Cryptic Bytes: WebAssembly Obfuscation for Evading Cryptojacking Detection

A Preprint

Håkon Harnes
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
haakaha@alumni.ntnu.no

Donn Morrison
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
donn.morrison@ntnu.no

March 25, 2024

Abstract

WebAssembly has gained significant traction as a high-performance, secure, and portable compilation target for the Web and beyond. However, its growing adoption has also introduced new security challenges. One such threat is cryptojacking, where websites mine cryptocurrencies on visitors' devices without their knowledge or consent, often through the use of WebAssembly. While detection methods have been proposed, research on circumventing them remains limited. In this paper, we present the most comprehensive evaluation of code obfuscation techniques for WebAssembly to date, assessing their effectiveness, detectability, and overhead across multiple abstraction levels. We obfuscate a diverse set of applications, including utilities, games, and crypto miners, using state-of-the-art obfuscation tools like Tigress and wasm-mutate, as well as our novel tool, emcc-obf. Our findings suggest that obfuscation can effectively produce dissimilar WebAssembly binaries, with Tigress proving most effective, followed by emcc-obf and wasm-mutate. The impact on the resulting native code is also significant, although the V8 engine's TurboFan optimizer can reduce native code size by 30% on average. Notably, we find that obfuscation can successfully evade state-of-the-art cryptojacking detectors. Although obfuscation can introduce substantial performance overheads, we demonstrate how obfuscation can be used for evading detection with minimal overhead in real-world scenarios by strategically applying transformations. These insights are valuable for researchers, providing a foundation for developing more robust detection methods. Additionally, we make our dataset of over 20,000 obfuscated WebAssembly binaries and the emcc-obf tool publicly available to stimulate further research.

Keywords: WebAssembly, obfuscation, cybersecuity, cryptojacking

1 Introduction

In 2015, Mozilla, Microsoft, Google, and Apple announced they were working on WebAssembly, a low-level bytecode language for the Web. It allows high-level languages like C, C++, and Rust to be executed in the browser at near-native performance. In 2019, WebAssembly received formal recognition as a Web standard [74], marking a significant milestone as the first new language to gain native support on the Web alongside JavaScript in over two *decades*. Since then, it has gained widespread adoption by large corporations, like Google, Figma, and eBay, who have leveraged it to

improve performance and port once desktop-only applications to the Web [21, 26, 22]. Beyond the Web, WebAssembly has been extended to desktop applications [46], mobile devices [58], cloud computing [25], blockchain virtual machines (VMs) [78], Internet of Things (IoT) [40], embedded devices [66], and stand-alone runtimes [77].

However, the growing adoption of WebAssembly has also introduced new security challenges. The most prominent example of this is cryptojacking, an attack strategy that exploits a website visitor's hardware resources to mine cryptocurrencies without their knowledge or consent. The introduction of cryptocurrencies like Monero and VerusCoin, which can be mined using consumer-grade CPUs, has made cryptojacking a feasible attack vector [47]. Although cryptojacking was initially implemented using JavaScript, WebAssembly has been the preferred method in recent years due to its superior performance. The release of WebAssembly in 2017 was followed by a 459% increase in cryptojacking incidents in 2018 [2]. By 2019, over half of all websites containing WebAssembly used it for cryptojacking [51]. Reports from 2022 confirm the steady growth of cryptojacking, indicating that it remains a pervasive problem [37].

In response to the escalating threat of cryptojacking, numerous detection methods have been proposed. Some methods rely entirely on static analysis [53, 62, 63], while others use dynamic analysis to detect cryptojacking [76, 60, 36]. Static methods leverage signature matching, control flow graph analysis, and neural networks, while dynamic methods focus on behavioral characteristics like CPU and memory usage, network traffic, and JavaScript events.

Surprisingly, there is limited research on how these detection methods can be sidestepped. Obfuscation, the process of making a program harder to analyze through applying various code transformations, has proven to be an effective evasion strategy for malware detection [57, 49]. Although obfuscated WebAssembly binaries are a common occurrence on the Web [33], the subject of WebAssembly obfuscation has been scarcely explored. A short paper from 2021 touched upon this topic [9], and during our research, two papers emerged with a similarly narrow focus [14, 42], considering only a limited set of obfuscation and detection methods. This recent surge of interest underscores the timeliness and relevance of this issue and the evident gap in the literature.

The primary objective of this paper is to investigate and understand the application, effectiveness, and implications of code obfuscation for WebAssembly. We evaluate how well obfuscation can disguise the underlying nature of the code and evade cryptojacking detection. Moreover, we quantify the overhead introduced by obfuscation, both in terms of code size and hash rate, and assess whether the overhead is justifiable given the obfuscation advantages. Guided by these objectives, we investigate the following research questions:

- **RQ1 – Effectiveness**. How effective are the transformations at obfuscating WebAssembly, and how is the resulting native code affected?
- **RQ2 – Detectability**. How effective is obfuscation at evading state-of-the-art cryptojacking detectors, and which transformations are the most effective?
- **RQ3 – Overhead**. To what extent do the transformations contribute to overhead in terms of code size and hash rate?

We extend the current body of literature with the following contributions:

- We perform a comprehensive evaluation of code obfuscation for WebAssembly across a diverse dataset, applying obfuscation at multiple abstraction levels – a first in the literature.
- We investigate how obfuscation can disguise the underlying code and evade state-of-the-art cryptojacking detectors.
- We develop and release emcc-obf,[1] a novel obfuscation method for WebAssembly, and compare it with existing methods.
- We quantify the overhead introduced by obfuscation, considering factors like code size and hash rate. We provide a granular analysis of the impact on different CryptoNight variants, a novelty in the existing body of literature.

[1] https://github.com/HakonHarnes/emcc-obf

- We compile a dataset of over 20,000 obfuscated WebAssembly binaries spanning diverse use cases, including all prominent CryptoNight variants, serving as a significant resource for future studies.[2]

2 Background

In this section, we provide an overview of the key concepts and technologies relevant to our research. We begin by discussing WebAssembly, its characteristics, and its growing adoption across various platforms. We then explore the threat of cryptojacking and the analysis techniques used to detect it, focusing on static methods. Finally, we delve into the concept of obfuscation and its potential for evading detection.

2.1 WebAssembly

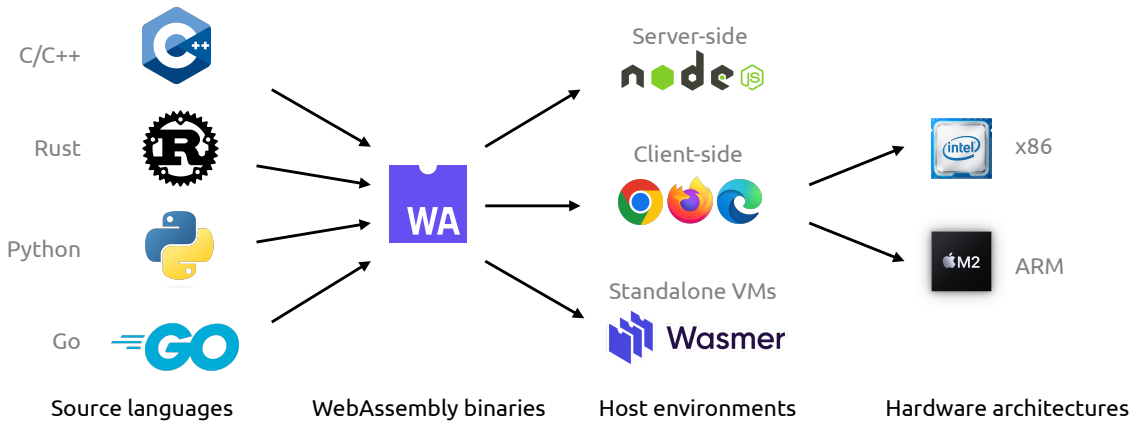


Figure 1: WebAssembly serves as an intermediate bytecode, bridging the gap between multiple source languages and host environments. The host environments compile the WebAssembly binaries into native code for underlying hardware architectures.

Overview. WebAssembly is a low-level bytecode language designed for the Web. It acts as an intermediate step, enabling interoperability between source languages and hardware architectures, as depicted in Figure 1. As a compilation target, WebAssembly provides a universal language into which different types of source code can be compiled and subsequently executed in various host environments. For example, programs written in Rust (a source language) can be compiled to WebAssembly and run in a browser using the V8 engine (a host environment), which then compiles the WebAssembly code into native code specific to the underlying hardware architecture.

Modules. WebAssembly modules are the fundamental units for deployment, loading, and compilation. These modules contain definitions for various components, including types, functions, tables, memories, and globals. Each function defined within the module is associated with a corresponding function body in the code section. The module can export its components, allowing the host environment to import and utilize them as needed.

Text and Binary Format. WebAssembly modules can be represented in two formats; the binary format (`wasm`) and the human-readable text format (`wat`). The binary format is designed to be compact, enabling efficient network transmission and parsing, and is typically generated by compilers and instantiated in runtimes. The human-readable text format serves a different purpose, being designed for debugging, testing, and occasional manual editing, akin to native assembly languages. It is possible to convert between these formats using tools like `wasm2wat` and `wat2wasm`, which are part of the WebAssembly Binary Toolkit (WABT).

Stack and Variables. WebAssembly modules are executed on a stack-based virtual machine (VM). In this system, instructions operate by popping their inputs from and pushing their results to an implicit evaluation stack. There

[2] https://github.com/HakonHarnes/wasm-obf/releases/tag/v1.0

are no registers in this system; instead, values are stored in local or global variables. Global variables are visible to the entire module, while local variables are only accessible within the current function. The evaluation stack, local variables, and global variables are managed by the VM.

Host Environment. WebAssembly modules are executed within a host environment, which provides the necessary functionality for the module to perform actions such as file or network access. In a browser, the host environment is typically provided by the JavaScript engine, such as V8 or SpiderMonkey. The WebAssembly-JavaScript API allows WebAssembly exports to be wrapped in JavaScript functions, enabling them to be called from JavaScript code. Conversely, WebAssembly code can import and call JavaScript functions, enabling bidirectional communication between the two languages. Beyond the browser, WebAssembly modules can be executed in other host environments, including server-side environments like Node.js and stand-alone VMs like Wasmer. These host environments provide their own APIs for WebAssembly modules to interact with, giving them access platform-specific features and resources. For example, modules running on stand-alone VMs can leverage the WebAssembly System Interface (WASI) to interact with the file system.

Source Languages. WebAssembly's low-level nature makes it an ideal compilation target for systems languages like C, C++, and Rust. Both the Clang and the Rust compilers have native support for WebAssembly, enabling direct generation of WebAssembly modules [17, 27]. Moreover, Emscripten, a toolchain built on Clang and LLVM, is capable of porting C and C++ code to the Web using WebAssembly [23]. In addition to compiling C and C++ code to WebAssembly, Emscripten also produces the corresponding JavaScript "glue" code. The JavaScript code is responsible for module instantiation and providing the necessary functionality for it to interact with the host environment. For example, it pipes the output from the `printf` function to the browser's console. Similarly, wasm-bindgen enables high-level interactions between Rust and JavaScript for use in the browser [64]. Even garbage-collected languages like C#, Python, and more recently Kotlin and Dart, can be compiled to WebAssembly.

