technology,* 394–401. Springer.

Ndichu, Samuel, Sangwook Kim, and Seiichi Ozawa. 2020. “Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement”. *CAAI Transactions on Intelligence Technology* 5 (3): 184–192.

Ndichu, Samuel, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. “A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors”. *Applied Soft Computing* 84:105721.

Ndichu, Samuel, Seiichi Ozawa, Takeshi Misu, and Kouichirou Okada. 2018. “A machine learning approach to malicious JavaScript detection using fixed length vector representation”. In *2018 International Joint Conference on Neural Networks (IJCNN),* 1–8. IEEE.

New York Times. 2022a. “The New York Times Buys Wordle”, ISSN: 0362-4331, visited on April 25, 2022. https://www.nytimes.com/2022/01/31/business/media/new-york-times-wordle.html.

———. 2022b. “Wordle”. Visited on April 25, 2022. https://www.nytimes.com/games/wordle/index.html.

———. 2022c. “Wordle Is a Love Story”, ISSN: 0362-4331, visited on April 25, 2022. https://www.nytimes.com/2022/01/03/technology/wordle-word-game-creator.html.

OpenJS Foundation. 2009. “NodeJS”. Visited on February 25, 2022. https://nodejs.org/en/.

Provos, Niels, Panayiotis Mavrommatis, Moheeb Rajab, and Fabian Monrose. 2008. "All your iframes point to us".

Prusinkiewicz, Przemyslaw, and Aristid Lindenmayer. 1990. *The Algorithmic Beauty of Plants.* Springer–Verlag.

Python Software Foundation. 2022. "Python 3.10.4 Documentation". Visited on April 6, 2022. https://docs.python.org/3.10/library/turtle.html.

Rajba, Pawel, and Wojciech Mazurczyk. 2021. "Data Hiding Using Code Obfuscation". In *The 16th International Conference on Availability, Reliability and Security,* 1–10.

Raychev, Veselin, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. "Learning Programs from Noisy Data". *ACM Sigplan Notices* 51 (1): 761–774.

Reiter, Andreas, and Alexander Marsalek. 2017. "WebRTC: Your Privacy is at Risk". In *Proceedings of the Symposium on Applied Computing,* 664–669. SAC '17. Marrakech, Morocco: Association for Computing Machinery. ISBN: 9781450344869. https://doi.org/10.1145/3019612.3019844.

Richards, Gregor, Christian Hammer, Brian Burg, and Jan Vitek. 2011. "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications", 52–78.

Roeder, Tom, and Fred B Schneider. 2010. "Proactive Obfuscation". *ACM Transactions on Computer Systems (TOCS)* 28 (2): 1–54.

Sakamoto, Yasutaka, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, and Masahide Nakamura. 2015. "Empirical study on effects of script minification and HTTP compression for traffic reduction". In *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC),* 127–132. https://doi.org/10.1109/DINWC.2015.7054230.

Sarker, Shaown, Jordan Jueckstock, and Alexandros Kapravelos. 2020. "Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage". In *Proceedings of the ACM Internet Measurement Conference,* 648–661. IMC '20. Virtual Event, USA: Association for Computing Machinery. ISBN: 9781450381383. https://doi.org/10.1145/3419394.3423616.

Selakovic, Marija, and Michael Pradel. 2016. "Performance Issues and Optimizations in JavaScript: An Empirical Study". In *Proceedings of the 38th International Conference on Software Engineering,* 61–72. ICSE '16. Austin, Texas: Association for Computing Machinery. ISBN: 9781450339001. https://doi.org/10.1145/2884781.2884829.

Serafim, Tiago. 2016. "obfuscator.io". Visited on March 29, 2022. https://obfuscator.io/.

Skolka, Philippe, Cristian-Alexandru Staicu, and Michael Pradel. 2019. "Anything to Hide? Studying Minified and Obfuscated Code in the Web". In *The World Wide Web Conference,* 1735–1746. WWW '19. San Francisco, CA, USA: Association for Computing Machinery. ISBN: 9781450366748. https://doi.org/10.1145/3308558.3313752.

Solomon, Cynthia J, and Seymour Papert. 1976. "A Case Study of a Young Child Doing Turtle Graphics in LOGO". In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition,* 1049–1056.

Son, Sooel, and Vitaly Shmatikov. 2013. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". In *NDSS Symposium 2013.* Internet Society.

Swire, Peter P. 2004. "A model for when disclosure helps security: What is different about computer and network security?" *Journal on Telecommunications and High Technology Law* 3:163.

