A First Look at Code Obfuscation for WebAssembly

Shrenik Bhansali[†,*], Ahmet Aris[†], Abbas Acar[†], Harun Oz[†], and A. Selcuk Uluagac[†]
[†]Cyber-Physical Systems Security Lab, Department of Electrical and Computer Engineering,
Florida International University, Miami, Florida, USA
[*]Georgia Institute of Technology, Atlanta, Georgia, USA
Emails: sbhansali8@gatech.edu, {aaris, aacar001, hoz001, suluagac}@fiu.edu

Abstract

WebAssembly (Wasm) has seen a lot of attention lately as it spreads through the mobile computing domain and becomes the new standard for performance-oriented web development. It has diversified its uses far beyond just web applications by acting as an execution environment for mobile agents, containers for IoT devices, and enabling new serverless approaches for edge computing. Within the numerous uses of Wasm, not all of them are benign. With the rise of Wasm-based cryptojacking malware, analyzing Wasm applications has been a hot topic in the literature, resulting in numerous Wasm-based cryptojacking detection systems. Many of these methods rely on static analysis, which traditionally can be circumvented through obfuscation. However, the feasibility of the obfuscation techniques for Wasm programs has never been investigated thoroughly. In this paper, we address this gap and perform the first look at code obfuscation for Wasm. We apply numerous obfuscation techniques to Wasm programs, and test their effectiveness in producing a fully obfuscated Wasm program. Particularly, we obfuscate both benign Wasm-based web applications and cryptojacking malware instances and feed them into a state-of-the-art Wasm cryptojacking detector to see if current Wasm analysis methods can be subverted with obfuscation. Our analysis shows that obfuscation can be highly effective and can cause even a state-of-the-art detector to misclassify the obfuscated Wasm samples.

CCS Concepts

• **Security and privacy → Software security engineering**; *Malware and its mitigation.*

Keywords

WebAssembly, Wasm, Obfuscation, Cryptojacking

ACM Reference Format:
Shrenik Bhansali[†,*], Ahmet Aris[†], Abbas Acar[†], Harun Oz[†], and A. Selcuk Uluagac[†]. 2022. A First Look at Code Obfuscation for WebAssembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '22), May 16–19, 2022, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3507657.3528560

1 Introduction

WebAssembly (Wasm) is a compilation target for high-level languages located within browsers that facilitates the creation of high-performance web applications (e.g., games, portable languages, cryptographic computation) that run at near native speed [20]. Wasm has permeated throughout the wireless domain due to its suitability for mobile agent solutions. However, due to the versatility of Wasm, not all Wasm programs are safe and benign. Over half of all Wasm instances in the Alexa Top 1 million websites were found to be of malicious nature, with most of these Wasm instances consisting of cryptojackers [11]. Wasm malware is not just limited to cryptojacking, since Wasm's unrestricted access to browser functionality and the JavaScript API enables a variety of potential attacks.

In order to combat Wasm-based malware and detect malicious Wasm programs (e.g., cryptojackers), numerous static analysis-based defenses have been proposed [9, 12, 18, 19]. All of these detection methods rely on the assumption that Wasm modules are unobfuscated or obfuscated using primitive techniques. However, a thoroughly obfuscated cryptojacking malware can become resistant to static analysis techniques such as matching or fuzzy hashing [14]. Therefore, existing security methods can be circumvented, allowing malicious Wasm programs to run undetected. Out of all Wasm instances in the wild, only 0.2 % of them were found to be obfuscated [11].

The nature of the Wasm compilation process requires multiple distinct compilation stages before execution. Therefore, understanding how obfuscation effects the final Wasm program running in the browser after these stages is necessary, for both benign software security and creating effective Wasm malware detection methods. Given the security threats that are presented by Wasm [8], along with the immense presence of malicious Wasm programs, an understanding of obfuscating and hiding Wasm programs is essential. However, in the literature, there are no in-depth discussions or analysis of obfuscation in Wasm.

