Detection and Analysis of Obfuscated and Minified JavaScript in the Croatian Web Space

Toni Dujmović, Bruno Skendrović, Ivan Kovačević, Stjepan Groš
University of Zagreb Faculty of Electrical Engineering and Computing
Zagreb, Croatia
{toni.dujmovic, bruno.skendrovic, ivan.kovacevic, stjepan.gros}@fer.hr

***Abstract*—JavaScript libraries allow for faster and easier programming of web content. In order to conceal know-how secrets and malicious code, obfuscation is used. Obfuscation is a deliberate act of reshaping something to make it harder to understand. Commonly used obfuscators provide many methods that produce different obfuscated versions of the same source code. Minification has similar techniques to obfuscation, but unlike obfuscation, minification reduces the size of the code, which speeds it up and is its primary functionality. This paper provides an overview of obfuscation and minification and the methods therein. The developed software tool uses regex, entropy and word size to detect and distinguish minified and obfuscated JavaScript libraries. The result of running the software tool on a database of pages in the Croatian web space is presented. The results show a high presence of minified and a small number of obfuscated JavaScript libraries. This automated detection has proven to be faster and in some cases more accurate than manual detection of obfuscation and minification. Observed problems with the tool implementation are commented on and potential improvements are discussed at the end of the paper.**

***Keywords*—Croatian web space, cybersecurity, obfuscation, minification, automated detection, JavaScript**

I. Introduction

Software security has become an increasingly important issue in recent years as more and more sensitive information is stored and transmitted online [1]. Obfuscation and minification are two techniques that can be used to increase the security of software by making the source code harder for attackers to understand or modify.

Obfuscation is a process that makes the source code more difficult to understand by transforming it into an equivalent but more complex form. The goal of obfuscation is to make it more difficult for attackers to reverse engineer the code, steal intellectual property, or find vulnerabilities. Minification, on the other hand, reduces the size of the source code, making it more difficult for attackers to find and exploit vulnerabilities.

Obfuscation and minification share similar techniques, which is why the results may look similar to the human eye or to detection software. Obfuscation can also be present in malicious programs, as it can hide commonly used functions or code patterns that an antivirus program would detect as malicious code. This paper describes a developed software tool that attempts to distinguish between source code, minified JavaScript, and obfuscated JavaScript.

In this paper, we make the following contributions:

- We introduce a developed software tool for detecting code minification and obfuscation in JavaScript libraries. The tool uses these parameters: entropy, word sizes, and regex.
- We demonstrate the effectiveness of the tool by applying it to a dataset of JavaScript libraries from the Croatian web space and analyzing the results.

The remainder of this paper is organized as follows. In Chapter 2, we provide a comprehensive overview of the existing literature on obfuscation and minification. Chapter 3 presents a detailed background on obfuscation and minification, describing some methods in the processes. Chapter 4 comments on methods for detecting obfuscated and minified JavaScript and discusses selected parameters for the detection. Chapter 5 presents the results of applying the implemented software tool to a set of JavaScript libraries used in the Croatian domain. Chapter 6 of this paper is a discussion chapter, where an analysis of the results of the study is made. Finally, Chapter 7 discusses the implications of our findings and discusses possible improvements for the software tool.

II. Related Work

Obfuscation and minification are techniques used to improve the security of software code. They make it more difficult for attackers to understand or modify the code, but can also be used by malicious actors to hide their malicious code. The detection and analysis of obfuscated and minified JavaScript has been the subject of increasing research in recent years [2], as JavaScript is one of the most commonly used programming languages in web applications [3].

Previous studies have focused on various aspects of obfuscation, its use in malware production, and the impact on detection of such techniques. W. Xu and coauthors [4] have conducted a comprehensive study on the use of JavaScript obfuscation in malware and have shown that obfuscation is a common technique used by attackers to evade detection by security tools. W. Xu's work highlights the importance of developing effective methods to detect and analyze obfuscated JavaScript, as this is a critical aspect of ensuring the security of web applications.

Other studies focused on the detection of such files, such as the work of YH. Choi [5]. The study evaluated the effectiveness of different detection methods using

parameters such as N-gram, entropy, and word size. The study found that the parameters were effective for obfuscation detection, but also indicated the need for additional parameters to improve classification accuracy.