V8 Compilation. The V8 engine, which is used in Google Chrome, employs a two-tiered WebAssembly compilation pipeline [31]. Initially, the baseline Liftoff compiler lazily compiles functions when they are first called. That is to say, if a function is never called, it is never compiled to native code. Liftoff iterates over the WebAssembly code just once and immediately emits native code, which allows for fast code generation, albeit with a limited set of optimizations. Once Liftoff compilation is done, the native code is registered with the WebAssembly module for immediate future use. For functions that are frequently invoked, termed "hot" functions, the V8 engine uses its optimizing TurboFan compiler. Unlike Liftoff, TurboFan is a multi-pass compiler that constructs multiple internal representations of the code during compilation, enabling advanced optimizations. When a function is deemed hot, TurboFan is triggered to recompile and optimize it in the background. The resulting optimized native code then replaces the existing Liftoff-generated code, ensuring increased performance for all subsequent calls to that function.

2.2 Cryptojacking

Cryptojacking[3] is a type of attack that involves using the hardware resources of a website visitor to mine cryptocurrencies without their knowledge or consent. Such an approach has become feasible with the advent of cryptocurrencies like Monero [48] and VerusCoin [72], which can be mined using consumer-grade CPUs [47]. Cryptojacking can be executed through self-hosted mining or by compromising web servers through software vulnerabilities or misconfigurations [71, 56], and then subsequently installing the mining scripts on the compromised web servers. Alternatively, mining scripts can be distributed through advertising platforms [15], compromised third-party libraries integrated within various websites [79], or through adversarial Docker images [16]. Interestingly, browser-based crypto mining has been used as an alternative income stream by organizations like UNICEF [38], although with user approval, differentiating it from adversarial cryptojacking.

Initially implemented with JavaScript only and popularized with the launch of CoinHive in 2017, WebAssembly has become the preferred method for cryptojacking in recent years due to its superior performance. The release of WebAssembly in 2017 was followed by a 459% increase in cryptojacking incidents in 2018 [2]. By 2019, more

[3] Also referred to as drive-by mining.

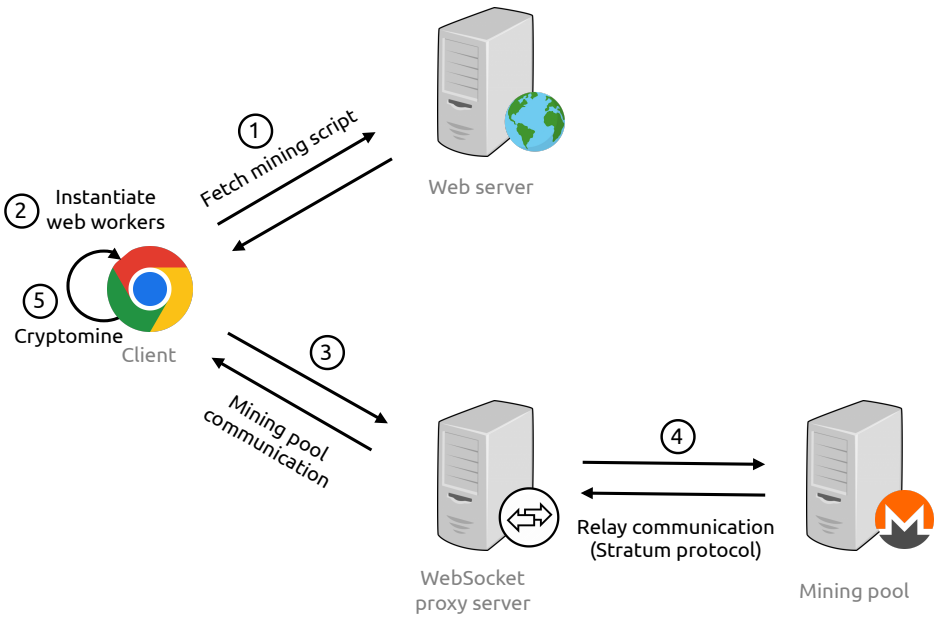


Figure 2: Cryptojacking process: The mining script is fetched from the web server, which instantiates the web workers and connects to the WebSocket proxy server. The proxy server relays the communication back to the mining pool.

than half of all websites containing WebAssembly used it for cryptojacking [51]. Although CoinHive shut down in 2019 [59], reports from the first three quarters of 2022 confirm the steady growth of cryptojacking [37], indicating that it remains a pervasive problem.

Today, cryptojacking is implemented with a combination of JavaScript and WebAssembly. The JavaScript code is responsible for coordinating the mining process by communicating with the mining pool, while WebAssembly is used to calculate the hashes. The process is depicted in Figure 2. First, (1) when a user visits a website, the JavaScript and WebAssembly code is fetched from the web server. Then, (2) the JavaScript code checks how many CPU cores are available on the host machine and spawns web workers, one for each available core. Each web worker instantiates the WebAssembly module and (3) connects to the WebSocket proxy server. The WebSocket proxy server (4) connects to the mining pool and retrieves the mining job. Lastly, the communication is relayed back to the web workers, which (5) calculate the hashes and send the results back to the mining pool through the proxy server. The communication between the web workers and the mining pool is usually implemented using the Stratum protocol [67].

2.3 Analysis Techniques

Analysis techniques can be used in a variety of ways, not only for detecting malware but also for vulnerability identification [43], performance optimization [70], and for understanding and debugging code [55]. Although different in their objective, these techniques often follow similar methodologies. As such, the analysis techniques, including those evaluated in this paper, can be classified along the following dimensions:

Static and Dynamic. Static analysis examines the program without executing it, enabling fast, real-time detection. However, its precision often falls short of dynamic analysis due to its reliance on approximating the actual runtime behavior [20]. Moreover, obfuscation has been found to be effective against static analysis [50]. In contrast, dynamic analysis observes the behavior of the program during execution, typically providing higher accuracy than static analysis [20]. Although advantageous, it can be resource-intensive and time-consuming, and it may struggle to explore all possible program behaviors due to a large or potentially infinite search space [80]. Although generally

more resilient to obfuscation than static methods, the presence of obfuscation can increase time consumption and still lead to a considerable false positive rate [8].

Rule-Based and Machine Learning. Rule-based methods evaluate programs according to a set of predefined rules or patterns, offering transparent and comprehensible decision-making. These methods can provide high accuracy, provided the rules are precisely formulated. However, their effectiveness can be undermined by novel threats or obfuscation techniques that circumvent established rules [49]. On the other end of the spectrum, machine learning models are trained on large datasets, where each entry is annotated with the expected output. After training, the models apply the learned patterns to predict outcomes on previously unseen instances. Although machine learning methods generally handle obfuscation better than rule-based methods, they are still vulnerable to adversarial attacks. These attacks involve making subtle modifications to input data with the intent to deceive the model, thereby reducing its accuracy [69, 30, 42].

2.3.1 Detecting Cryptojacking

To address the increasing threat of cryptojacking, a number of analysis techniques have been proposed. The literature presents methods based on both static [53, 62, 63] and dynamic analysis [10, 60, 36], using both rule-based and machine learning-based approaches. Static methods use a wide range of techniques, including signature matching, control flow graph (CFG) analysis, and neural networks. Dynamic methods rely on behavioral characteristics such as CPU and memory usage, network traffic, and JavaScript events.

In this paper, we focus on static analysis techniques for several reasons. Primarily, dynamic analysis proves impractical due to its substantial overhead, ranging from 40% to 100% [60, 76]. This would likely degrade the user experience and, therefore, it is not a viable option for real-world applications. Although dynamic analysis can be used for offline analysis, it is not a feasible solution either, as the blacklists need to be updated frequently and can be circumvented through diversification [14]. Moreover, several dynamic methods depend on platform or browser-centric features, such as the MessageLoop event [36], or Chrome debugging features [60], rendering it difficult to implement in real-world applications. Lastly, empirical evidence demonstrates that static methods are just as effective as dynamic methods in combating cryptojacking, with F_1 scores ranging from 0.95 [53] to 1.00 [63], compared to dynamic methods scoring between 0.96 [60] and 0.98 [76].

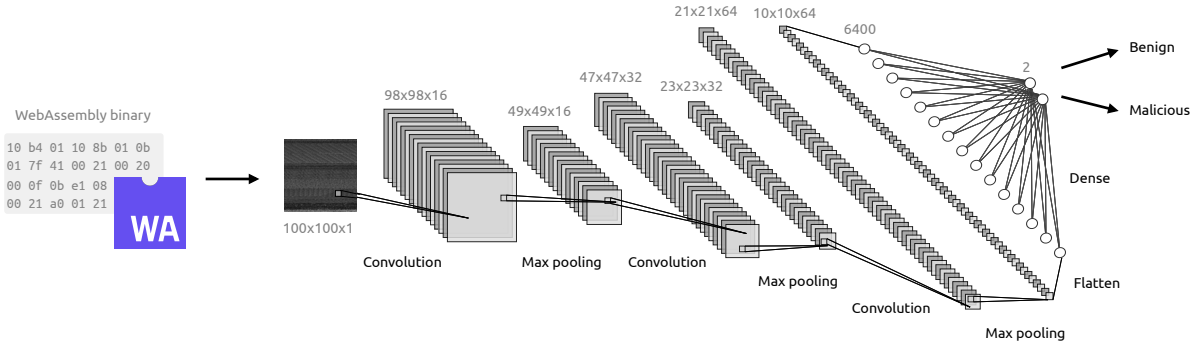


Figure 3: Overview of MINOS: The WebAssembly binary is converted to a grayscale image and fed to the MINOS network. The network predicts whether the binary is benign or malicious.

MINOS. MINOS [53] is a machine learning-based method that uses an image-based classification deep learning approach to identify cryptojacking. As illustrated in Figure 3, the WebAssembly binary is first converted into a 100x100 grayscale image. This image is then used as input to a convolutional neural network (CNN), which has been trained on a dataset of malicious and benign WebAssembly binaries. The CNN attempts to determine whether the

WebAssembly binary performs cryptojacking based on the patterns it observes in the grayscale image. The model was able to achieve an F_1 score of 0.95 with an average detection time of just 25.9 milliseconds.