In this paper, we address this gap and perform the first look at code obfuscation for Wasm. We perform two sets of experiments to analyze the effect of Wasm obfuscation. In the first set of experiments, we apply seven different obfuscation techniques and utilize the Cosine Similarity of the resulting obfuscated and unobfuscated web applications as a metric to evaluate the effectiveness of the obfuscation techniques. While this metric gives us the result of how successful the obfuscation technique is, it alone cannot tell us if the obfuscation would be able to affect Wasm static analysis tools. Therefore, in the second set of experiments, we feed the obfuscated benign and malicious samples to a state-of-the-art cryptojacking detection system named MINOS [12] Our results show that MINOS cannot detect any of our obfuscated cryptojacking malware samples as well as produces false positives on some benign Wasm web applications such as games.

Contributions: The contributions of this paper are as follows:

- We, first time in the literature, investigate the feasibility of obfuscation of the Wasm web applications.
- We present a comparative analysis of multiple obfuscation techniques for Wasm across a variety of different types of Wasm-based web applications.
- We demonstrate a use-case that illustrates how obfuscated Wasm web applications can affect the static Wasm analysis tools in discriminating benign and malicious Wasm applications.

Organization: The organization of our paper is as follows: Section 2 gives background information. Section 3 defines the problem scope and threat model. Section 4 details the obfuscation techniques and results. Section 5 discusses the related work. Finally, Section 6 concludes the paper.

2 Background

2.1 WebAssembly

WebAssembly (Wasm) is a low-level bytecode that runs in browsers and acts a compilation target for higher-level languages. Wasm is designed with efficiency in mind, executing programs within browsers at near-native speed [20]. It additionally prides itself on its safety, with programs being executed in a memory-safe environment, and platform independence, being able to run in browsers, integrated environments, and VMs. The safety and platform independence is achieved via a stack-based virtual machine in which the Wasm code is executed

The most popular compiler for Wasm development is Emscripten, a source-to-source LLVM/clang-based compiler and toolchain for all LLVM source languages. It takes in a C/C++/Rust source file, and outputs a Wasm module. This Wasm module is then instantiated within a browser with JavaScript. Once instantiated, the machine instructions held in the module are run within a JS stack machine, converting the Wasm module's instructions into binary and executing within the browser. During this process, the WebAssembly module has access to the JS context in addition to browser functionality, therefore also having access to the computer's computational resources.

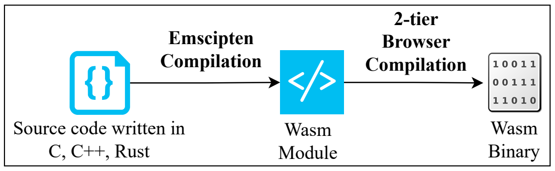
Emscipten Compilation
2-tier Browser Compilation
10011
00111
11010
Source code written in C, C++, Rust
Wasm Module
Wasm Binary

Figure 1: Compilation process of Wasm applications.

Before Wasm modules are run in the JS stack machine within the browser, they go through a second compilation process by the Wasm engine of the browser (e.g., V8 for Chrome[1] and SpiderMonkey[2] for Mozilla), as shown in Figure 1. In this process, the Wasm engine of the browser compiles the Wasm module twice to convert the Wasm module to an optimized machine code for the platform. For instance, V8 engine of Chrome uses Liftoff and TurboFan[1] in this process while Mozilla utilizes Cranelift and IonMonkey[2].

[1]https://v8.dev/blog/liftoff
[2]https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/spidermonkey.md

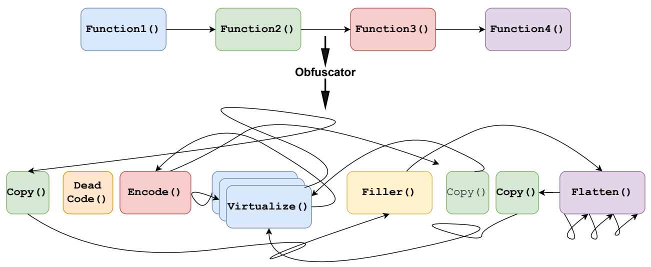

Figure 2: Visualization of Control Flow Obfuscation.

