A Survay of Deobfuscation Techniques in JavaScript

1st XXXXXXX
Department of Electronic Information
Xi'an Jiaotong University
Xi'an, China
XXXXXXXXXX@stu.xjtu.edu.cn

***Abstract*—JavaScript (JS) is widely used in the development of website scripts and software design. When writing frontend programs, reading and understanding JS code is often an essential step. To protect their designs from being comprehended by other developers, some programmers employ code obfuscation techniques. Concurrently, certain malicious programs utilize JS obfuscation to reduce the efficacy of existing detection methods, posing threats to society. This has led to the emergence of JS deobfuscation technologies.**

Currently, there is a growing array of diverse JS deobfuscation tools that employ various techniques to transform previously obscure JS code into more understandable formats, thereby increasing the success rate of software detection of obfuscated malicious programs. This review focuses on JS deobfuscation techniques, categorizing them into four directions based on their underlying approaches: "symbolic execution," "compiler optimization," "machine learning," and "dynamic scheduling." While analyzing their strengths, we also identify their shortcomings, providing insights and considerations for the future development of more efficient JS deobfuscation techniques.

***Index Terms*—JavaScript, Obfuscation, Deobfuscation**

I. Introduction

JavaScript (JS) has become one of the core languages in modern web development, software development, and various application development. Whether frontend or backend developers, it is common to handle a large number of JavaScript-written program files in daily development to understand the program's operation, debug applications, or optimize code. Particularly in web development, JS is the key technology for achieving dynamic page effects and interacting with users. To ensure the proper implementation of program functions and optimize performance, developers usually need to read and understand these complex codes.

However, in certain application scenarios, especially for developers who wish to protect their intellectual property and enhance program security, directly exposing source code is not ideal. To address this, frontend developers often employ JS code obfuscation techniques. By using obfuscation tools, the originally clear and understandable code is transformed into something difficult to comprehend. The obfuscated code is typically renamed with meaningless variable names, comments are removed, and redundant code is added, all with the goal of making it difficult for other developers to reverse-engineer or maliciously alter the program. This technique not only prevents the code from being easily understood but also effectively protects the developer's creativity and intellectual property.

However, obfuscation techniques have also been adopted by malicious code creators as a means to hide harmful behaviors. Malicious programs use obfuscation to encrypt their concealed malicious intentions and actions, making the malicious code even harder for security personnel to detect and analyze. For undetected malicious code, once deployed in a production environment, it can lead to severe consequences such as data breaches, account theft, malicious advertisements, system hijacking, and more, causing significant losses for both businesses and users. The obfuscated code is typically difficult to understand and debug, which presents an even greater challenge for developers when tracking and fixing issues.

Research shows [1]–[4] that 95.90% of the top 100,000 websites on Alexa contain at least one obfuscated JS program, and about 71% of JS malware samples employ obfuscation techniques, significantly reducing the effectiveness and accuracy of detection. In the face of these challenges, developing efficient parsing tools becomes particularly important.

As a result, with the widespread use of obfuscation techniques, JS deobfuscation technologies have emerged. The core goal of JS deobfuscation is to restore the obfuscated code, which is originally difficult to understand, into a clear and comprehensible structure. This helps developers quickly restore the true intent of the program during debugging, auditing, and analysis processes. Deobfuscation techniques not only provide developers with convenient tools but also enhance the ability to detect malicious programs, thereby avoiding potential risks caused by the inability to decrypt and analyze the code.

For example, the tool JSNice [5] developed under the leadership of Professor Martin Vechev from ETH Zurich uses machine learning to assign meaningful names to variables that originally have obfuscated and unclear names. This transforms a program that is difficult to read into a format that can be easily understood by programmers.

Currently, deobfuscation tools like JSNice have been widely released on open-source platforms such as GitHub for use by developers and security researchers. These tools enhance code parsing efficiency, helping programmers quickly identify malicious code and reduce the potential losses caused by undetected malicious code. At the same time, with the continuous development of deobfuscation technologies, more and more innovative tools are emerging. These tools integrate multiple analysis methods and techniques, making JS deobfuscation more accurate and powerful.