One of the biggest challenges in obfuscated and minified JavaScript detection is that these techniques are constantly evolving. Obfuscation and minification tools and methods are constantly being improved and updated, making it difficult to develop a single, comprehensive detection method that is effective against all forms of obfuscation and minification.

Despite these challenges, previous studies have shown that it is possible to effectively detect obfuscated and minified JavaScript using a combination of lexical analysis, semantic analysis, and code similarity analysis. These methods have proven effective in identifying different types of code transformations that occur during obfuscation and minification.

There are also recent studies on the use of machine learning based approaches to detect obfuscated JavaScript, as in the comprehensive study by S. Aebersold [6]. The results of this study demonstrate that machine learning-based methods exhibit high accuracy in detecting code transformations. However, it is important to note that a reliable dataset is essential to achieve this goal. In addition, future research could explore the possibility of augmenting machine learning with other methods and using the parameters discussed in this work to provide more comprehensive code transformation detection.

In a recent study by A. Alazab [2], a machine learning method was developed to detect malicious JavaScript. The proposed solution used a total of 170 features, including statistical and lexical categories, to detect both obfuscated and unobfuscated malicious code with 98% accuracy. The malicious code samples used in the study often had one or more obfuscation layers. The main features used in the detection system were entropy, string length, number of occurrences of certain strings, and JavaScript keyword frequency. While the results of the study show the effectiveness of the selected parameters in accurately detecting malicious JavaScript, it is worth noting that further research with even better parameters or other machine learning approaches could potentially achieve even higher accuracy in detecting and mitigating security threats.

In summary, the literature on obfuscation and minification detection in JavaScript has shown the effectiveness of using various parameters such as N-gram, entropy, and word size in combination with and without machine learning-based approaches. These studies have provided valuable insights into the current state of the art in obfuscation and minification detection. However, further research in this area is essential to advance the understanding and development of robust detection techniques.

III. Background

Obfuscation is the deliberate act of reshaping something to make it harder to understand. In programming, obfuscation is the transformation of code into a version that retains the same functions but makes it extremely difficult or even impossible to understand, replicate, or modify without additional tools. Program code is often obfuscated to protect intellectual property or trade secrets. A notable disadvantage of obfuscation is its potential misuse to circumvent the detection of malicious programs by antivirus software [7]. Obfuscation serves as a defense mechanism against reverse engineering and is often used in the development of malicious code to intentionally obscure its understanding and increase its complexity. Antivirus programs typically scan code as part of their detection mechanisms for commonly used features and patterns that are often associated with malicious programs. With obfuscation, developers do not develop new malicious code, but place a layer of obfuscation over important functions to hide from detection.

Obfuscation uses various methods to achieve the desired result, including "Control flow flattening"," "Dead code injection" and "String Array." "Control flow flattening" is a technique that aims to streamline the code flow. For this purpose, all fundamental blocks of code,such as function bodies, loops, and conditional branches are broken down and reassembled in an infinite loop. A switch statement is then used to determine which part of the code will continue to execute. Such a structure makes it much more difficult to follow the execution of the program, since the usual structures that make the code easier to read are no longer present. Figure 1, titled "Control Flow Flattening," shows an abstract representation of the transformation of a simple code structure into a flattened version by manipulating the flow. The colored blocks represent the separated code blocks, while the new black block represents the main switch command that governs the flow.

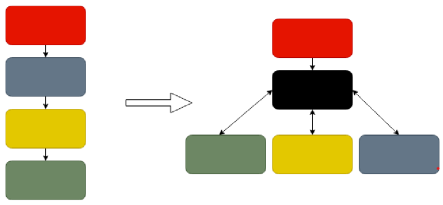

Fig. 1: Control flow flattening

The term "dead code" can be defined in two different ways. First, it refers to code that is never executed, such as a function enclosed in an "if" condition that is never satisfied. Second, it can refer to code whose output has no effect on subsequent code execution. Dead code can slow down execution time and consume system memory, making it difficult to understand the program. In the context of obfuscation, dead code is often inserted into source code to increase its complexity, complicate reverse engineering efforts, and increase the time required for deciphering.

Another method used to try to make the code more difficult to understand is to split strings into smaller parts and arrange them in arrays or array-like objects.

An example of this method would be splitting the word "classify" into "ssi", "fy" and "cla" and then storing the values in different parts of an array.