2.2 Obfuscation

Obfuscation is the act of rewriting or editing a program's source code to preserve its functionality while deliberately making it harder to understand or analyze [1]. Traditionally, it is used as a form of source code protection against reverse engineering. There are two major forms of obfuscation, control flow obfuscation and data obfuscation [1]. Control flow obfuscation is an attempt to mask a program's control flow graph (CFG). An example is demonstrated in Figure 2. The end goal of CFG obfuscation is to create a CFG that is radically different from the original, unobfuscated CFG. Data obfuscation is an attempt to make the data being manipulated harder to decipher. Data obfuscation is much more effective in high-level languages, where data is represented with variable names, or strings which contain explicit data. These two obfuscation methods are usually used together to make a program as incomprehensible as possible.

Obfuscation techniques are most commonly applied to the source code of a program by an obfuscator. Obfuscators are programs specifically designed to take a source file as input and apply a set of obfuscation techniques in an automated manner. They will then output the obfuscated source code. Obfuscators for higher-level languages are easier to find compared to low-level obfuscators and tend to be less primitive, being able to take advantage of more advanced obfuscation techniques.

To the best of our knowledge, *there are no known obfuscators for Wasm.* While obfuscated Wasm programs do exist, there is nothing to indicate that these programs were obfuscated at the Wasm level. Therefore, there are also no known techniques that have been specifically applied at the Wasm machine code level.

3 Problem Scope and Threat Model

3.1 Problem Scope

Obfuscation can be used both for benign and malicious purposes. Software developers can employ obfuscation to avoid the studying, analysis, tampering, and recycling of their software while it can be abused by attackers to bypass the detection and blacklisting methods [17]. This area of Wasm development and implementation, where Wasm code is obfuscated, is a completely undiscovered research area. Understanding the different ways that Wasm can be obfuscated, and the effect it has in terms of malware detection, is vital to understanding the future of Wasm development and browser-based cryptojacking programs. However, analyzing obfuscated Wasm applications is challenging because Wasm applications go through three distinct compilation and optimization steps before their execution on browsers. To fill this gap in the literature, we research and evaluate the feasibility of Wasm obfuscation.

3.2 Threat Model

WebAssembly has become a popular target for cryptojacking authors due to its isolation within the user's web browser while retaining access and control over computational resources. Because the Wasm program is run within the browser, there are no red flags for a traditional antivirus to catch, as there are no system calls, native compilation or execution, etc. As a result of the popularity of Wasm cryptojacking malware, there has been a flood of successful anti-cryptojacking extensions and detectors. To bypass these security measures, malware authors will have to begin to disguise their code. They can plug their source code into an obfuscator, and receive an obfuscated equivalent of their program, ready to be deployed without detection.

An adversary capable of successful obfuscation can theoretically have full access to browser functionality through the WebAssembly and the JavaScript API, meaning attackers can do far more than just ordinary cryptojacking. Attacker capabilities are only checked by the permissions that the user grants to the browser. An attacker could virtually acquire whole data of a user, employ ransomware, or even perform remote code execution, granting further access to their system [8].

4 Analysis and Evaluation of Obfuscation Techniques for WebAssembly

In this paper, we perform the first look at code obfuscation for WebAssembly and test the effectiveness of high-level obfuscation. To test the efficacy of high-level obfuscation on Wasm, we chose several obfuscation techniques and applied them to C source code of various Wasm web applications. In this section, we describe the obfuscation techniques we selected, our evaluations, and the results of the evaluations.

4.1 Obfuscation Techniques

Currently, to the best of our knowledge, *there are no Wasm obfuscators*, so pursuing machine code obfuscation is not practical, and it would most likely be too primitive to be effective. Since Wasm was designed to be a compilation target for high-level languages like C++ or Rust [10], obfuscation would most likely occur at the highest level. Additionally, more advanced obfuscation techniques are only available at a high level.