Therefore, in today's world, where deobfuscation tools are constantly emerging, systematically summarizing and categorizing different JS deobfuscation techniques is particularly important. By analyzing the principles of these techniques, we can not only gain a clearer understanding of the advantages and application scenarios of existing tools but also gain deeper insights into their limitations and shortcomings. For developers, these analyses and summaries help them make more informed decisions when selecting the appropriate deobfuscation tool. At the same time, these experiences provide valuable references and a foundation for future development of more efficient and intelligent deobfuscation technologies, driving the continuous evolution of deobfuscation techniques in the security field.

A. *Major contributions*

A systematic review of existing JS deobfuscation techniques is conducted by classifying them into four categories based on the technologies they employ: "symbolic execution," "compiler optimization," "machine learning," and "dynamic analysis." The review summarizes the strengths of these techniques in dealing with JS obfuscation and proposes possible directions for future development.

II. Prerequisites

A. *JavaScript Obfuscation*

The core idea of JS obfuscation is to transform JS code that is originally easy for developers to understand (since the JS code they write and maintain is inherently readable) into JS code that is difficult for developers to understand. The process does not change the original semantics of the code, meaning that whatever operations the original JS code would perform, the obfuscated JS code will perform the same operations, and the final result will not change; the code will simply become difficult to comprehend.

This technology may encompass many aspects, including but not limited to minification obfuscation, OB obfuscation, variable obfuscation, string obfuscation, control flow flattening, etc. The purpose is to make the code as complex and difficult to understand as possible without altering the original semantics of the code.

For example, minification obfuscation involves removing spaces from formatted JS code, turning it into a continuous string. Without formatting tools, this code becomes difficult for programmers to understand.

It is important to note that JS obfuscation tools available in the market often do not use just one obfuscation technique. They typically combine multiple techniques to achieve an effect where the obfuscation results in a more complex outcome, often resulting in an effect where 1+1>2.

Here's an example of JS obfuscation. As shown in Figure 1, the original JS code simply outputs the string "Hello, World," which is easily understood by developers and can be modified or used as needed. If malicious code were written this way, it would be easily detected by either the developer or detection tools, and would not pose a threat.

However, after obfuscation, as shown in Figure 2, the code structure that was easy to understand is completely destroyed. Typically, without the aid of any tools, developers would struggle to understand that the code's essence is still to output the string "Hello, World." Although the code may look long and complex, the core semantics have not changed. Much of the added code either does not execute or executes with no side effects, having no impact on the core execution.

```
function hi() {
  console.log("Hello World!");
}
hi();
```

Fig. 1. JS code to output 'Hello, World'

```
function _0x5737(){var
_0x3de046=['13797910djQtgr','202NzEpzv','2273TLhUKk','6976590XeTkcs','4633335tPFIvf','460SzVdaa','1260225mbb
ZER','49056QtXjli','1736NJoeHX','42116DYgHBM'];_0x5737=function(){return _0x3de046;};return _0x5737();}function
_0x5e71(_0x1e04fb,_0x168fdd){var _0x57378a=_0x5737();return
_0x5e71=function(_0x5e7194,_0x30106f){_0x5e7194=_0x5e7194-0xb6;var
_0x3c5c20=_0x57378a[_0x5e7194];return
_0x3c5c20;},_0x5e71(_0x1e04fb,_0x168fdd);}(function(_0x41d572,_0x45db5e){var
_0x306ede=_0x5e71,_0x408f15=_0x41d572();while(!![]){try{var _0x4c3c37=-
parseInt(_0x306ede(0xbb))/0x1*(parseInt(_0x306ede(0xba))/0x2)+-
parseInt(_0x306ede(0xbd))/0x3+parseInt(_0x306ede(0xb8))/0x4*(parseInt(_0x306ede(0xbe))/0x5)+-
parseInt(_0x306ede(0xbc))/0x6+-parseInt(_0x306ede(0xb6))/0x7*(-parseInt(_0x306ede(0xb7))/0x8)+-
parseInt(_0x306ede(0xbf))/0x9+parseInt(_0x306ede(0xb9))/0xa;if(_0x4c3c37===_0x45db5e)break;else
_0x408f15['push'](_0x408f15['shift']());}catch(_0x8596b2){_0x408f15['push'](_0x408f15['shift']());}}}(_0x5737,0xc174
3));function hi(){console['log']('Hello\x20World!');}hi();
```

Fig. 2. The obfuscated JS code does not change the semantics, but is difficult to understand.