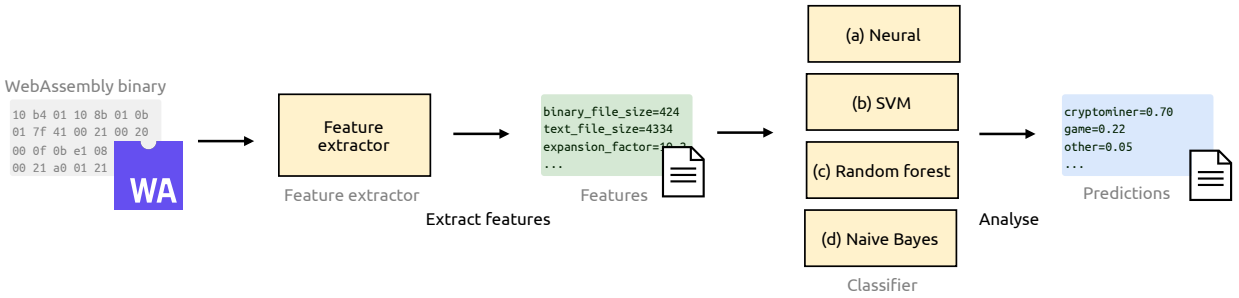


Figure 4: Overview of WASim: Features are extracted from the WebAssembly binaries and fed into a classifier. The classifier model is either: (a) Neural, (b) SVM, (c) Random forest, or (d) Naive Bayes. The classifier outputs a usage report containing the predictions.

WASim. WASim [62] is a machine learning-based method that extracts and analyzes a set of features for detecting cryptojacking. The procedure is depicted in Figure 4. First, the WebAssembly binary is converted from the binary format (`wasm`) to the text format (`wat`), which is then parsed to extract a set of features. Then, these features are used by the classifier models to predict the use case of the WebAssembly module, for example, classifying it as a crypto miner, game, or other application. The authors implemented several classifier models, including neural, support vector machine (SVM), random forest (RF), and naive Bayes, achieving accuracies of 91.6%, 87%, 82%, and 64%, respectively.

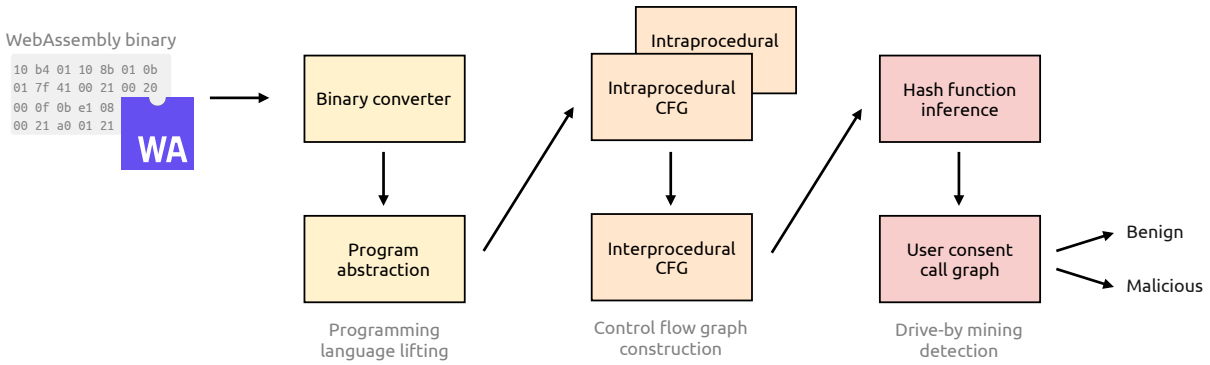


Figure 5: Overview of MinerRay: The WebAssembly binary is converted into a custom intermediate language, from which an interprocedural CFG is constructed. The control flow is then analyzed to detect cryptojacking, as well as for checking user consent.

MinerRay. MinerRay [63] is a rule-based method that analyzes the control flow of the WebAssembly module to detect cryptojacking. As illustrated in Figure 5, the process is divided into three parts: Programming language lifting, CFG construction, and cryptojacking detection. For programming language lifting, it uses a set of abstraction rules to translate the WebAssembly opcodes to a custom intermediate representation. Given the intermediate representation, an intraprocedural CFG is constructed for each function. These intraprocedural CFGs are then linked together to create an interprocedural CFG that represents the entire program. MinerRay uses this interprocedural CFG to identify potential hashing algorithms by analyzing the control flow of the program and looking for patterns that match the semantics of hashing functions. To determine whether the user is informed about crypto mining, MinerRay employs a dynamic approach that explores `onclick` events of HTML objects. It checks if the `onlick` events can

trigger WebAssembly APIs, such as `WebAssembly.instantiate`. MinerRay achieved an F_1 score of 0.99 with an average detection time of 1.9 seconds.

VirusTotal. VirusTotal [73] uses an extensive set of antivirus scanners to detect malware. Of the 70 antivirus scanners, 59 are able to scan WebAssembly binaries, including prominent ones such as AVG, Avast, and McAfee. Each antivirus scanner integrated within the system incorporates distinct heuristic methods tailored for the detection of specific types of malware. In the literature, it has been used to detect cryptojacking [33, 76, 14].

2.4 Obfuscation

Obfuscation involves transforming a given program into one that is syntactically different but semantically equivalent [52]. It has been used for a variety of purposes, including prevention of reverse engineering [11], prevention of software modification [28], hiding static data [35], and for malware evasion [49, 57]. The obfuscation process involves the application of a set of code transformations, which can be formally defined as follows:

Definition 1 (Transformation). Let $P \xrightarrow{T} P'$ be a transformation of a program P to a program P'. The transformation is an obfuscating transformation if P and P' have the same observable behaviour but are different syntactically [19].

The transformations applied in this paper, aligned with the taxonomy proposed by Collberg et al. [19], are categorized as follows:

- **Control Obfuscation**. Involves manipulating a program's control flow through techniques like adding false conditional statements and altering loop conditions.
- **Data Obfuscation**. Alters the representation of data structures and values within the code, utilizing methods like splitting data structures and using complex encodings, while maintaining their semantic meaning.
- **Preventive Transformations**. Designed to thwart specific types of code analysis or reverse engineering tools, incorporating techniques like anti-debugging, anti-tampering, and encoding mechanisms.
- **Layout Obfuscation**. Modifies the arrangement of code elements to disrupt readability and traceability, including the removal of source code formatting and reordering of instruction sequences.

2.4.1 Diversification

Diversification is a technique that generates multiple distinct yet semantically equivalent versions of a program, with an emphasis on enhancing resilience against attacks rather than obscuring the analysis or understanding of the code. Unlike obfusation, which primarily aims to obscure code analysis, diversification focuses on creating multiple program variants, ensuring that a single exploit cannot compromise all instances. While diversification can inadvertently lead to code obfuscation, its primary goal is to improve security through variability. In the context of this discussion, the terms *mutations* and *transformations* are used interchangeably to refer to either obfuscating transformations or diversifying mutations.

2.4.2 Obfuscation Tools

There are no known obfuscation tools that operate at the WebAssembly level. However, the wasm-mutate [12] diversifier can be used to diversify WebAssembly binaries, which may inadvertently lead to obfuscation. Another option is to obfuscate code at a higher abstraction level, such as source code or LLVM bitcode, and then subsequently compile the obfuscated code to WebAssembly. In this paper we apply obfuscation at multiple abstraction levels, using the following tools:

- **Tigress**. Tigress [18] is a source-to-source obfuscator for C written in OCaml. It has been thoroughly evaluated in the literature [9, 7], and has been found to be successful in evading cryptojacking detection [9].
- **OLLVM**. Obfuscator-LLVM (OLLVM) [34] is implemented as middle-end passes in the LLVM compilation suite. It has been used for preventing reverse engineering [7, 39], but not for evading cryptojacking detection. In this paper, we develop emcc-obf for applying OLLVM obfuscation to WebAssembly.

- **wasm-mutate**. wasm-mutate [12] is a wasm-to-wasm diversifier written in Rust. Wasm-mutate has been used for fuzzing and has been found to be successful in evading cryptojacking detection [3, 14].

2.5 Code Similarity

Traditional code similarity evaluation methods, like cyclomatic complexity [45], Halstead complexity measures [32], and lines of code, focus on structural aspects of code such as control flow, operator usage, and code size. However, these metrics are limited as they fail to capture the nuanced semantics of code and are vulnerable to superficial changes like code refactoring. These shortcomings are particularly evident in the context of WebAssembly, as it is a stack-based language where the order of instructions affects the semantics of the program, a factor these traditional metrics do not consider.

Sequence alignment methods take into account the order of instructions, making them well-suited for stack-based languages like WebAssembly. Among these methods, dynamic time warping (DTW) has proven to be more accurate than other sequence alignment algorithms [1], and it has been successfully used for aligning and comparing stack traces and WebAssembly binaries [5, 4]. The fundamental concept of DTW is to identify the optimal alignment between two sequences. The process aims to minimize the sum of absolute differences, commonly referred to as the *distance*, between corresponding elements within the sequences. DTW computes a cost matrix representing the pairwise distances between all possible pairs of points in the two sequences. the goal is to find a path through this cost matrix, the so-called *warping path*, which minimizes the total cumulative distance. To measure code similarity, we represent WebAssembly binaries as sequences of instructions and compare them using DTW.

3 Related Work

In this section, we review the existing literature on WebAssembly obfuscation and its application in evading cryptojacking detection. We discuss the recent studies that have explored the use of obfuscation techniques such as Tigress, wasm-mutate, and binary manipulation to disguise WebAssembly binaries and bypass state-of-the-art detectors.

Bhansali et al. [9] used Tigress to obfuscate C source code before compiling it to WebAssembly. They found they could successfully evade detection from MINOS. However, they did not ensure the WebAssembly binaries were still functional after obfuscation. The overhead caused by obfuscation was not measured, either. Cabrera Arteaga et al. [14] demonstrated how wasm-mutate can be utilized to diversify WebAssembly binaries, effectively evading both MINOS and VirusTotal with minimal performance overhead. Loose et al. [42] proposed a novel technique that employs binary manipulation through instrumentation to incorporate adversarial examples into code sections within WebAssembly modules, successfully evading MINOS. However, they measured the performance overhead using a generic SHA256-hashing library instead of crypto mining binaries, which may limit the validity of their results.

In a similar vein, studies have turned to WebAssembly as a means of obfuscating JavaScript. Romano et al.[61] proposed Wobfuscator, a technique based on a set of code transformations that opportunistically translates specific parts of JavaScript code into WebAssembly. Similarly, Wang et al.[75] introduced JSPro, a tool that converts JavaScript into WebAssembly for obfuscation purposes. It is important to clarify that their objective diverges from ours; they seek to obfuscate JavaScript code by using WebAssembly, while we concentrate on obfuscating the WebAssembly code itself.

4 Methodology

In this section, we describe the methodology employed in our research. We start by detailing the experimental setup, including the system configuration and dataset used. We then discuss the implementation of the obfuscation methods, cryptojacking detectors, and the dynamic time warping algorithm used for comparing WebAssembly binaries. Finally, we define the evaluation metrics used to assess the effectiveness, detectability, and overhead of the obfuscation methods.