For these reasons, we decided to apply obfuscation in high-level source-code, compile the obfuscated code to Wasm, and evaluate the effect of obfuscation. Specifically, we chose to perform obfuscation in C source-code of Wasm web applications. For this aim, we used Tigress, a free to use C obfuscator that has been widely used in the literature [4]. We researched the obfuscation techniques and selected a set of techniques to obfuscate a dataset of C-based Wasm web applications that include both benign and malicious samples. Each technique was selected based on its proven effectiveness in the literature for traditional binaries, as there is no existing discussion of any obfuscation technique's effectiveness for Wasm binaries. Particularly, we apply the following obfuscation techniques:

- **Virtualize:** Each function in the program is turned into an interpreter which has its own unique bytecode. When executed, the program will interpret each bytecode, creating a virtual machine that then executes the instructions held within the bytecode, representing the original function.
- **Flatten:** The CFG of a program is flattened, creating an unstructured control flow [7].
- **Opaque Predicates:** An opaque predicate is a condition or branch (predicate) that is evaluated as a Boolean value. The outcome of this predicate is predetermined by the programmer but unknown to anyone else (opaque). When this predefined value is reached, the original code is executed.
- **Function Copies:** Making copies of an existing function. This results in unique function calls where they may have originally the same. This can cause the jump table to be allocated differently when compiled, making jump table references unnecessarily complex.
- **Taint Disruption:** To disrupt dynamic analyses tools that attempt to use taint analysis, the code is modified to not use a conventional data flow for variable copying but instead uses the control flow itself [16].
- **Alias Disruption:** To disrupt static analysis methods that use alias analysis, all direct function calls are replaced with indirect ones.
- **Encode Literals:** A conversion of static data to procedural data - both integer and string literals are either generated by function or replaced with an expression that at runtime is equivalent to the initial unobfuscated data.

4.2 Evaluation Setup

Wasm Application Category	Sample #	Portion of Dataset
Games	6	40.0%
Libraries	4	26.6%
Applications / Custom / Testing	3	20.0%
Cryptojackers	2	13.3%
Total	15	100%

Table 1: Sample counts and relative distribution of programs within the dataset.

4.2.1 Dataset. To perform the set of experiments in assessing the effectiveness of obfuscation techniques, we built a dataset of Wasm web applications. Our dataset is comprised of unique open-source Wasm applications that are written in C language. Each of the Wasm applications can fall under one of four categories: Games, Applications, Cryptojackers, or Libraries. The breakdown of the dataset is detailed in Table 1. Since the obfuscator requires C source files as its input, we needed to access the source code of the Wasm programs that would make up our dataset, giving us a smaller dataset, but still diverse enough to let us analyze the effects of obfuscation on different categories of Wasm program. Although we obtained the comprehensive cryptojacking dataset of MINOS [13], we could not incorporate the all samples into our dataset due to limitations of the Tigress obfuscator. We specifically chose Wasm programs that are used for web and mobile development, as these are the common uses of Wasm and can be found in the wild.

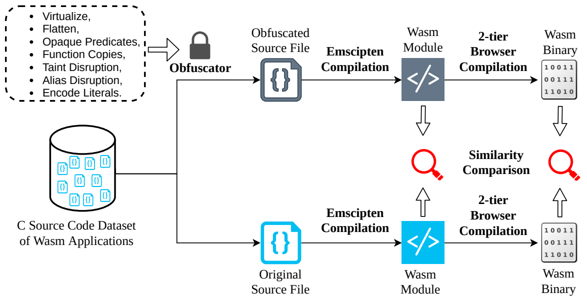

Figure 3: The pipeline used to obfuscate and analyze the similarity of obfuscated and original Wasm applications in two stages: after Emscripten compilation and after the 2-tier compilations of browsers.