B. *JavaScript Deobfuscation*

As the opposite of JS obfuscation techniques, JS deobfuscation technologies function in exactly the reverse way. They use various techniques to restore obfuscated JavaScript code, which is originally difficult to understand and read, into a form that is easy for programmers to comprehend and more readable. This process typically includes restoring variable names, optimizing function structures, simplifying control flows, etc., with the goal of making the code clearer and easier to analyze.

However, it is important to note that JS deobfuscation does not aim to fully restore the obfuscated code to its original, un-obfuscated form. During the obfuscation process, some information is lost (for example, variable names are renamed to meaningless characters, such as changing a variable length to a), so the obfuscated code generally cannot be perfectly restored to its original state. While the functionality of the program may not be affected, the obfuscation process is irreversible, and some information cannot be recovered. Therefore, the goal of deobfuscation is not to "perfectly restore" the original code, but rather to make the code more readable and understandable without losing functionality.

The core idea of deobfuscation is to make it easier for programmers to understand the JS code, rather than fully restoring the code to its original form. It involves analyzing,

restructuring, and optimizing the obfuscated code to help developers quickly understand and modify the code. This is especially useful during debugging, vulnerability analysis, or security audits. Similar to obfuscation, there are many different deobfuscation tools and techniques available, and these tools typically do not rely on a single deobfuscation method but rather combine multiple techniques to achieve the best deobfuscation results.

Compared to JS obfuscation techniques, the combination of JS deobfuscation techniques is often more complex. The main reason is that the code structure involved in the deobfuscation process is often more intricate, and the diversity of obfuscation methods means that deobfuscation strategies need to be flexible and adaptable. For example, obfuscation techniques not only involve replacing variable names and function names but may also alter the control flow of the code (such as by inserting useless code segments or rearranging function calls). These obfuscation techniques make the deobfuscation task more complex.

Therefore, this review will classify and analyze the commonly used deobfuscation tools on the market from a technical perspective. These tools typically adopt various methods to restore the readability of code, based on different deobfuscation strategies and techniques, including but not limited to symbol restoration (e.g., restoring original variable names), control flow recovery (e.g., restoring originally complex control structures), function extraction and merging (e.g., separating inline code blocks into independent functions), etc. Through these means, deobfuscation tools help developers better understand and analyze obfuscated code, especially in complex JavaScript applications, where the role of deobfuscation technology is crucial.

In conclusion, JS deobfuscation is not a simple “restoration” process, but rather a gradual process of restoring code comprehensibility and enhancing maintainability. In modern development environments, especially during security analysis, vulnerability fixes, or code audits, deobfuscation techniques are particularly important.

III. Deobfuscation Technology

A. Symbolic Execution

Symbolic Execution is a technique used for program analysis, where the core idea is to replace concrete inputs in a program (such as integers, strings, etc.) with symbols (typically variables or expressions) and trace the flow of these symbols during the program’s execution. Through this approach, symbolic execution can simulate the behavior of a program under various inputs without actually running the program itself. Therefore, symbolic execution not only helps analyze the logic and behavior of a program but also uncovers potential vulnerabilities, errors, or malicious code within the program.

Unlike traditional dynamic execution, which observes the behavior of a program by running it with concrete inputs, symbolic execution analyzes each program statement, converts the inputs into symbolic quantities, and tracks how these quantities propagate throughout the program. Whenever the program encounters a branch, symbolic execution analyzes all possible paths based on the current symbolic inputs and continues to simulate the execution of those paths. Eventually, through symbolic operations, it generates symbolic representations of multiple program execution paths. This allows analysts to explore all potential behaviors of the program through symbolic reasoning without relying on actual program execution.