Minification is a technique used to reduce the size of code required to perform the same functionality. It is commonly used in the implementation of web pages and script files. Minification is an important method to optimize page loading times for users, as it speeds up page rendering. It also benefits users with limited internet connection as they consume less data compared to non-minified pages. The difference between the original and the minified jQuery JavaScript library version 3.1.1 is 176 kb [8]. Minification removes unnecessary or redundant elements from the source code, such as whitespace, comments, and naming conventions. This process results in more compact and efficient code that retains the same functionality as the original code.

After minification, the code becomes a less readable version that has the same functionality. For this reason, it can be said that minification is a type of obfuscation. Obfuscated and minified files are easy to distinguish from the files on which such operations were not performed, but in some cases it is difficult to distinguish minification from obfuscation. The goals of minification are different from those of obfuscation, but to minify the program code, some of the same methods are used. Some of these methods include shortening variable names, removing delimiters, and refactoring code. Shortening variable names reduces the size needed to store the code, refactoring code involves methods similar to "Control flow flattening".

IV. Detecting obfuscated and minified JavaScript

There are several ways to detect obfuscated JavaScript code. One way to detect obfuscated JavaScript code is to identify unusual or hard-to-read code patterns, such as long or overly complicated statements or unconventional naming conventions for variables and functions. Another approach to detecting obfuscated JavaScript code is to check for the presence of unnecessary or redundant code or statements, as this could be an indication of obfuscation techniques being used. In addition, developers can use a decompiler, a tool that can convert compiled code back into a more readable form, to understand and detect obfuscated code.

Minified code can also be difficult to read and understand since it removes variable names, comments, and block delimiters that serve to make the code readable but are not necessary for its execution. However, there are several tools that can help developers deal with minified code. One option is to use a code beautifier, which reformats and reorganizes the code to make it more readable. Another option is to use a source map [9], a file that maps the original, unminified code to the minified code, making it easier to debug and understand. Also, a debugger can be used to better understand the code and identify potential problems. In addition to the challenges posed by obfuscated and minified code, there are also security risks to consider. Malicious actors can use obfuscation and minification techniques to conceal malicious code, making it harder to detect.

In this work, the software tool initially categorizes JavaScript code into three categories: Minified Code, Obfuscated Code, and Source Code. This categorization is done through a separation process in which the tool identifies and distinguishes code that has been minified or obfuscated and code that remains in its original source code form. The initial separation of minified and obfuscated code from source code was performed using 5 parameters. The parameters increase the security index variable by 1 or 2, depending on the value given to the parameters from the given JavaScript. The specified parameters are:

- Regexes
- Entropy
- Size of the biggest word
- Average word size
- The ratio of the largest word to the number of characters of the entire file
- File size

After analyzing about 1000 obfuscated and minified JavaScript libraries crawled from the Croatian web space, regexes were written that could be used to detect the commonly used methods and patterns in their implementations. Some of the notable ones are regexes that detect parts of the code where variable names are used with one letter in a row and have been given values with one letter. This pattern could be used in minification to shorten variable names because the code no longer needs to be readable, or in obfuscation to shuffle values so that the code is harder to understand. Another regex was written for unusual return values that had been noticed in obfuscated code. Programmers would not normally overcomplicate code if it were possible to write it in a clearer way. For this reason, they would never write a return statement with only 3 Boolean values in it. In JavaScript, an empty array evaluates to false when computed as a boolean value, and the two exclamation points change the value to true and back to false, as seen in the second regex example. Since such code would never be written by a human, the regex was implemented to detect such patterns. Another regex was implemented to recognize a large number of "|" characters. In normal coding, such characters would not appear in the code, but in obfuscation, these characters are used in the string array method mentioned earlier, where strings in the code are broken into smaller pieces and stored in array-like structures. In this example, the large amount "|" simulates an array-like structure, where splitting the string by the "|" returns an array.

"[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,[a-zA-Z]=.,"

"return !!\[\]"

"\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|"