4.2.2 The Evaluation Pipeline. C source code of each web application in the dataset is fed individually into our pipeline. We begin by taking the C source files and passing them into the obfuscator, applying an obfuscation technique. For every obfuscation technique listed in Section 4.1, a new obfuscated source file is created. Both the resulting obfuscated source file, along with the original one, are then compiled to Wasm modules using Emscripten, and analyzed to check the Cosine Similarity between the two. We then run these Wasm modules in a browser (e.g., Chrome). Before running the Wasm modules, the browser performs a 2-tier compilation on Wasm modules and optimizes them as explained in Section 2.1. Here, we extract the resulting Wasm binaries both for obfuscated and unobfuscated web applications and check their similarity for the second time. This cycle, as shown in Figure 3, is repeated for every program in our dataset.

4.2.3 Evaluation Criteria. The success of an obfuscation technique is evaluated at two different points in the compilation process. The first is done at the Wasm-level, comparing the obfuscated Wasm module to its unobfuscated counterpart. The second comparison is done at the binary level, comparing the binaries of the obfuscated and unobfuscated programs against each other. These binaries are the result of the Wasm module being instantiated in the browser by JavaScript, triggering a two-tier compilation process (in our case, Chrome's V8 engine).

At either stage, the process for comparing is identical - our goal to find their similarity. To quantify the similarity between the unobfuscated and obfuscated modules, we use n-gram analysis since it has been shown effective at analyzing similarity between samples of fuzzy data [21]. We extract n-grams from the machine code by partitioning the code into sequences of length n. Using n-grams allows us to deal with the noisy data presented in Wasm modules. These n-grams are then mapped onto vectors based on their Term Frequency Inverse Document Frequency (TF-IDF). We calculate the similarity score between the unobfuscated and obfuscated samples using Cosine Similarity formula: $\cos(\theta) = (\mathbf{A} \cdot \mathbf{B})/(\|\mathbf{A}\|\|\mathbf{B}\|) = (\sum_{i=1}^{n} A_i B_i)/(\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2})$. Cosine similarity is measured on a scale of 0 to 1, with 1 meaning the two samples are identical. The lower the Cosine similarity value, the more dissimilar the two samples are, which means the obfuscation is more effective, thereby successful. Cosine similarity gives us the best metric to do so by checking our n-grams in-context, meaning what code precedes and succeeds it, granting us a better metric for measuring stack-machine code, where order is extremely important.

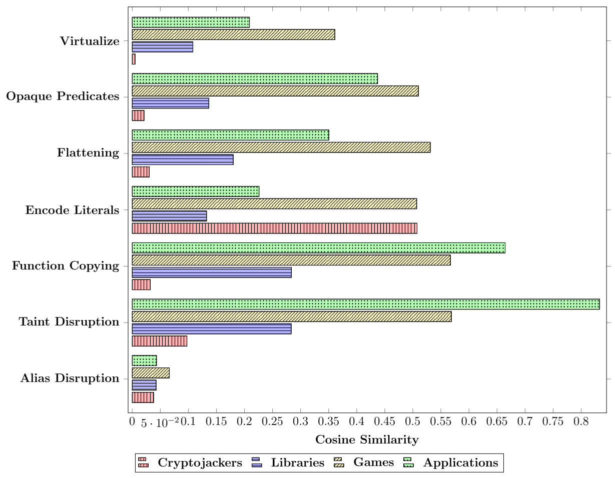

Figure 4: Average Cosine similarities of each obfuscation technique for each program category.

4.3 Results

Figure 4 is a visualization of average Cosine similarities of each obfuscation technique for each program category. The smaller the bar is, the less similar the resulting binaries are, and therefore the more effective the technique is. During the experiments, we found that once the Wasm modules are instantiated in the browser, there is a 1-to-1 correlation between the Cosine similarities of the Wasm modules and Wasm binaries pulled from the browser while it is running. Therefore, the data represented in Figure 4 is representative of both the Wasm binaries running in the browser, and the Wasm module before being instantiated.