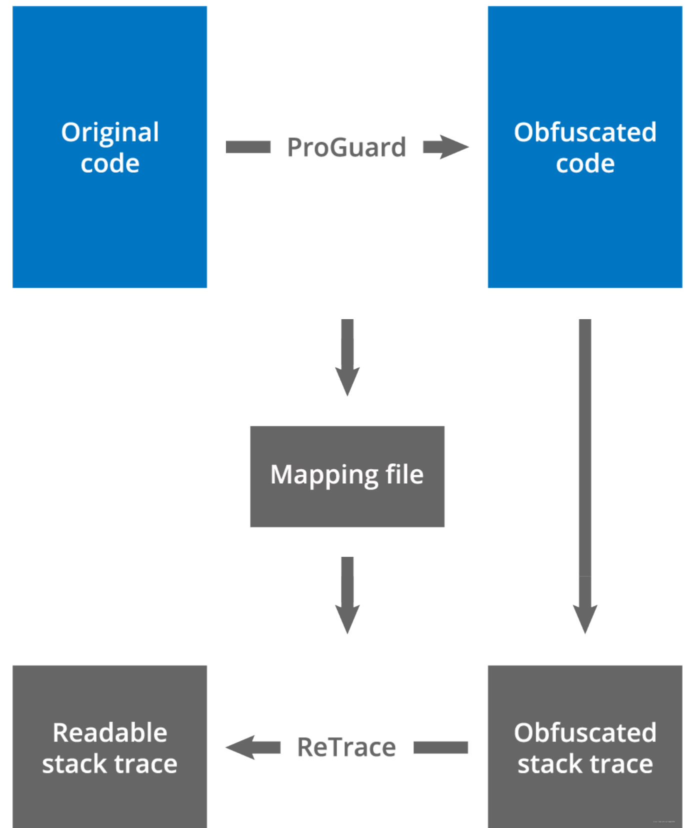

Fig. 3. Obfuscation and Deobfuscation

The advantage of symbolic execution is that it can provide a comprehensive understanding of a program’s execution logic, especially when dealing with complex control flow and data flow. Symbolic execution is particularly effective when dealing with obfuscated code, as obfuscation techniques often disrupt the original control flow through complex conditional branches, loops, or dynamic calculations, making the program’s execution paths more difficult to comprehend. Symbolic execution, by simulating the effect of various inputs on the program’s paths, helps uncover the true intent behind these complex control flows and identify and restore the program’s actual logic.

Obfuscation techniques not only change the control flow structure but also often store strings (such as variable names, function names, sensitive data, etc.) through encryption or encoding, which are dynamically decrypted at runtime. This makes the obfuscated code extremely difficult to understand

without proper analysis tools. Symbolic execution is especially useful in such scenarios because it can trace the execution process of the program, analyze the input and output of encryption or decryption functions, and help recover the original string data. Furthermore, symbolic execution can also simulate complex mathematical operations, XOR operations, etc., to uncover the true data or logic hidden behind these operations. By simulating these mathematical expressions, symbolic execution can effectively uncover hidden logic and data.

For example, Angr [6] and SymbEX [7] are two representative symbolic execution tools. Although they are not specifically designed for JavaScript (JS) deobfuscation, they possess symbolic execution capabilities and have demonstrated good performance in JS deobfuscation tasks. They can analyze the control flow and data flow of JS code through symbolic execution, helping analysts reveal hidden information in obfuscated code.

However, symbolic execution is not a panacea, and it has some limitations in practical applications. First, many obfuscated codes tend to have verbose structures, with large amounts of meaningless code added solely for obfuscation purposes. This not only increases the complexity of the analysis but also often leads to path explosion problems. As the number of paths in the program increases, the computational load of symbolic execution grows exponentially, making the analysis process very slow. This is especially true when there are many complex branches in the program, where the complexity of symbolic execution can increase sharply, potentially requiring significant computational time to complete path analysis.

Moreover, many obfuscation tools use complex mathematical operations, nested conditionals, and even XOR operations to hide the true logic of the program. These operations create very complex path conditions during symbolic execution, making it difficult for the solver to produce solvable path conditions within a reasonable time frame. These challenges often mean that symbolic execution cannot handle all deobfuscation tasks by itself.

As a result, symbolic execution is typically combined with other techniques to conduct more efficient JS deobfuscation. For instance, static analysis and dynamic analysis can complement symbolic execution. By combining different analysis techniques, a more comprehensive understanding of the structure and behavior of obfuscated code can be achieved, while mitigating the impact of path explosion problems. Symbolic execution's static analysis features, compared to dynamic analysis methods, save time by avoiding the need to run the program and help prevent malicious code from being executed, thus reducing the risk during the analysis process.