Category	Name	Description
Utilities	lcs	Calculates the longest common subsequence of two strings
	tree	Lists folder contents in a tree format
	wgsim	Whole-genome simulation tool for generating sequencing reads
	seqtk	Toolkit for processing sequences in FASTA/Q formats
	smith-waterman	Algorithm for local sequence alignment of protein and DNA
	needleman-wunsch	Algorithm for global sequence alignment of protein and DNA
Games	pong	Arcade game simulating table tennis
	snake	Arcade game where the player controls a snake
	f1-race	Racing game that simulates Formula 1 car races
	breakout	Arcade game where you control a paddle to hit a bouncing ball
	game-of-life	Cellular automaton simulating the evolution of cells
	wasm-asteroids	Arcade game where you control a spaceship to shoot asteroids
Crypto miners	cn-0	Variant 0 – Original CryptoNight algorithm
	cn-1	Variant 1 – Also known as Monero v7
	cn-2	Variant 2 – ASIC-resistant version
	cn-r	Variant 4 – Also known as CryptoNightR
	cn-lite-0	Lite variant 0 – cn-0 with half the memory and iterations
	cn-lite-1	Lite variant 1 – cn-1 with half the memory and iterations
	cn-lite-2	Lite variant 2 – cn-2 with half the memory and iterations
	cn-half	Half variant – cn-2 with half the iterations
	cn-rwz	Reduced work variant – cn-2 with a quarter the iterations
	cn-pico-trtl	Pico turtle variant – cn-2 with an eighth the memory and iterations

Table 1: Dataset used in this paper, spanning a wide range of categories, including utilities, games, and crypto miners.

4.1 Experimental Setup

The experiments were performed on a VM running Debian 10 with kernel 4.19.0-22, equipped with an AMD EPYC 7742 CPU running at 2.24 GHz. Although the VM had 64 CPU cores available, the experiments relating to cryptojacking were performed using only 4 cores to ensure that the results would be comparable with consumer-grade hardware and mobile devices, which typically have fewer cores. To guarantee the reproducibility of the experiments across various system configurations, the code has been containerized using Docker.

4.1.1 Dataset

Detailed in Table 1, the dataset used in this paper covers a broad spectrum of categories, including utilities, games, and crypto miners. All the applications are open-source projects written in C. The utilities were carefully selected to represent a wide range of functionality, featuring Linux utilities, sequence alignment algorithms, and simulations. The games were chosen to represent varying levels of complexity, from classical arcade games to cellular automaton simulations. The crypto miners contain the predominant versions of the CryptoNight algorithm and their less memory and computation-intensive variants,. Importantly, this provides extensive coverage of crypto mining malware, as several studies have found that in-the-wild cryptojacking implementations are all based on the CryptoNight algorithm [14, 33].

4.2 Implementation

The implementation, aligned with the research strategy depicted in Figure 6, is described in the following sections. Each application in the dataset undergoes obfuscation using Tigress, emcc-obf, and wasm-mutate, as specified in Section 4.2.1. After obfuscation, the original and obfuscated binaries are fed to the cryptojacking detectors, detailed

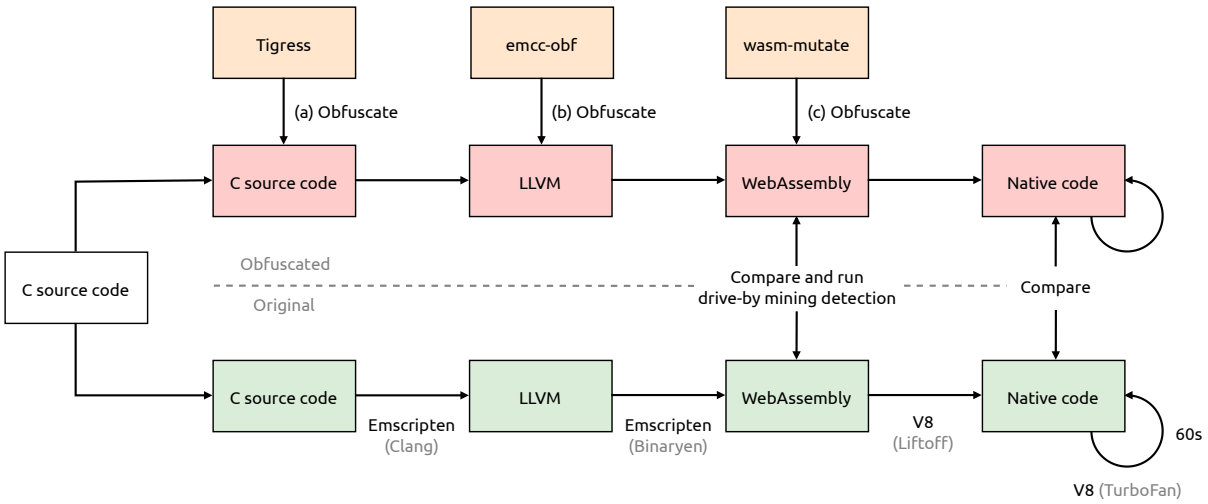


Figure 6: Overview of the research strategy: Each program is obfuscated using either (a) Tigress, (b) emcc-obf, or (c) wasm-mutate at the corresponding abstraction level, and finally compiled to WebAssembly. The WebAssembly binaries are then run through the cryptojacking detectors, instantiated in the browser to extract the native code, and compared with their non-obfuscated counterparts.

in Section 4.2.5. Then, the WebAssembly binaries are instantiated within the Chrome browser to extract the native code, as explained in Section 4.2.6. To measure the performance overhead of the crypto miners, we implement a cryptojacking client, web server, and WebSocket proxy server, as described in Section 4.2.7. Lastly, we implement the DTW algorithm as a means of comparing the WebAssembly binaries, detailed in Section 4.2.8. In the interest of transparency and reproducibility, the code used to conduct the experiments is publicly available on GitHub.[4]

4.2.1 Obfuscation

For obfuscation, we use a variety of obfuscation methods, each operating at a different abstraction level. We obfuscate the source code using Tigress, the LLVM bitcode using emcc-obf, and the WebAssembly code using wasm-mutate. In other words; we either apply obfuscation *before* compilation using Tigress, *during* compilation using emcc-obf, or *after* compilation using wasm-mutate.

The applications in Table 1 are obfuscated using Tigress, emcc-obf, and wasm-mutate, with the number of binaries generated for each application shown in Table 2. For Tigress and emcc-obf, eight transformations are applied per application, while for wasm-mutate, six transformations are applied per application. However, since wasm-mutate produced unsatisfactory results when applying individual transformations, and in an effort to replicate results from other studies [14], we also apply stacked transformations using wasm-mutate. Stacked transformations involve applying one transformation after the other, in succession, to each application in the dataset. To this end, we apply a sequence of 1000 random transformations to each application This process results in 22 WebAssembly binaries in the original dataset and a total of 22,484 binaries in the obfuscated dataset.

Since Tigress expects only one source file as input, we merge all the source files for each application into one file using the C intermediate language (CIL) [54]. Although only Tigress requires a single source file, we use the same merged source file for emcc-obf and wasm-mutate to ensure that the source code is identical for all obfuscation methods. To further ensure consistency, we compile all the WebAssembly binaries using the same Emscripten version, version 3.1.35. The applications were compiled with optimization turned off to ensure the compiler did not optimize away the obfuscation applied.

[4] https://github.com/HakonHarnes/wasm-obf

Category	Original	Tigress	emcc-obf	wasm-mutate	wasm-mutate (stacked)
Utilities	6	48	48	36	6000
Games	6	48	48	36	6000
Crypto miners	10	80	80	60	10,000
Sum	22	176	176	132	22,000

Table 2: Number of WebAssembly binaries in the original and obfuscated case.

To ensure the correctness of the obfuscated binaries, we invoke all of them in the browser and manually check that they are still functioning as intended. For the stacked wasm-mutate transformations, we perform this check every 100th iteration. In the case of crypto miners, we verify that the hashes reach the mining pool and are accepted by it as valid hashes. However, due to the mining network difficulty, we were only able to directly verify this for cn-r, cn-lite-0, and cn-pico-trtl, as the other variants could not solve and submit the hashes before receiving new jobs. To address this, we compare the first 100 hashes of the original and obfuscated binaries for all CryptoNight variants and ensure that they are identical. Through this process, we found that the applications in our dataset continued to function as intended after obfuscation.

4.2.2 Tigress

We use the latest version of Tigress, version 3.3.2, to obfuscate the source code of each application.

The following transformations were applied to the source code of each application:

- **Flattening**. Control obfuscation that transforms the control flow of an application into a flat hierarchy, thereby eliminating structured control flow.
- **Random Functions**. Control obfuscation that generates a unique random function. Random function calls are also inserted into the generated code for increased complexity.
- **Function Splitting**. Control obfuscation that splits a function into smaller sub-functions. This technique disguises the structure of the original function, thus complicating the process of code analysis.
- **Virtualization**. Control obfuscation that transforms a function into a specialized interpreter by constructing a unique bytecode. This technique involves the creation of a virtual instruction set architecture (ISA) and a bytecode program, with each function essentially executing as a self-contained VM.
- **Encode Arithmetic**. Data obfuscation that replaces integer arithmetic with more complicated but equivalent expressions using mixed boolean-arithmetic (MBA) [24].
- **Encode Literals**. Data obfuscation that replaces constant integers and strings with code that dynamically generates them at runtime. Specifically, it uses opaque expressions to substitute integers and replaces strings with functions that generate them at runtime.
- **Anti-Alias Analysis**. Preventive transformation that replaces all direct function calls with indirect ones to disrupt static analysis techniques that make use of alias analysis.
- **Anti-Taint Analysis**. Preventive transformation that replaces the conventional data flow used for variable copying with control flow instead, with the aim of disrupting dynamic analysis tools that make use of taint analysis.

4.2.3 emcc-obf

We build and release emcc-obf,[5] the first WebAssembly compiler with built-in obfuscation support. Emcc-obf is a modified version of the Emscripten compiler, one of the most widely-used WebAssembly compilers. The modifications

[5] https://github.com/HakonHarnes/emcc-obf

are based on OLLVM [34], which is no longer maintained. Instead, we use the Hikari obfuscator,[6] a maintained fork of OLLVM which is compatible with LLVM 16.0.0. We build emcc-obf using the Hikari-modified version of LLVM 16.0.0, and compatible Binaryen and Emscripten versions.

The following transformations were applied to the LLVM-bitcode of each application:

- **Control Flow Flattening**. Control obfuscation that flattens the control flow of the program, similar to that of the flattening transformation of Tigress.
- **Bogus Control Flow**. Control obfuscation that modifies the function call graph by inserting a new basic block preceding the original block. This new block includes an opaque predicate and executes a conditional jump to the original block.
- **Indirect Branches**. Control obfuscation that replaces branching instructions with indirect branching. This technique thwarts disassemblers' ability to accurately predict the complete control flow through static analysis.
- **Basic Block Splitting**. Control obfuscation that splits basic blocks, thereby breaking the structure by artificially increasing the number of basic blocks in a function.
- **Function Wrapper**. Control obfuscation that encapsulates each target function within a generated wrapper function, introducing an additional layer of indirection to hinder control flow analysis.
- **Substitute Instruction**. Data obfuscation that replaces arithmetic and boolean expressions with more complicated but equivalent instruction sequences. This is similar to the encode arithmetic transformation of Tigress.
- **Constants Encryption**. Data obfuscation that encrypts constant integer values using the XOR cipher. The encrypted values are decrypted at runtime to their original form. This complicates reverse engineering as an analyst cannot directly read the constant values from the static code.
- **String Encryption**. Data obfuscation that encrypts string values using the XOR cipher. The encrypted values are decrypted at runtime to their original form. Similar to constants encryption, but for strings.