Almost every single obfuscation technique, regardless of program type, resulted in a Cosine similarity score less than 0.5, with the exception of encoding literals for cryptojackers. This means that almost every obfuscated program had binaries that appeared distinct from their original unobfuscated counterpart, e.g. the program was effectively obfuscated. This serves as general proof that obfuscation at a high level can result in an obfuscated Wasm binary. The data demonstrates that Alias Disruption is the most effective obfuscation technique on average overall, but that does not hold true when looking for the best technique for each specific program type. For example, Virtualization is more effective than Alias Disruption for cryptojackers, demonstrating Alias Analysis is not the best in every case.

Findings: The distribution of the data in our results illustrates that there is no one "best" technique. While successful obfuscation is present in all programs, not all have been obfuscated to the same degree of success. The efficacy of each technique is dependent on the content of the program being obfuscated.

Application Category	Application Name	Unobfuscated	Alias Disruption	Taint Disruption	Function Copying	Encoding Literals	Flattening	Opaque Predicates	Virtualize
Application	BF Interpreter	✗	✗	✗	✗	✗	✗	✓	✓
Application	Conway Life	✗	✗	✗	✗	✗	✗	✗	✗
Application	Spiro	✗	✗	✗	✗	✗	✓	✗	✗
Cryptojacker	Garliccoin	✓	✗	✗	✗	✗	✓	✗	✗
Cryptojacker	kNight	✓	✗	✗	✗	✗	✗	✗	✗
Game	2048	✗	✗	✗	✗	✗	✗	✗	✗
Game	Breakout	✗	✗	✗	✗	✗	✗	✗	✗
Game	Circle Collide	✗	✗	✗	✗	✗	✗	✗	✗
Game	Dave	✗	✗	✗	✗	✗	✗	✗	✗
Game	Interplanetary	✗	✗	✗	✗	✗	✗	✓	✗
Game	Asteroids	✗	✗	✗	✗	✗	✗	✗	✗
Library	Galaxy sim	✗	✗	✗	✗	✗	✗	✗	✗
Library	OpenGL	✗	✗	✗	✗	✗	✗	✗	✗
Library	Topological ML	✗	✗	✗	✗	✗	✓	✓	✗
Library	Tracing ML	✗	✗	✗	✗	✗	✓	✓	✗

Table 2: Results of MINOS cryptojacking detector (✓: MINOS detected sample as malware, ✗: MINOS detected sample as benign. red : false positive and false negative results e.g. MINOS was incorrect, green : true positive and true negative results e.g MINOS was correct.).

Moreover, we also observed that compared libraries and cryptominers, applications and games tend to have more complex control flows because of more external dependencies such as images or UI components. These attributes may act as roadblocks for obfuscation, as with the exception of encoding literals, applications and games were consistently the least obfuscated. Dependencies like images is harder to obfuscate, as images themselves are not a part of the control flow, so the importing and loading the images can often seem to survive the obfuscation and remain decipherable. The opposite of this is also true; the lack of external dependencies seems to lead to a more fully obfuscated program.

Finally, we also found that for every type of program, at least one obfuscation technique was demonstrated to be successful in outputting an obfuscated program. It is worth noting that these were tests of singular obfuscation techniques being applied independently, meaning the obfuscation can be furthered by stacking multiple techniques and applying them together to the same program. Because of these results, we have arrived at the finding that obfuscation at the C-level can result in a successfully obfuscated Wasm program if the correct technique is applied.

4.4 Effects of Obfuscated Cryptojacking Malware on MINOS

In this section, we feed the obfuscated benign and malicious Wasm samples to a state-of-the-art cryptojacking detection system named MINOS [12] and evaluate how MINOS classifies the obfuscated samples. MINOS [12] is a novel lightweight defense system that detects malicious Wasm binaries based on their gray-scale image representations. It uses a Convolutional Neural Network (CNN) based classifier to classify a gray-scale image of a Wasm binary as malicious or benign. The performance analysis of MINOS showed that MINOS is 99% accurate in detecting Wasm cryptojackers.