In the future, as symbolic execution technology continues to evolve, especially with improvements in optimization algorithms and solver performance, the application prospects of symbolic execution in deobfuscation are vast. It is expected that symbolic execution will continue to develop in the direction of increasing accuracy and reducing analysis time, playing a greater role in uncovering obfuscated variable names, control flow structures, and more. As symbolic execution tools continue to improve, they will become an important technique in the deobfuscation field, helping analysts tackle the increasingly complex challenges posed by obfuscation techniques.

B. Compilation Optimization

Deobfuscation is the process of transforming obfuscated code back into a version that is more comprehensible and easier to analyze, while compilation optimization techniques focus on improving the performance and efficiency of code by modifying its structure during compilation. While their core goals differ—deobfuscation aiming at clarity and optimization aiming at performance—some compilation optimization techniques can unintentionally or indirectly facilitate deobfuscation, especially in areas related to code simplification, restructuring, and clarity.

Compilation optimization techniques generally work by improving the quality of code, eliminating redundancies, removing ineffective or duplicate elements, and optimizing both control flow and data flow to enhance execution efficiency. Examples include eliminating unused variables and functions, removing dead code, merging duplicate code blocks, constant propagation, and loop unrolling. These improvements can significantly simplify the obfuscated code, transforming complex, hard-to-follow structures into more manageable and understandable ones. Obfuscators often add unnecessary or redundant elements or make control flows overly complex in order to disguise the program's functionality. Optimization techniques can reverse these efforts, automatically removing the unnecessary parts and making the code more readable and maintainable.

A common obfuscation technique is control flow flattening, which changes the natural execution order of the program, introducing confusion and complexity. By merging or altering conditional branches and loops into a single control structure, the program's original logic is obscured, making the code harder to interpret. In this case, optimization techniques such as code reordering, simplifying loop structures, or combining conditional branches can aid in restoring the original control flow, improving the efficiency of deobfuscation and enhancing code readability.

Dead Code Elimination (DCE) is another foundational optimization technique that removes code that is never executed—whether due to unreachable code paths or redundant statements. Obfuscators often insert dead code deliberately to confuse analysis tools and add to the program's size and complexity. During the deobfuscation process, dead code serves as an obstacle, making it difficult to identify the critical parts of the program. By applying dead code elimination during compilation, the redundant or unreachable segments are removed, leaving only the relevant code. This results in a more compact and manageable program, making it easier for deobfuscation tools to focus on the actual functional parts of the code. This is particularly valuable when working

with obfuscated JavaScript code, where unnecessary code can significantly slow down analysis.

In summary, while compilation optimizations and deobfuscation serve different purposes, the overlap between their methods can be harnessed to simplify obfuscated code and streamline the process of reversing obfuscation. Through techniques like dead code elimination, control flow optimization, and redundant code removal, deobfuscation can be made more efficient and effective, ultimately leading to clearer and more maintainable code.

Take UglifyJS [8] as an example. It is a widely used JavaScript code compression and obfuscation tool commonly used to minimize JavaScript code. However, while UglifyJS reduces the size of the code, it often makes the code harder to understand and debug. In such cases, the reverse operation, deobfuscation, can help restore the compressed code structure, variable names, and logic, making it easier to read. During deobfuscation, certain compilation optimization techniques (such as constant propagation and loop unrolling) can help restore the code structure, assist analysts in identifying constants or optimized loop structures, and speed up the understanding and reconstruction of the code. This reverse engineering process actually borrows ideas from compilation optimization, helping developers regain the readability of the code by restoring or simplifying its complex parts.

Although compilation optimization techniques can assist deobfuscation in certain scenarios, they are not a direct solution for deobfuscation tasks. The primary goal of deobfuscation is to make obfuscated code more understandable, while compilation optimization mainly focuses on improving execution efficiency. Therefore, compilation optimization techniques do not always directly promote code readability. In fact, many compilation optimization techniques may alter the original code structure, reorder, or merge parts of the code, which could make the code harder to read. For example, the compiler may perform excessive optimization, making the original logic more compact, but this compactness is not always helpful for human understanding. In JavaScript obfuscation, sometimes compilation optimization can “overwhelm” the code structure, making it difficult for the deobfuscator to restore the original program logic.