In computer science, entropy refers to a mathematical value that measures the degree of randomness in a set of characters. For JavaScript files, the standard entropy is usually 4.75, but after the process of minification or obfuscation, the entropy of a file tends to increase to around 5.1. By analyzing the entropy of a file using Shannon's formula [10], we can increase the probability of identifying obfuscated or minified code. The formula for calculating entropy is as follows:

$$H(X) = -\sum p(X) \log p(X)$$

The next three parameters analyze the word sizes in the code and compare them to word sizes normally found in source code files. After minification and obfuscation, many spaces, tabs, and other delimiters are removed from the code, resulting in larger words than in the source file. Following this logic also means that the average word sizes become larger as the word separators are removed. One problem that was encountered with this logic was that very small files can have large average word sizes, resulting in false positives. For this reason, the "largest word compared to the whole file" parameter was added. The last parameter contributes to the security index when the file size exceeds a certain threshold, as it was observed that larger files tend to be obfuscated, since JavaScript libraries do not usually exceed this threshold. For each parameter, an integer variable in the code would be incremented by 1 or 2, depending on the threshold value of that parameter. At the end of the classification, if the value is 4 or higher, the JavaScript is classified as minified or obfuscated. The threshold values for increasing the integer variable can be found at TABLE 1.

After separating the source code JavaScript from the obfuscated and minified JavaScript, the second round of classification begins, separating obfuscated and minified JavaScript. In the second round, a similar approach is taken as in the first round, but different regular expressions (regexes) are used and the thresholds for other parameters are adjusted to better distinguish between minified and obfuscated JavaScript code. This allows for a more accurate separation of the two types of code during analysis. Some types of obfuscation produce an exceptionally low entropy value, so a new threshold has been introduced when the entropy is below 3 to classify JavaScript as obfuscated. The regexes written for obfuscation detection are very specialized to the examples acquired during the research for the implementation of the software tool. In JavaScript, the eval() function is used to execute code written as a string. Some obfuscation tools may leave a signature such as "eval(function(p,a,c,k,e,r))" as a hint or indicator of deobfuscation. This signature can be used by analysts to identify and understand obfuscated code, as it often indicates the presence of code that is dynamically executed with eval(). Hexadecimal values can also be found in obfuscated code to change the values within the code without changing the values. This is not done in minification because it only increases the size of the code and does not increase the execution speed of the code. The regex for detecting hexadecimal values in the code can also be found below.

"eval\($function \ (p, a, c, k, e, r\)$"

"$\backslash b(0x[-fA - F]+) \ \backslash \ b$"

V. Experiments

The developed software tool was implemented in Python on a sample set of JavaScript libraries included in the Croatian web space. The source of the JavaScript libraries should not influence the written regexes because they were written according to commonly known methods of minification and obfuscation described in the previous two chapters. Only individual JavaScript libraries were used, as they were collected by their hash value. The tool is designed as a service that retrieves URL addresses from a database and retrieves their content using the Python request library. Once the content is retrieved, it is analyzed using six parameters as described in the previous chapter. The security index variable is then used to determine the type of the library based on the final index value.

Applying the software tool to a large number of JavaScript libraries revealed a significant flaw in the implemented solution. Despite its accuracy in detecting various forms of obfuscation, the solution suffered from a critical performance issue: catastrophic backtracking [11]. This problem occurs when a regular expression takes an exponential amount of time to match a string, resulting in a dramatic increase in processing time.

The cause of catastrophic backtracking was attributed to the use of complex regular expressions designed for a wide range of obfuscation techniques. However, these regular expressions also tended to match large amounts of irrelevant data, resulting in a large number of false positive matches. This in turn led to an exponential increase in the processing time required to match the regular expressions, resulting in catastrophic backtracking. One approach to solving this problem is to implement a timer during code processing or to skip very large files, since such files are more likely to be obfuscated. In such cases, only semantic and entropy analysis can be performed on these examples to optimize processing time. On an Intel(R) Core(TM) i5-1035G4 CPU @ 1.10GHz 1.50 GHz processor, the average processing speed for each sample was about half a second. However, for samples affected by the catastrophic backtracking problem, the processing time increased to five to ten minutes, resulting in slower classification.

The final test was performed on a set of 204842 JavaScript libraries crawled from the Croatian web space. The choice of the breakpoint for the security index had a large impact on the occurrence of false positives and false negatives when classifying libraries as obfuscated or minified. Selecting an appropriate breakpoint is critical to achieving accurate results and minimizing the occurrence of misclassification. In the first iterations of the tool, a value of 3 was used as the breakpoint for the classification security index, and there were no thresholds at which the security index would increase by 2. Using this logic, the

TABLE I: Security index increase table

	Increase by 1	Increase by 2
Regex	For each hit regex	
Entropy	Bigger than 5.05	Bigger than 5.3
Biggest word size	Bigger than 200	Bigger than 350
Average word size	Bigger than 11	Bigger than 45
Largest word compared to the whole file	Bigger than 10% of the whole file	
File size	Bigger than 1000000	

program contained a large number of false positives for source code JavaScripts that were classified as minified. If the analyzed library code contained an extremely long string, a function, or other built-in tools such as regex, these parts of the code contained very long amounts of text without spaces that would increase the security index by two in the case of false positives.