4.2.4 wasm-mutate

We use the latest version ofwasm-tools,[7] version 1.0.33, which contains the wasm-mutate tool. We use the `–preserve-semantics` flag to ensure that only semantics-preserving transformations are applied.

The following transformations were applied to WebAssembly code of each application:

- **Code motion**. Control obfuscation that modifies the abstract syntax tree (AST) of a WebAssembly module by selectively applying a defined set of mutators, modifying the control flow or other aspects of the code while preserving its functionality.
- **Peephole**. Data obfuscation that applies random, localized modifications on portions of the WebAssembly module. This is achieved by generating a minimal data flow graph (DFG) from a selected operator, applying predetermined rewriting rules to this DFG, resulting in a subtly modified version of the original segment in the module.
- **Add Function**. Layout obfuscation that adds a function to the module.
- **Add Type**. Layout obfuscation that adds a type to the module.
- **Add Custom Section**. Layout obfuscation that adds a custom section to the module.
- **Remove Item**. Layout obfuscation that removes an item (e.g. function) from the module.

[6] https://github.com/61bcdefg/Hikari-LLVM15
[7] https://github.com/bytecodealliance/wasm-tools

4.2.5 Cryptojacking Detection

In order to identify cryptojacking, we implement the detection methods presented in the background section; namely MINOS, WASim, MinerRay, and VirusTotal. For MINOS, we use the reproduction by Cabrera et al. We use the publicly available implementation for WASim, although we encountered challenges due to out-of-date dependencies, which we then updated to their latest versions. We also use the public implementation of MinerRay, albeit faced with issues related to the JavaScript heap limit due to the path explosion problem. This led us to disable the function call linking for larger files as advised by the authors, although it may affect the detection rate. Moreover, we implemented a two-minute timeout delay to prevent indefinite execution. For VirusTotal, we use their API.

The MINOS reproduction by Cabrera et al., trained on 144 benign and 49 crypto mining binaries, resulted in a 0% detection accuracy for our dataset. To address this issue, we re-trained MINOS using the same dataset as Cabrera et al., augmenting it with the non-obfuscated binaries from our dataset. We did not re-train the other machine learning-based detection methods as they delivered higher accuracies.

4.2.6 Extracting the Native Code

Extracting the native code compiled by the V8 engine proved to be a challenging task. After dialogue with the V8 developers, it was made clear that there is no convenient method to extract the native code generated by the V8 engine. Despite this, we are able to determine the size of the native code that the V8 engine generates. To do this, we instantiated the WebAssembly modules in the browser and let them run for 60 seconds, allowing time for TurboFan optimization. Then, we used the `--print-wasm-code` flag in V8 to print the size of the native code generated by V8 for both Liftoff and TurboFan. Although we could not directly extract the native code, its size provides insight into how, or if, obfuscation affects the native code.

4.2.7 Measuring the Hash Rate

To measure the hash rate, we implement a cryptojacking client, web server, and WebSocket proxy server as described in Section 2.2. We use the public Webminerpool[8] implementation as a starting point, but we modify the code in several ways. First, we extend the list of crypto mining pools so that we can support all CryptoNight variants. Second, we fixed a bug in the code that caused the hash rate to be measured incorrectly. Lastly, we containerized the client and server, as well as disabled caching, to ensure that the environment was consistent across all experiments.

4.2.8 Dynamic Time Warping

We use DTW to measure the dissimilarity between the WebAssembly binaries through the distance metric. Since the DTW algorithm expects numerical data, we preprocessed the WebAssembly binaries. First, we convert the WebAssembly binaries from the binary format (`wasm`) to the human-readable format (`wat`). Then, we use Python's built-in `hash` method to convert each instruction to a unique integer. In practice, the WebAssembly binary is converted into a sequence of instructions represented as integers. Given the considerable length of the WebAssembly binaries, we use FastDTW, an approximation of DTW that reduces the overall time and space complexity [65].

4.3 Evaluation Metrics

To address the research questions, we use several metrics to evaluate the effectiveness, detectability, and overhead introduced by obfuscation. These metrics are presented in the following sections.

4.3.1 RQ1 – Effectiveness

To evaluate obfuscation effectiveness, we use the distance between the original and obfuscated WebAssembly binaries, as derived from the DTW algorithm.

[8] https://github.com/notgiven688/webminerpool

Definition 2 (Distance). The distance denotes the least cost of aligning two sequences of instructions, each representing a WebAssembly binary. The value represents the number of adjustments needed for one or both sequences to correspond to the other. As such, large distances indicate a substantial dissimilarity.

Moreover, we investigate how obfuscation affects the size of the native code generated by the V8 engine and whether the TurboFan compiler can effectively eliminate instructions introduced by obfuscation. To this end, we extract the native code size as described in Section 4.2.6 and compute the relative increase in native code size after obfuscation has been applied. This is performed for the native code generated by both Liftoff and TurboFan.

Definition 3 (Native code size increase). The increase in native code size indicates the relative increase in the size of the native code after obfuscation. This metric is calculated for the native code generated by the Liftoff and TurboFan compilers separately.

If N denotes the size of the original native code, and N' is the size after obfuscation, then:

$$\text{Native code size increase} = \frac{N' - N}{N} \cdot 100$$

4.3.2 RQ2 – Detectability

To assess how effective the obfuscation methods are in evading detection, we feed the obfuscated binaries to the cryptojacking detectors presented in Section 4.2.5. Following this, we calculate the resulting precision, recall, and F_1 scores to assess the accuracy of the detection methods. These measures are formally defined as follows:

Definition 4 (Precision). Precision measures how many of the retrieved items are relevant. In the context of cryptojacking, it measures how many of the items identified as crypto miners are actually crypto miners.

Precision is mathematically defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Definition 5 (Recall). Recall measures how many of the relevant items are retrieved. In the context of cryptojacking, it measures how many of the crypto miners are identified as crypto miners.

Recall is mathematically defined as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Definition 6 (F_1 score). The F_1 score serves as a single metric that combines precision and recall. It is the harmonic mean of these two quantities, and hence, it gives equal weightage to both.

The F_1 score is mathematically defined as:

$$F_1 \text{ score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

In this paper, we use the F_1 score as the primary metric to evaluate the overall accuracy of the detection methods.

4.3.3 RQ3 – Overhead

We determine the size overhead by comparing the sizes of the original and obfuscated binaries. The file size is measured in bytes using Python's `getsize` method. The relative increase in file size is then determined.

Definition 7 (File size increase). The file size increase refers to the relative increase in file size caused by obfuscation.

If S is the original file size and S' the size after obfuscation, then:

$$\text{File size increase} = \frac{S' - S}{S} \cdot 100$$

For the crypto mining binaries, we measure and compare the hash rates in the original and obfuscated cases. To this end, we implement a cryptojacking setup as described in Section 4.2.5. We let the binaries calculate hashes for 100 seconds before measuring the total hashes.

Definition 8 (Hash rate). The hash rate is defined as the number of hashes calculated per second.

If h is the total number of hashes calculated in time t, then:

$$\text{Hash rate} = \frac{h}{t}$$

To quantify the performance overhead introduced by obfuscation, we calculate the relative hash rate of the obfuscated binaries compared to the original binaries.

Definition 9 (Relative hash rate). The relative hash rate is a measure of the change in performance due to obfuscation.

If H is the original hash rate, and H' the hash rate after obfuscation, then:

$$\text{Relative hash rate} = \frac{H'}{H} \cdot 100$$

5 Results

In this section, we present the results of our experiments, addressing the research questions outlined in the introduction. We begin by examining the effectiveness of the obfuscation methods in producing dissimilar WebAssembly binaries and their impact on the resulting native code. We then assess the detectability of the obfuscated binaries by state-of-the-art cryptojacking detectors. Finally, we quantify the overhead introduced by the obfuscation methods in terms of file size and hash rate.

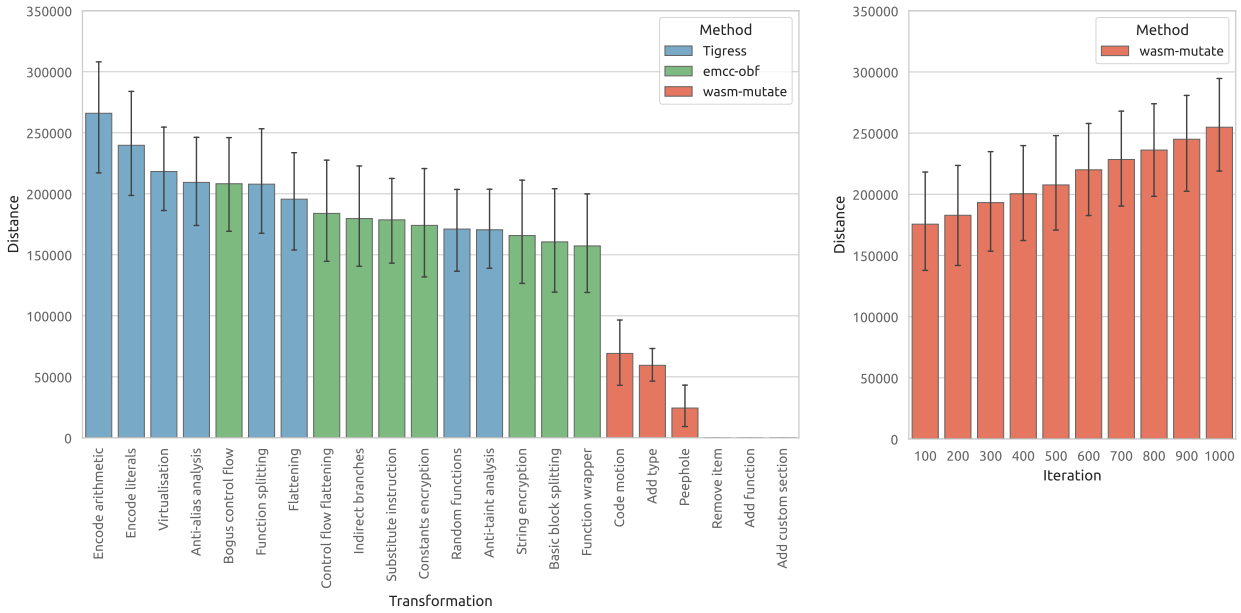
5.1 Effectiveness

5.1.1 Distances After Obfuscation

To assess the effectiveness of obfuscation methods, we measured the distances between the original and obfuscated WebAssembly binaries for each method, transformation, and iteration. The distance metric (Definition 2) quantifies the dissimilarity between binaries, with larger distances indicating more effective obfuscation.