In fact, one of the core strategies of many obfuscation tools is to use compilation optimization techniques to disrupt the original syntax structure to confuse analysts. Obfuscators intentionally make the code structure and execution flow more complex by inserting useless code, changing the control flow, renaming variables, or even using encryption algorithms. In such cases, deobfuscation tasks not only involve restoring the obfuscated parts but also require reverse reasoning and code logic reconstruction. Compilation optimization techniques, in this case, can be used as a tool for reverse cracking, helping analysts identify and restore redundant code or complex logic generated during obfuscation.

In summary, while compilation optimization techniques and deobfuscation techniques have different goals, reasonable compilation optimization can assist the deobfuscation process in certain scenarios, especially when simplifying code and restoring logical structures. However, this does not mean that compilation optimization can completely replace deobfuscation methods; rather, it acts as an auxiliary tool in the deobfuscation process. How to appropriately apply compilation optimization techniques and choose the right compilers or optimization strategies to achieve the best results in deobfuscation tasks remains an important area of research in the current JavaScript deobfuscation field. With further exploration of these techniques, more efficient deobfuscation tools may emerge, better integrating compilation optimization and reverse analysis, thus enhancing the accuracy and speed of obfuscated code analysis.

C. Machine Learning

With the rapid development of artificial intelligence, an increasing number of advanced models have emerged, making significant strides in various domains. Among these, JavaScript (JS) deobfuscation tools that incorporate machine learning (ML) techniques have garnered particular attention. These tools aim to transform obfuscated JS code—originally difficult to understand—into a more comprehensible and readable format.

One of the most effective ways for a programmer to understand a program is through variable names. Well-chosen variable names provide meaningful context and hints about the code’s functionality. For example, when a developer encounters the window object in JS, they can immediately infer that the code likely interacts with browser window-related functions or parameters. This is because many JS objects and their properties are well-known and commonly used across applications, and front-end developers tend to design these objects with predefined functionalities, creating an easy entry point for other programmers to understand the purpose of the code.

However, after obfuscation, these descriptive variable names are typically replaced with random, meaningless characters—often a series of seemingly arbitrary letters or numbers. This change makes it much harder for developers to understand the code’s structure or functionality just by inspecting the variable names. Without meaningful names, it becomes a challenge to deduce the original intent of the code, slowing down both analysis and debugging processes. Thus, restoring meaningful names to obfuscated variables can significantly enhance the readability and understanding of the code.

JSNice is a prime example of a JS deobfuscation tool that leverages machine learning to address this problem. The tool’s primary focus is on transforming obfuscated JS variable names into more meaningful, contextually appropriate ones. The core idea behind JSNice is its ability to analyze the context of a given variable within the code and assign a name that reflects its intended use. Since the model has been trained on a vast corpus of non-obfuscated JS code, it can accurately predict which variable name would make the most sense in a particular context, even when the original name is nonsensical or random.

The power of machine learning in this context comes from its ability to understand patterns in variable naming conventions across a large dataset of well-written code. The model learns to associate variable names with their roles in different programming contexts—such as whether a variable is a function, an object, or a specific parameter. As a result, JSNice can automatically suggest more meaningful names, making the code easier to understand, maintain, and debug.

In addition to JSNice, other JS deobfuscation tools also utilize machine learning techniques to assign appropriate names to obfuscated variables. Tools like ProGuard, for instance, use similar ML-driven approaches to analyze the context in which variables appear and make educated guesses about their intended meanings. However, these tools often differ in their approach and the specific domains of JS obfuscation they target. Some may be optimized for certain coding patterns, libraries, or frameworks, allowing them to make more accurate predictions based on the specific coding style used in the project. As a result, these tools can be more precise in their variable naming, especially for highly specialized or domain-specific JS applications.

Overall, machine learning-based JS deobfuscation tools like JSNice and ProGuard represent a significant leap forward in the battle against obfuscation. By restoring meaningful variable names, they make it easier for developers to understand and analyze obfuscated code, improving both security and maintainability. As these tools continue to evolve, they will likely play an even larger role in simplifying code comprehension and improving the efficiency of reverse engineering efforts.