4.4.1 Test Environment. To test the effect of Wasm obfuscation with MINOS, we first apply each obfuscation technique listed in Section 4.1 to each of the Wasm binaries from our dataset presented in Table 1. This gives us 120 samples (including both unobfuscated and obfuscated binaries) to test MINOS with. As explained in Section 4.4.2, although we obtained the comprehensive cryptojacking dataset of MINOS we were not able to employ them in this study. After obfuscating our dataset, we run MINOS on these unobfuscated and obfuscated versions of the Wasm binaries and analyzed the results.

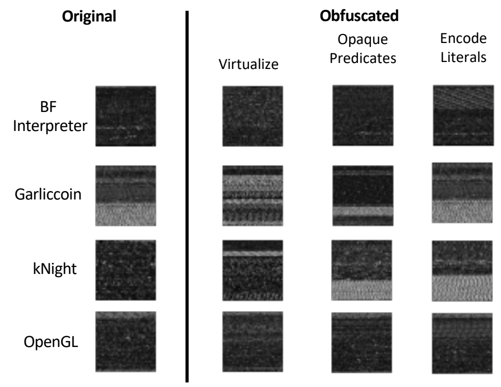

Figure 5: Grayscale images of Wasm binaries before and after obfuscation.

4.4.2 Challenges with Obfuscating Cryptojacking Malware. We could only test MINOS with a limited number of malicious samples due to limitations of the Tigress obfuscator [4] because most of the cryptojacking source code found in the wild is is written in C++, or uses libraries that are not supported with Tigress [4]. This limited us to two samples, which were written in C, and did not include unsupported libraries. However, since the cryptojackers obfuscated are derivatives of the CryptoNight webminer [19], the results can be generalized for most CryptoNight and Monero miners [5].

4.4.3 Results. We present our results in Table 2. Our results show that MINOS is not able to detect any obfuscated cryptojacker samples for six obfuscation techniques while it can only detect one of two cryptojacker samples for the flattening obfuscation technique. We were successful in effectively obfuscating malware, with a success rate of 93%, with only one obfuscated cryptojacker sample being detected by MINOS. Our obfuscated benign programs additionally created false positives, with a false positive rate of 70% i.e., 70% of all samples categorized as malware (including the unobfuscated ones) were actually benign.

Visually, in Figure 5, it is clear that the binaries look significantly different after obfuscation. In the case of cryptominers, the difference between what MINOS was looking for and obfuscated binaries are evident. In the case of BF Interpreter, the visible changes are slightly less obvious when viewed by a human, but evident to MINOS, demonstrated by the BF Interpreter obfuscated with opaque predicates, which was detected as malware, despite the program itself being benign.

These results gave us two key findings. The first is that MINOS' inability to effectively detect obfuscated cryptojacking samples proves obfuscation of Wasm binaries can generally be successful for malware in the wild, as our samples successfully bypassed static analysis. The second finding is that false positives created by the obfuscated benign samples render MINOS completely ineffective, as it now detects some obfuscated benign programs as malicious, and most obfuscated malicious programs as benign.

5 Related Work

There are numerous studies in the literature examining obfuscation techniques and their effectiveness as a means against static analysis methods [2, 17]. Musch et al. presented the first study on the prevalence of Wasm [11]. Along with a plethora of static analysis based cryptojacking detectors [3, 6, 12], Wasm analysis tools are becoming more prevalent in the literature [9]. Lehman et al. created a framework to dynamically analyze Wasm programs, allowing for a multitude of analysis methods like taint analysis or memory access tracing [9]. Lehman et al. also presented several security vulnerabilities of WebAssembly, in which they show the exploitation of the usage of memory unsafe source languages and attacks that can be contained in Wasm binaries through their undefined behavior [8]. Romano et al. presented a method to obfuscate JS malware using Wasm, splitting a program between the JS layer and the Wasm layer, making the program harder to analyze [15]. Their study resulted in obfuscated JavaScript malware, but their results cannot be extended to Wasm malware.