Figure 7 presents the distances for each obfuscation method and transformation, revealing that Tigress is the most effective method with an average distance of 209,000, followed by emcc-obf (176,000) and wasm-mutate (30,000). Notably, wasm-mutate achieves distances up to 252,000 when transformations are stacked, rivaling Tigress in effectiveness. The most effective transformations overall are encode arithmetic, encode literals, and virtualization, all applied by Tigress. However, the type of obfuscation (data or control) that performs best varies by method. Data obfuscations like encode arithmetic and encode literals are more effective for Tigress, while control obfuscations such as control flow flattening and code motion are more effective for emcc-obf and wasm-mutate.

Figure 8 shows the distances grouped by application category. The noticeable variations in distances between application categories can be attributed to the average sizes of the applications within each category. Longer sequences usually lead to larger distances, and games are typically larger than utilities due to the inclusion of external libraries.



(a) Distances for each transformation.

(b) Distances for each iteration.

Figure 7: Distances for each obfuscation method, transformation, and iteration sorted in descending order. The error bars shown are indicative of a 95% confidence interval.

Therefore, the distances should not be compared directly across application categories. Instead, the focus should be on the relative effectiveness of the transformations within each specific application category.

Comparing transformations within each category, we observe that the most effective transformations depend on the application type. For crypto miners, encode arithmetic and substitute instructions prove to be the most efficient. In the case of games, encode literals and constants encryption are most effective. Conversely, virtualization and string encryption are most effective for utilities, although string encryption is the *least* effective for games and crypto miners. These observations highlight the fact that the effectiveness of the transformations is largely dependent on the nature of the application being obfuscated.

This observation is further reinforced by the observation that transformations found effective at one abstraction level also perform well at other abstraction levels. Encode arithmetic (applied to the source code) and the similar substitute instructions (applied to the LLVM bitcode) are the most effective transformations for crypto miners. Similarly, encode literals (applied to the source code) and the corresponding constants encryption (applied to the LLVM bitcode) are the most effective transformations for games. This highlights the significant influence the content of the application has on the effectiveness of the transformations, regardless of the abstraction level at which transformations are applied.

5.1.2 Native Code Size Increase

To assess the impact of obfuscation on the size of native code, we measured the relative increase in native code size (Definition 3) for each obfuscation method, transformation, and iteration after lazy compilation (Liftoff) and optimization (TurboFan) in the V8 engine. These values are calculated relative to the initial native code sizes before obfuscation for both Liftoff and TurboFan-produced code.

Figure 9 presents the relative increase in native code size for each obfuscation method and transformation. Although TurboFan may show a larger relative increase in some instances, it still reduces the overall native code size by about 30% on average compared to Liftoff. For example, consider the situation where Liftoff initially generates 100MB of native code. TurboFan optimizes this to 50MB. After obfuscation, Liftoff's output increases to 200MB, and TurboFan's optimization reduces this to 150MB. So, despite Liftoff showing a 100% relative increase and TurboFan a 200% relative

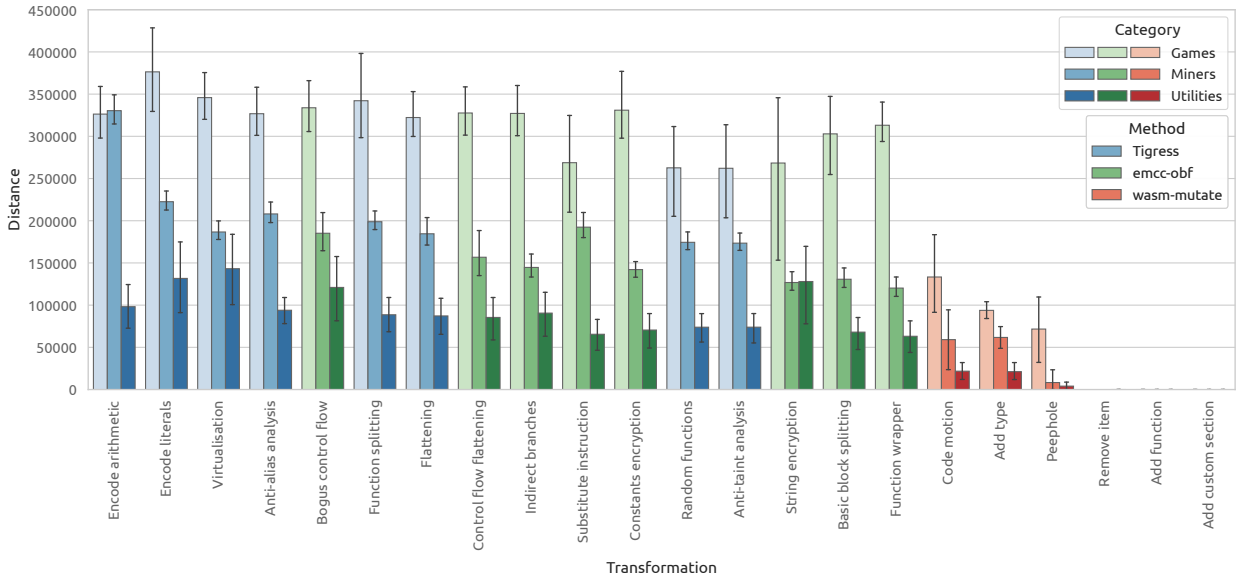


Figure 8: Distances for each obfuscation method and transformation grouped by program category, sorted by the average distances in descending order. The error bars shown are indicative of a 95% confidence interval.

increase after obfuscation, TurboFan's optimization still results in an overall reduction of native code in the original and obfuscated case.

Our findings show that the transformations from Tigress lead to the largest increase in native code, with an average of 87.25% and 73.25% after compilation by Liftoff and TurboFan, respectively. Emcc-obf causes considerably smaller increases, averaging 20% and 22% after Liftoff and TurboFan compilation, while wasm-mutate demonstrates the slightest increase of 10% for both Liftoff and TurboFan. However, when stacking transformations, wasm-mutate substantially increases the native code size, with relative increases ranging from 16% to 140%, as shown in Figure 10.

The impact of data and control obfuscations on native code size varies by obfuscation method. For Tigress, data obfuscations like encode literals cause a more substantial increase than control obfuscations like virtualization. In contrast, for emcc-obf, control obfuscations such as bogus control flow and control flow flattening impose a larger increase compared to data obfuscations like constants encryption and string encryption. For wasm-mutate, there are no discernible differences between control and data obfuscation.

Interestingly, for Tigress, the relative increase in native code is larger for Liftoff than for TurboFan, while the opposite is true for emcc-obf and wasm-mutate. In either case, TurboFan always decreased the size of the native code, but it is unlikely that it entirely eliminated the instructions introduced by obfuscation.

5.1.3 Summary

The effectiveness of WebAssembly obfuscation depends on the obfuscation method, transformation, and application being obfuscated. Tigress was found to be the most effective method, with encode arithmetic, encode literals, and virtualization being the most effective transformations overall. The type of obfuscation (data or control) that performs best varies by method, with data obfuscations being more effective for Tigress and control obfuscations being more effective for emcc-obf and wasm-mutate. The effectiveness of transformations also depends on the application type, with crypto miners benefiting from arithmetic encoding, games from literal encoding and constant encryption, and utilities from virtualization and string encryption. Obfuscation consistently increased the size of the native code generated by the V8 engine, with Tigress causing the largest increase. Although the V8 engine's TurboFan optimizer reduced the native code size by 30% on average, it was unable to completely remove the additional instructions introduced by obfuscation.

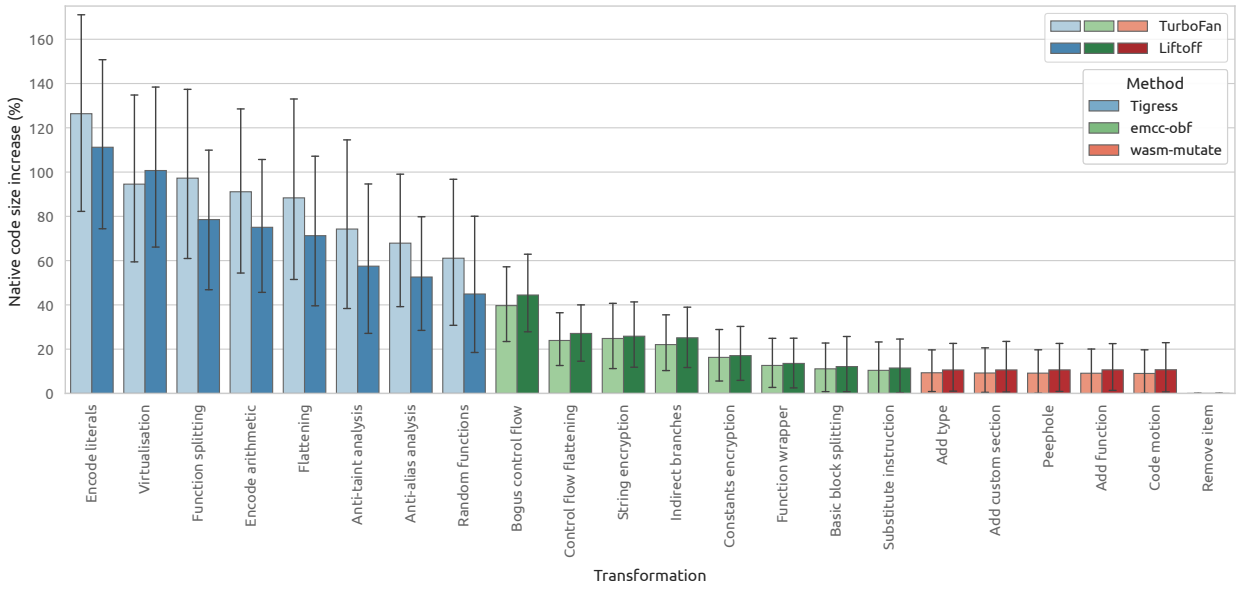


Figure 9: Native code size increase for each obfuscation method and transformation after lazy compilation (Liftoff) and optimization (TurboFan) in the V8 engine, sorted by the average native code size increase in descending order. The error bars shown are indicative of a 95% confidence interval.

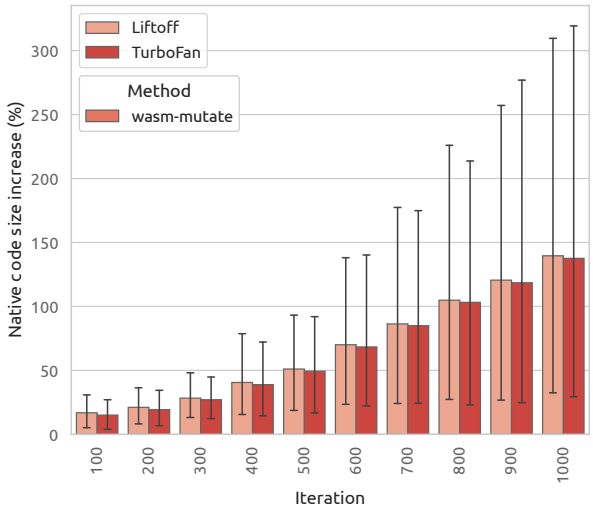


Figure 10: Native code size increase for each iteration applied with wasm-mutate after lazy compilation (Liftoff) and after optimization (TurboFan) in the V8 engine. The error bars shown are indicative of a 95% confidence interval.