Al-Taharwa, Ismail Adel, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. 2014. "JSOD: JavaScript obfuscation detector". *Security and Communication Networks,* number 8 (6): 1092–1107. https://doi.org/10.1002/sec.1064.

Verhoeff, Tom. 2010. "3D Turtle Geometry: Artwork, Theory, Program Equivalence and Symmetry". *International Journal of Arts and Technology* 3 (2-3): 288–319.

Vukušić, Mirko. 2011. "benchmark.js". Visited on March 29, 2022. https://github.com/bestiejs/benchmark.js/.

———. 2017. "JSBench.Me". Visited on March 29, 2022. https://jsbench.me/.

W3Techs. 2022. "Usage statistics of client-side programming languages for websites". Visited on February 25, 2022. https://w3techs.com/technologies/overview/client_side_language.

Xu, Wei, Fangfang Zhang, and Sencun Zhu. 2012. "The power of obfuscation techniques in malicious JavaScript code: A measurement study". In *2012 7th International Conference on Malicious and Unwanted Software,* 9–16. https://doi.org/10.1109/MALWARE.2012.6461002.

———. 2013. "JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code", 117–128. CODASPY '13. San Antonio, Texas, USA: Association for Computing Machinery. ISBN: 9781450318907. https://doi.org/10.1145/2435349.2435364.

Yahoo! 2007. "YUI Compressor". Visited on February 28, 2022. https://yui.github.io/yuicompressor/.

You, Evan. 2014. "Babylon.js". Visited on April 15, 2022. https://vuejs.org/.

Appendices

A Baseline options for JavaScript Obfuscator

```
{
  compact: false,
  controlFlowFlattening: false,
  controlFlowFlatteningThreshold: 0,
  deadCodeInjection: false,
  deadCodeInjectionThreshold: 0,
  debugProtection: false,
  debugProtectionInterval: 0,
  disableConsoleOutput: false,
  domainLock: [],
  domainLockRedirectUrl: 'about:blank',
  forceTransformStrings: [],
  identifierNamesCache: null,
  identifierNamesGenerator: 'hexadecimal',
  identifiersDictionary: [],
  identifiersPrefix: '',
  ignoreRequireImports: false,
  inputFileName: '',
  log: false,
  numbersToExpressions: false,
  optionsPreset: 'default',
  renameGlobals: false,
  renameProperties: false,
  renamePropertiesMode: 'safe',
  reservedNames: [],
  reservedStrings: [],
  seed: 1,
  selfDefending: false,
  simplify: false,
```

```
  sourceMap: false,
  sourceMapBaseUrl: '',
  sourceMapFileName: '',
  sourceMapMode: 'separate',
  sourceMapSourcesMode: 'sources-content',
  splitStrings: false,
  splitStringsChunkLength: 0,
  stringArray: false,
  stringArrayCallsTransform: false,
  stringArrayCallsTransformThreshold: 0,
  stringArrayEncoding: [],
  stringArrayIndexesType: ['hexadecimal-number'],
  stringArrayIndexShift: false,
  stringArrayRotate: false,
  stringArrayShuffle: false,
  stringArrayWrappersCount: 0,
  stringArrayWrappersChainedCalls: false,
  stringArrayWrappersParametersMaxCount: 2,
  stringArrayWrappersType: 'variable',
  stringArrayThreshold: 0,
  target: 'browser',
  transformObjectKeys: false,
  unicodeEscapeSequence: false
}
```

B LT options for JavaScript Obfuscator

```
{
  numbersToExpressions: true,
  splitStrings: true,
  splitStringsChunkLength: 1,
  stringArray: true,
  stringArrayCallsTransform: true,
```

```
  stringArrayCallsTransformThreshold: 1,
  stringArrayEncoding: [’rc4’],
  stringArrayIndexesType: [’hexadecimal-numeric-string’],
  stringArrayIndexShift: true,
  stringArrayRotate: true,
  stringArrayShuffle: true,
  stringArrayWrappersCount: 5,
  stringArrayWrappersChainedCalls: true,
  stringArrayWrappersParametersMaxCount: 5,
  stringArrayWrappersType: ’function’,
  stringArrayThreshold: 1,
  transformObjectKeys: true,
  unicodeEscapeSequence: true
}
```

C CFF options for JavaScript Obfuscator

```
{
  controlFlowFlattening: true,
  controlFlowFlatteningThreshold: 1
}
```

D DCI options for JavaScript Obfuscator

```
{
  deadCodeInjection: true,
  deadCodeInjectionThreshold: 1
}
```