Within the libraries tested, there were also those that were so small that their largest word was 10 percent or more of the total size of the library, which increased the security index by one. Combining these two cases, the file is classified as obfuscated or minified, even though it is a source code library.

The tool proved to be better than manual classification in some cases. In certain cases where the code is so long that manual classification is difficult, the regexes were able to detect small functions that are obfuscated or minified and increase the security index, increasing the likelihood that the library will be classified as minified or obfuscated. These obfuscated functions in source code libraries usually contain know-how secrets or malicious functions that developers try to hide.

In this study, we present the results of a comprehensive analysis of a large sample set consisting of 204,842 randomly selected individual libraries. Our analysis revealed that a significant proportion of the samples were in either minified or source code format, while the proportion of obfuscated code was only 2% of the total set of examples. It should be noted, however, that the analyzed dataset was subject to potential bias because many identical files with the same hash value were treated as a single sample. Therefore, it is plausible that the actual number of minified files in the Croatian web space could be even higher than found in our analysis. This is especially true for commonly used libraries such as the popular JavaScript library jQuery, which may have the same hash value on different websites, leading to a possible discrepancy in the data. In summary, our results provide valuable insights into the prevalence of minification in the Croatian web as well as the prevalence of obfuscation in software libraries. This information could support the development of more effective tools for code analysis and optimization. The results are visually represented in the pie chart below.

After the experiment was completed, a study of the parameters of the classification system was performed. The entropy of the three file types was significantly different, with obfuscated code having the highest entropy of 5.51, followed by minified code at 5.23, and finally source

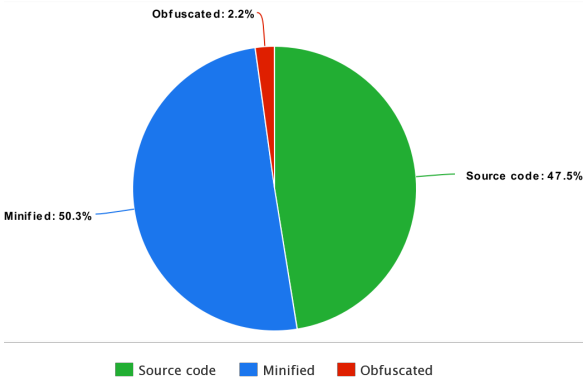

Fig. 2: Experiment results

code with an average entropy of 4.82. After manually analyzing some files that had an entropy of 5.3 or higher but were classified as source code files, it was clear that although the software had classified the files correctly, the files were written in such an ambiguous manner that the entropy was unusually high. The author of the code used few spaces, tabs, and new lines, and did not write the code in a way that made it easy to read, which increased its entropy. The code was also clearly not minified or obfuscated, as it followed other coding practices, such as clear and precise variable names, that are not followed in obfuscation or minification. When looking at the length of the longest word, it was found that the average length was 156 characters for source code, 2810 characters for minified code, and 18428 characters for obfuscated code. However, when analyzing the average word lengths, it was found that source code files had a length of 10 characters, minified files 116, and obfuscated files 69 characters. Although obfuscated files have a much longer largest word, the average word is shorter than the average minified word. One reason for this could be some functionality that creates very long words that are used in obfuscation but not in minification.

Some of the possible 3-pair parameters were examined for all file types. For source code files with entropy greater than 5.3, longest word size greater than 350, and average word size greater than 45, 2229 files were found. The best combination for minified files was entropy between 5.05 and 5.3, biggest word size greater than 350, and average word size greater than 45 with 20762 file matches. The combination with the most samples for obfuscated files was the same as for source code files, but there were 1170 obfuscated files.

Overall, the study provides valuable insight into the

characteristics of source code, minified code, and obfuscated code, as well as parameters that can be used for effective classification. These insights can be used to improve the performance and accuracy of code classification systems, leading to better detection of such methods.

VI. Discussion

The results of our study show that the implemented obfuscated JavaScript detection software tool effectively identifies the different types of code transformations that occur in software libraries. The results show that 45% of the libraries analyzed in the study were source code libraries, 53% were minified, and 2% were obfuscated.