5.2 Detectability

5.2.1 Detection Results

To assess the effectiveness of the obfuscation methods in evading detection, we fed the obfuscated binaries to the cryptojacking detectors presented in Section 2.3.1. We then calculated the resulting precision (Definition 4), recall (Definition 5), and F_1 scores (Definition 6) to measure accuracy of the detection methods before and after obfuscation.

Figure 11: F_1 scores for each detection and obfuscation method. Darker colours indicate a higher F_1 score, while lighter colours indicate a lower F_1 score.

Figure 11 shows the F_1 scores for each detection and obfuscation method. We find that detection methods with higher accuracy tend to decrease in accuracy after obfuscation. For example, MINOS, having been re-trained for better accuracy, exhibits decreased accuracy after obfuscation. Here, Tigress proves the most effective, reducing the F_1 score from 1.0 to 0.67, with emcc-obf and wasm-mutate decreasing the F_1 score to 0.77 and 0.81, respectively. In contrast, less accurate detection methods generally improve in accuracy after obfuscation. Both WASim (neural) and WASim (naive) see an increase in accuracy after obfuscation. Tigress, which was the most effective in decreasing the accuracy of MINOS, is the least effective for WASim (neural), increasing the F_1 score from 0.33 to 0.66. Similarly, emcc-obf and wasm-mutate also increase WASim accuracy (neural), albeit to a lesser extent, increasing the F_1 scores to 0.37 and 0.65, respectively.

Table 3 presents the precision, recall, and F_1 scores of MINOS and WASim (neural) after obfuscation. We exclude WASim (naive) and the other detection methods from the table due to their low accuracy. From this, anti-alias analysis emerges as the most effective transformation, resulting in an F_1 score of 0 for both MINOS and WASim. Several control obfuscations, such as flattening, control flow flattening, virtualization, and indirect branches, prove highly effective, resulting in F_1 scores of 0 for WASim. However, not all control obfuscations achieve such results; function splitting and random functions increase the F_1 score of WASim to 0.95. Despite data obfuscations like encode arithmetic and string encryption showing some effectiveness in evading detection, control obfuscation tends to be more effective overall.

Moreover, we find that the effectiveness of the transformations varies significantly depending on the specific detection method. While flattening is effective for WASim (F_1 score of 0), it is not nearly as effective for MINOS (F_1 score of 0.80). Similarly, function splitting is effective for MINOS (F_1 score of 0.27) but not WASim (F_1 score of 0.95).

Interestingly, even after obfuscation, the recall often exceeds the precision for both MINOS and WASim , which suggests that the obfuscation methods are more effective at causing false positives than false negatives. In other words, the drop in accuracy is primarily due to benign applications being mistakenly identified as crypto miners, rather than crypto miners evading detection. There are certainly exceptions to this observation; function splitting and bogus control flow effectively reduce recall for MINOS, and most control obfuscations do the same for WASim.

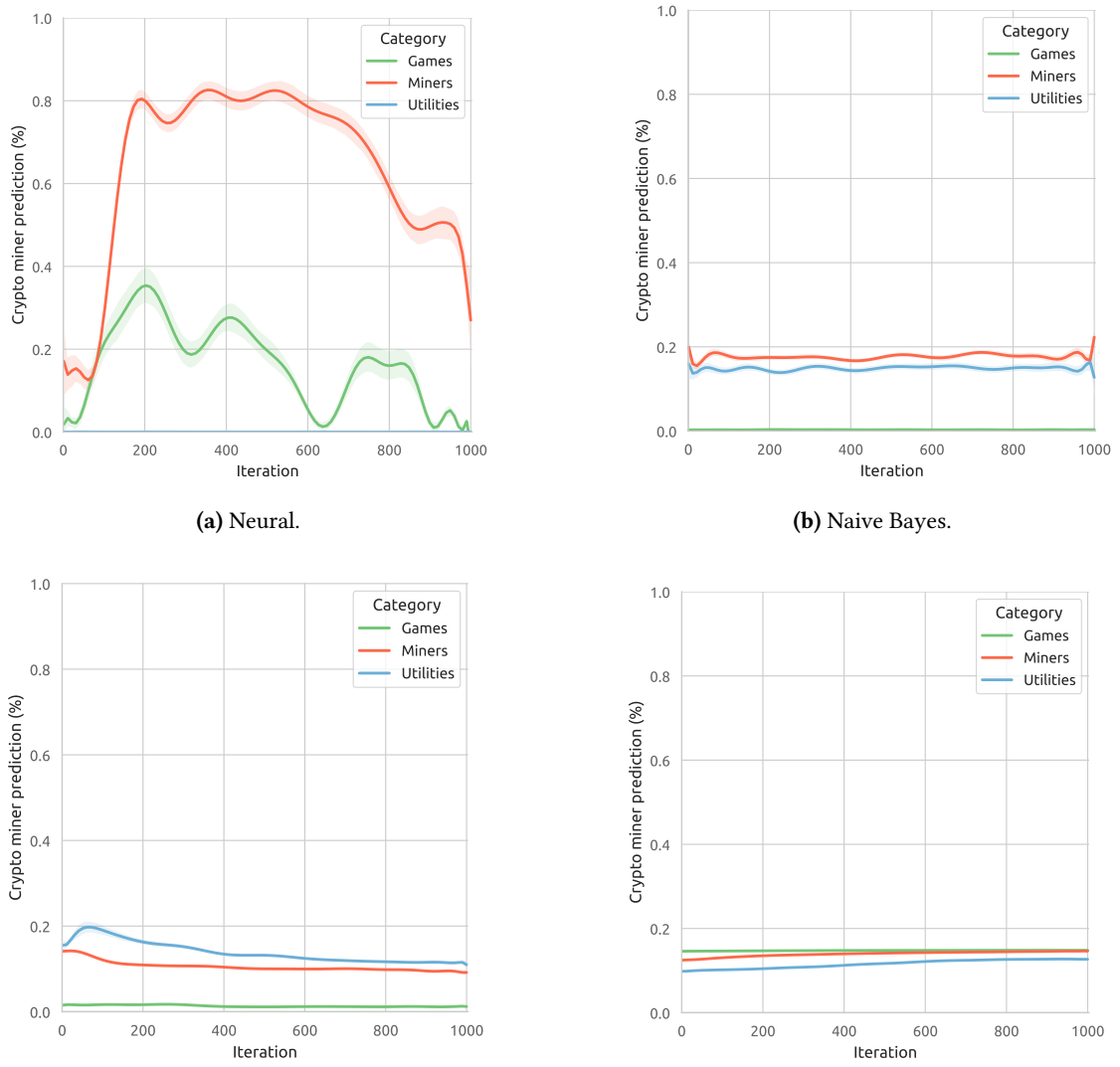
As observed previously, applying individual transformations with wasm-mutate does not significantly impact the accuracy of the detection methods. However, stacking multiple transformations yields more promising results. For

		MINOS			WASim (neural)		
Obfuscation	Transformation	P*	R*	F_1	P*	R*	F_1
Original	None	1.00	1.00	**1.00**	1.00	0.20	**0.33**
Tigress	Flattening	0.67	1.00	**0.80**	**0.00**	**0.00**	**0.00**
	Random functions	0.71	1.00	0.83	0.91	1.00	**0.95**
	Function splitting	0.40	0.20	**0.27**	0.91	1.00	**0.95**
	Virtualization	0.75	0.90	0.82	**0.00**	**0.00**	**0.00**
	Encode arithmetic	0.58	0.70	0.63	0.78	0.70	0.74
	Encode literals	0.56	1.00	0.72	0.83	1.00	0.91
	Anti-alias analysis	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	Anti-taint analysis	0.77	1.00	0.87	0.89	0.80	0.84
emcc-obf	Control flow flattening	0.67	1.00	**0.80**	**0.00**	**0.00**	**0.00**
	Bogus control flow	0.55	0.60	0.57	0.60	0.30	0.40
	Indirect branches	0.64	0.90	0.75	**0.00**	**0.00**	**0.00**
	Basic block splitting	0.77	1.00	0.87	1.00	0.20	0.33
	Function wrapper	0.91	1.00	0.95	1.00	0.60	0.75
	Substitute instruction	0.73	0.80	0.76	1.00	0.50	0.67
	Constants encryption	0.62	0.80	0.70	0.67	0.20	0.31
	String encryption	0.62	1.00	0.77	0.40	0.20	0.27
wasm-mutate	Code motion	0.83	1.00	0.91	1.00	0.20	0.33
	Peephole	0.91	1.00	0.95	1.00	0.20	0.33
	Add function	0.91	1.00	0.95	1.00	0.20	0.33
	Add Type	0.91	1.00	0.95	1.00	0.20	0.33
	Add custom section	0.83	1.00	0.91	1.00	0.20	0.33
	Remove item	0.91	1.00	0.95	1.00	0.20	0.33
wasm-mutate	Iteration 100	0.75	0.90	**0.82**	0.75	0.30	**0.43**
	Iteration 200	0.75	0.90	0.82	0.82	0.90	0.86
	Iteration 300	0.69	0.90	0.78	0.88	0.70	0.78
	Iteration 400	0.69	0.90	0.78	0.82	0.90	0.86
	Iteration 500	0.69	0.90	**0.78**	0.90	0.90	**0.90**
	Iteration 600	0.67	0.80	0.73	1.00	0.80	0.89
	Iteration 700	0.69	0.90	0.78	0.89	0.80	0.84
	Iteration 800	0.67	0.80	0.73	0.88	0.70	0.78
	Iteration 900	0.64	0.70	0.67	1.00	0.40	0.57
	Iteration 1000	0.64	0.70	**0.67**	1.00	0.20	**0.33**

* Abbreviations: Precision (P), and recall (R).

Table 3: Precision, recall, and F_1 scores for MINOS and WASim (neural) after applying obfuscation with Tigress, emcc-obf, and wasm-mutate.

MINOS, the F_1 score consistently decreases as more transformations are applied, indicating that the obfuscation becomes more effective at evading detection. For WASim, however, the F_1 score inconsistently increases from 0.43 to 0.90 after 500 iterations, before falling back to 0.33 after 1000 iterations. This suggests that the effectiveness of stacked transformations may not always scale linearly with the number of transformations applied, and there may be a point of diminishing returns or even a decrease in effectiveness.



(a) Neural.

(b) Naive Bayes.

(c) RF.