ProGuard [9], developed by Benjamin Bichsel, is a deobfuscation tool based on machine learning that focuses on Android system JS code. Its training data primarily targets JS code in Android applications, which is why it performs better in this domain.

Machine learning-based deobfuscation tools, such as JSNice, primarily focus on the task of renaming variables to restore the original meaning and functionality of the code. These tools are particularly effective in addressing the challenges posed by variable obfuscation, where meaningless names are assigned to variables to make the code harder to understand. However, when it comes to more complex obfuscation techniques, such as structural obfuscation, the performance of machine learning-based tools may not be as effective or reliable.

For example, when JSNice encounters JS code that has undergone structural obfuscation, the tool often fails to assign meaningful variable names. This is because structural obfuscation deliberately alters the normal flow and structure of the code, disrupting the logical sequence and control flow. JSNice, being primarily trained on well-structured, sequential code, struggles to adapt to these altered structures. As a result, it is less capable of accurately predicting the appropriate names for variables when the code's structure is no longer intact.

Structural obfuscation works by breaking the original sequential flow of the program, transforming it into a fragmented or "chunked" structure. This can involve techniques like converting sequential code into loop-switch combinations (e.g., while-switch statements), which flatten the language structure and obscure the natural flow of execution. While the semantic relationships between different code blocks may remain intact, the readability and logical progression are lost, making it much harder for deobfuscation tools to map the obfuscated code back to its original form.

For instance, Figure 3 illustrates how structural obfuscation can scramble the original JS code. A simple, linear program that executes instructions in sequence is transformed into a more complex, branching structure that introduces unnecessary loops or conditionals. The flow becomes harder to follow, even though the underlying logic remains unchanged. JSNice and similar tools that rely on sequential patterns struggle to interpret this transformed structure, often producing incorrect or nonsensical variable names as a result.

Given these challenges, the future of machine learning in JS deobfuscation must shift toward addressing structural obfuscation more effectively. One key area of research will be the development of machine learning models that can better understand and adapt to the structural changes introduced by obfuscation. This includes teaching models to recognize and map the relationships between obfuscated JS code with disrupted structure and non-obfuscated, sequential JS code. Language models, which have shown promising results in natural language processing, could be leveraged to improve machine learning's understanding of the program's structure and logic, even when traditional sequential patterns are disrupted.

Another promising direction for research is the combination of machine learning with other traditional deobfuscation techniques. Machine learning could be applied as a secondary step, where other methods—such as control flow analysis, dead code elimination, and pattern recognition—are used to process the code first. Once these techniques have simplified or partially deobfuscated the code, machine learning models could then take over to refine the analysis, especially in areas like renaming variables or restoring semantic meaning. This hybrid approach could potentially combine the strengths of both traditional and machine learning-based deobfuscation methods, leading to more robust and accurate results.

In summary, while machine learning has proven effective in dealing with variable renaming in obfuscated JavaScript, it faces limitations when applied to more complex obfuscation techniques like structural obfuscation. Future research will need to focus on enhancing machine learning's ability to handle these challenges, possibly through advancements in language models and the integration of multiple deobfuscation techniques. This multi-faceted approach holds the potential for more effective and comprehensive solutions to the problem of JS obfuscation.

D. Dynamic analysis

Dynamic analysis plays an important role in the field of JavaScript deobfuscation, especially when dealing with com-

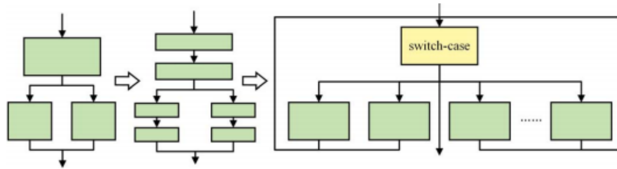

Fig. 4. JS Structure Obfuscation

plex obfuscation techniques. Dynamic analysis helps in understanding obfuscated code structures by executing the code and observing its behavior. Unlike static analysis, dynamic analysis allows us to observe the execution paths, data flows, variable values, etc., at runtime, providing additional clues and information during the deobfuscation process.