The results are consistent with the expected trend that a significant portion of software libraries are minified, as minification is a widely used technique to improve the performance and security of software code. Nevertheless, the relatively low incidence of obfuscated libraries is unexpected, considering that obfuscation is also an important technique for bolstering software security and can be used for both legitimate and malicious purposes. It is important to conclude that even if obfuscation does not occupy a large portion of the JavaScript library ecosystem, even the 2% of these libraries can do great harm if they are malicious and will not be detected if not protected against.

It is important to note that the results of our study are specific to the dataset and software libraries analyzed, and may not be representative of the overall distribution of minified and obfuscated code in real-world applications. Nevertheless, these results provide valuable insights into the state of software security and the prevalence of various code transformation techniques.

The results of our study highlight the importance of using code transformation detection tools, as this can help identify vulnerabilities and improve the security of software applications. Our study also highlights the need to continue research on minification and obfuscation detection as these techniques continue to evolve and new methods emerge.

In summary, the results of our study demonstrate the effectiveness of the software tool for detecting minified and obfuscated JavaScript and provide valuable insight into the state of software security and the prevalence of various code transformation techniques. These results can inform future research on minification and obfuscation detection and be used to improve the security of software applications.

VII. Conclusion & Future Work

Obfuscation is the deliberate act of reshaping something so that it is harder to understand. It is used to keep secrets such as important code functions and to prevent the detection of malicious code. Minification involves reducing the amount of code, which is often used when implementing web pages and script files. Obfuscation and minification are two very similar processes that use a large amount of the same methods in their implementations.

Classifying obfuscated, minified, and source code libraries by hand is a straightforward but time-consuming process. This is because obfuscated libraries are on average more complex and therefore larger. It has been shown that using methods that examine word size and similar parameters in addition to regex and entropy provides a favorable result. The developed tool also speeds up the classification of libraries and confirms the success of the detection with the chosen combination of parameters. Further development of the tool would allow even more accurate detection of obfuscated and minified libraries by introducing new and better detection parameters. A limitation of using regex-based approaches is that they are inflexible and can only detect a fixed set of minification and obfuscation patterns. One possible way to mitigate this limitation is to use a machine learning approach, which could be more flexible in detecting a wider range of minification and obfuscation patterns.

The developed software tool has demonstrated its potential in accelerating the classification of source code, minified and obfuscated JavaScript. However, there is still room for improvement, such as exploring the use of machine learning-based approaches with additional detection parameters or better breakpoint adaptations for the existing ones to further increase the accuracy and efficiency of the tool.

References

[1] S. Shea, "Cybersecurity," https://www.techtarget.com/searchsecurity/definition/cybersecurity, accessed on April 2., 2023.

[2] A. Alazab, A. Khraisat, M. Alazab, and S. Singh, "Detection of obfuscated malicious javascript code," *Future Internet*, vol. 14, no. 8, p. 217, 2022.

[3] "Stack overflow developer survey 2022," https://survey.stackoverflow.co/2022/#technology, accessed on April 2., 2023.

[4] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012, pp. 9–16.

[5] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Future Generation Information Technology: First International Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings 1*. Springer, 2009, pp. 160–172.

[6] S. Aebersold, K. Kryszczuk, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting obfuscated javascripts using machine learning," in *ICIMP 2016 the Eleventh International Conference on Internet Monitoring and Protection, Valencia, Spain, 22-26 May 2016*, vol. 1. Curran Associates, 2016, pp. 11–17.

[7] N. Malviya, "Simple malware obfuscation techniques," https://resources.infosecinstitute.com/topic/simple-malware-obfuscation-techniques/, accessed March 14. 2023.

[8] Imperva, "Minification," https://www.imperva.com/learn/performance/minification/.

[9] "Use a source map," https://firefox-source-docs.mozilla.org/devtools-user/debugger/how_to/use_a_source_map/index.html, [Accessed: March 3. 2023.].

[10] E. Chiu, J. Lin, B. McFerron, N. Petigara, and S. Seshasai, "Mathematical theory of claude shannon. a study of the style and context of his work up to the genesis of information theory," *submitted for The Structure of Engineering Revolutions (MIT course 6.933 J/STS. 420J), nd*, 2018.

[11] J. Goyvaerts, "Catastrophic backtracking," https://www.regular-expressions.info/catastrophic.html, accessed March 12. 2023.