Differences from existing work: Discussion of the effectiveness of obfuscation techniques in the literature is limited to traditional programs and executables. In the context of Wasm obfuscation, Wasm has been utilized to obfuscate JS code [15], but the obfuscation of Wasm code itself has not been explored in the literature. Even though Wasm-based cryptojacking malware detection is widely discussed in the literature, there is no prior study investigating the feasibility of obfuscation for WebAssembly to bypass current malware detection methods. We additionally did not create any novel Wasm malware or Wasm-based attacks, but rather investigate how current Wasm malware can be obfuscated and disguised to be more effective by being less detectable.

6 Conclusion

In this paper, we presented the first analysis and evaluation of obfuscation for WebAssembly. Our results have shown that obfuscation at the high level can be effective if the correct techniques are applied, which allows programmers to create Wasm programs with an effective deterrent against decompilation or reverse engineering. We also found that obfuscated Wasm malware can bypass status quo detection methods. We believe this study is important to show a potentially unexplored attack vector before it is used by the malware authors in the wild. Consequently, creating analysis methods that cannot be circumvented by obfuscation techniques will be necessary to curb the potential influx of obfuscated Wasm malware.

Acknowledgment

This work was supported by the U.S. National Science Foundation (Award: NSF-CAREER CNS-1453647 and 1757761). The views expressed are those of the authors only, not of the funding agencies.

References

[1] Arini Balakrishnan and Chloe Schulze. 2005. Code obfuscation literature survey. *CS701 Construction of compilers* 19 (2005).

[2] Chandan Kumar Behera and D Lalitha Bhaskari. 2015. Different obfuscation techniques for code protection. *Procedia Computer Science* 70 (2015), 757–763.

[3] Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*. 3112–3118.

[4] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. 2022. The Tigress Obfuscator. https://tigress.wtf. [Online; accessed 4-April-2022].

[5] Monero Documentation. 2013. CryptoNight. https://monerodocs.org/proof-of-work/cryptonight/. accessed: 2022-02-12.

[6] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1714–1730.

[7] Tımea László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30, 1 (2009), 3–19.

[8] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of webassembly. In *29th USENIX Sec. Symp.* 217–234.

[9] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 1045–1058.

[10] MDN. 2021. WebAssembly Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. [Online; accessed 10-November-2021].

[11] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[12] Faraz Naseem, Ahmet Aris, Leonardo Babun, Ege Tekiner, and A Selcuk Uluagac. 2021. Minos*: A lightweight real-time cryptojacking detection system. In *Network and Distributed Systems Security (NDSS) Symposium*. 21–25.

[13] F. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac. 2021. MINOS: A Lightweight Real-Time Cryptojacking Detection System. In *NDSS*.

[14] Harun Oz, Ahmet Aris, Albert Levi, and A. Selcuk Uluagac. 2022. A Survey on Ransomware: Evolution, Taxonomy, and Defense Solutions. *ACM Comput. Surv.* (jan 2022). https://doi.org/10.1145/3514229

[15] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*.

[16] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. 2013. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices.. In *SECRYPT*, Vol. 96435.

[17] S. Schrittwieser and S. Katzenbeisser. 2011. Code obfuscation against static and dynamic reverse engineering. In *Int. workshop on information hiding*. Springer.

[18] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk. 2021. In-Browser Cryptomining for Good: An Untold Story. In *2021 IEEE Int. Conf. on Decentralized Applications and Infrastructures (DAPPS)*. 20–29.

[19] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk. 2021. SoK: Cryptojacking Malware. In *2021 IEEE European Symposium on Security and Privacy*.

[20] Conrad Watt. 2018. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN Int. Conf. on certified programs and proofs*. 53–65.

[21] Jun Ye. 2011. Cosine similarity measures for intuitionistic fuzzy sets and their applications. *Mathematical and computer modelling* 53, 1-2 (2011), 91–97.