Many JavaScript obfuscators use control flow flattening to alter the execution paths, making the code harder to understand. Through dynamic analysis, we can track the program's control flow during execution and determine which paths are actually executed, thus helping to restore the original execution order and structure.

Some JavaScript obfuscators encrypt strings (such as Base64 encoding, hexadecimal conversion, etc.) to hide the actual code or strings. In static analysis, these encrypted strings are often invisible or difficult to understand. However, in dynamic analysis, we can observe the decryption process by executing the code and capture the actual plaintext content.

Many obfuscation methods encode data structures (such as arrays or objects) into obfuscated forms, such as storing string arrays through numeric indexing. Dynamic analysis allows us to track the restoration process of these obfuscated structures during code execution and identify the true meaning of the data.

Through dynamic analysis, developers can capture anomalies and errors during code execution, which often provide important clues about the code structure. For example, exceptions related to obfuscation might be thrown during execution, which can help analysts identify which parts of the code have been altered or obfuscated.

JSDetox [10] is a tool used for dynamic analysis of JavaScript code. It supports both static and dynamic analysis, helping to analyze complex obfuscated code. JSDetox allows developers to run obfuscated code in a browser environment, capture the code's behavior during execution, and reverse-engineer the original structure.

Additionally, Frida [11] can be used to dynamically analyze obfuscated JavaScript code, particularly by analyzing the obfuscated control flow and data flow to help reconstruct the pre-obfuscated code structure.

Therefore, dynamic analysis is a key technology in JS deobfuscation. It can directly reveal the real values of previously unreadable code. Obfuscation techniques like structure obfuscation, which alter the syntax structure of JS code, can also be effectively deobfuscated through dynamic analysis.

However, dynamic analysis has its drawbacks, one of which is the time consumption. Additionally, since it requires executing the code, ensuring the security of the code is another crucial concern.

In the future, dynamic analysis will likely continue to be combined with various other deobfuscation techniques. While leveraging its strengths, it can also benefit from the advantages of other deobfuscation techniques to compensate for its shortcomings.

IV. Conclusion

This review categorizes the technologies related to JS deobfuscation based on their principles into four aspects: "Symbolic Execution," "Compilation Optimization," "Machine Learning," and "Dynamic Analysis." While analyzing how these techniques can be applied to deobfuscate JavaScript code, it also summarizes their strengths, the domains and scopes in which they are most applicable, and suggests future directions for development. Through this summary, the review aims to provide ideas and references for the development of new JS deobfuscation methods in the future.

References

[1] Marvin Moog, 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. 569–580.

[2] Shaown Sarker, 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.

[3] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and CheolWon Lee. 2009. Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. In Future Generation Information Technology.Springer Berlin Heidelberg, Berlin, Heidelberg, 160–172.

[4] Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. 2021. Web Application Testing: Using Tree Kernels to Detect Near-duplicate States in Automated Model Inference. In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 1–6.

[5] Bogdan Vasilescu,Casey Casalnuovo,Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated JS names.Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.School of Computer Science Carnegie Mellon University, USA, Computer Science Department University of California, Davis, USA.2017:8.

[6] J. Doe, A. Smith,"SymbEX: A Tool for Symbolic Execution of Programs," [Online]. Available: https://github.com/simonw/symbex/blob/main. [Accessed: Nov. 13, 2024].

[7] M. E. Peinado, "Angr: A Python Framework for Analyzing Binaries," [Online]. Available: https://angr.io/. [Accessed: Nov. 13, 2024].

[8] M. H. L. D. "UglifyJS: JavaScript parser, minifier, compressor and beautifier, v3.14.0," [Online]. Available: https://github.com/mishoo/UglifyJS. [Accessed: Nov. 13, 2024].

[9] Richard M. Banks, Benjamin A. Chambers, Edward A. McBurney, Tilman P. Wolter, Antonio A. P. Tavares, Paul J. Trevorrow. Proceedings of the 10th Workshop on Security in High-Performance and Distributed IT Systems, 2017.

[10] C. A. Wang, "JSDetox: A JavaScript malware analysis framework," [Online]. Available: https://github.com/sirtori/JSDetox. [Accessed: Nov. 13, 2024].

[11] "Frida: A Dynamic Instrumentation Toolkit," [Online]. Available: https://frida.re/. [Accessed: Nov. 13, 2024].