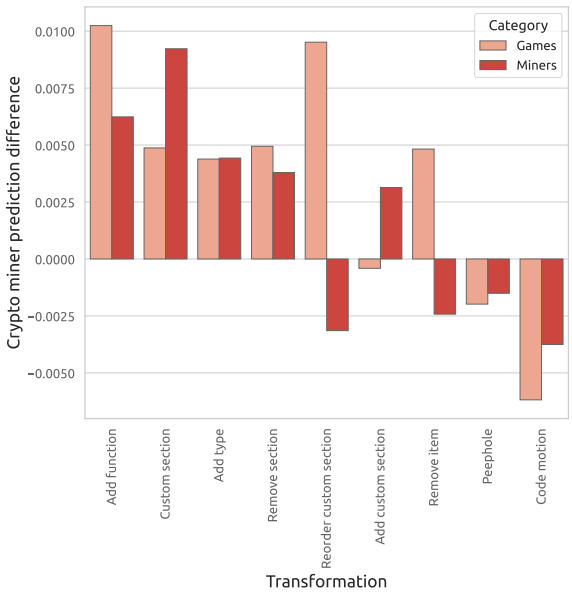
(d) SVM.

Figure 12: The four different WASim classifiers and their respective prediction likelihoods for identifying a WebAssembly binary as a crypto miner as the binaries undergo iterative transformations using wasm-mutate. For each iteration, a randomly selected transformation is applied.

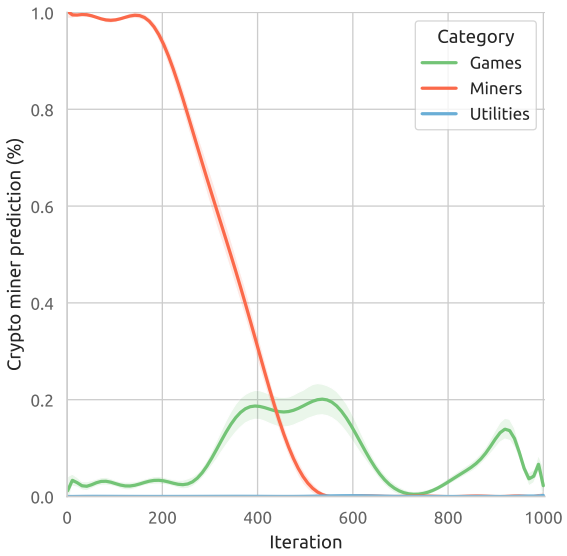
5.2.2 WASim Classifiers

Figure 12 shows the predictions of the different WASim classifiers in response to stacked wasm-mutate transformations. Predictions for the naive Bayes, RF, and SVM classifiers remain relatively constant as more transformations are applied. This shows that they are not affected by the transformations, indicating that they are resilient to obfuscation. On the other hand, the neural classifier exhibits variance in its predictions as more transformations are applied, signifying that it is sensitive to the transformations. Interestingly, this suggests that the neural classifier is not nearly as obfuscation-resilient as the other classifiers, despite being recommended by the authors of WASim.

This observation encouraged us to investigate which wasm-mutate transformations are the most effective for evading detection. Specifically, which transformations decrease the crypto mining predictions of WASim (neural) the most? As can be seen in Figure 13a, the code motion and peephole transformations emerge as the most effective in reducing the crypto miner predictions, making them prime candidates for evading cryptojacking detection.



(a) Most effective wasm-mutate transformations for altering the crypto miner predictions of the WASim (neural) classifier.



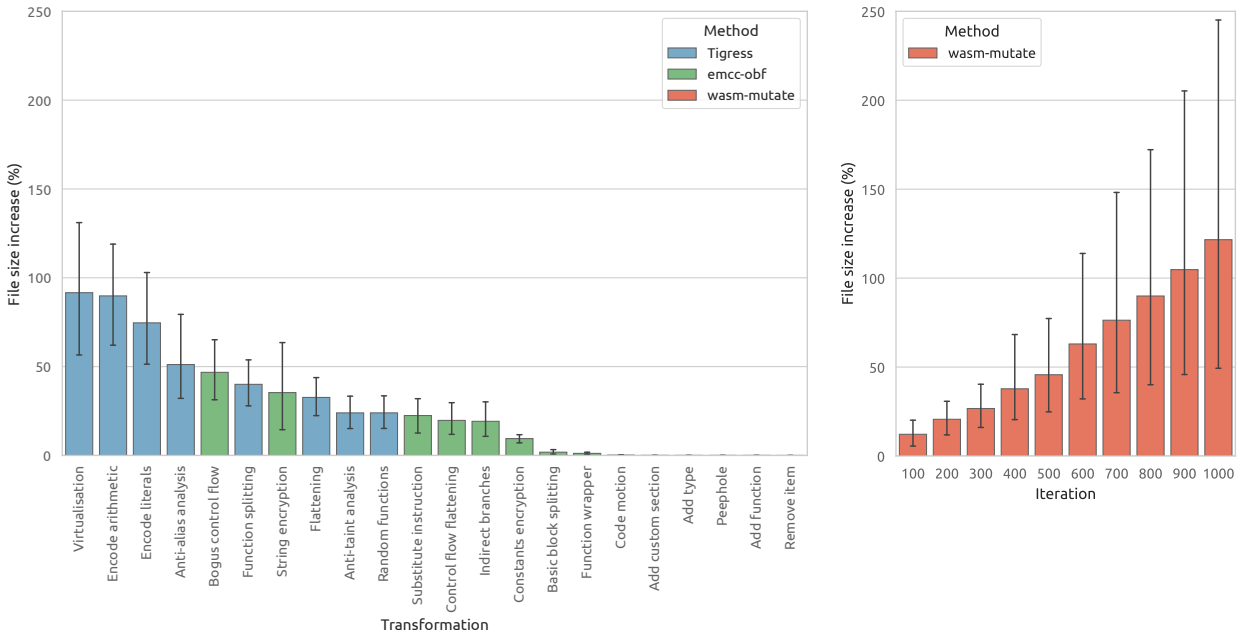
(b) WASim (neural) predictions for WebAssembly binaries that have been iteratively obfuscated with the code motion and peephole transformations.

Figure 13: Figure (a) shows the most effective wasm-mutate transformations for evading WASim (neural) detection. Figure (b) shows the predictions of WASim (neural) for WebAssembly binaries that have been iteratively obfuscated using the most effective transformations; namely code motion and peephole.

Intrigued by these findings, we explored the possibility of strategically applying these transformations to evade WASim (neural) detection. First, we apply random mutations to the crypto mining binaries and select the resulting crypto mining binaries that have the highest crypto miner predictions. Then, we iteratively apply the code motion and peephole transformations to those binaries and observe the predictions of the WASim (neural) classifier. The results are shown in Figure 13b. The crypto miner binaries are initially labeled as crypto miners with 100% probability. Then, after 550 iterations, the crypto miner binaries are labeled as benign with 100% probability, successfully evading detection and demonstrating how WASim (neural) can be strategically evaded using targeted transformations.

5.2.3 Summary

Obfuscation can be effective at evading state-of-the-art cryptojacking detectors, but its effectiveness depends on the specific detection method, obfuscation method, and transformation applied. Tigress proved most effective in evading MINOS, while emcc-obf was more effective against WASim. Anti-alias analysis was the only transformation that completely evaded both MINOS and WASim, but several control obfuscation transformations also achieved this for WASim. In general, control obfuscation was more effective than data obfuscation. Interestingly, the decrease in detection accuracy was often due to benign applications being misclassified as crypto miners, rather than crypto miners successfully evading detection. Stacking transformations with wasm-mutate yielded promising results, with the effectiveness increasing as more transformations were applied for MINOS, but fluctuating for WASim. However, by strategically applying the most effective transformations, namely code motion and peephole, wasm-mutate could completely evade WASim (neural) detection. The resilience of WASim's classifiers to obfuscation varied, with the neural classifier being more sensitive to transformations despite being recommended by WASim's authors.



(a) File sizes for each transformation.

(b) File sizes for each iteration.

Figure 14: File size increase for each obfuscation method, transformation, and iteration sorted in descending order. The error bars shown are indicative of a 95% confidence interval.

5.3 Overhead

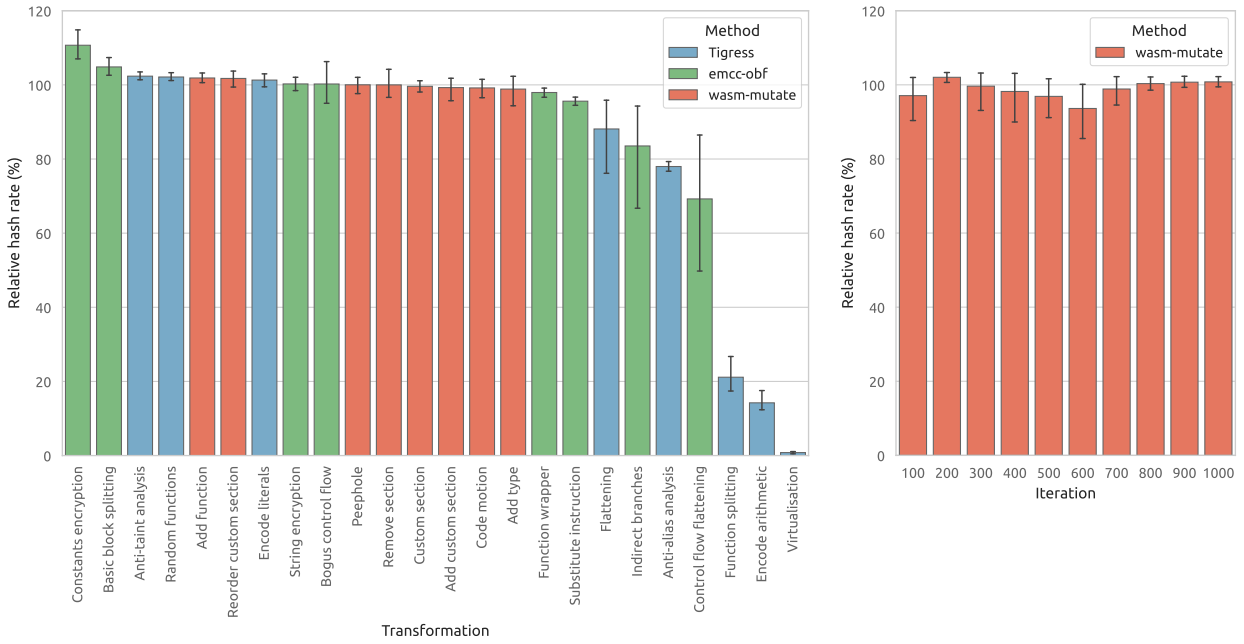
5.3.1 File Size Overhead

Figure 14 shows the relative increase in file size (Definition 7) after obfuscation for each obfuscation method, transformation, and iteration. Tigress increased the file size the most, averaging 53%, followed by emcc-obf and wasm-mutate at 19% and 0.2%, respectively. Despite wasm-mutate's minimal file size overhead when applying individual transformations, the overhead increases linearly when stacked transformations are applied, averaging a 59% increase, ranging from 12% to 121%. The transformations that contribute most significantly to file size increase are virtualization, encode arithmetic, and encode literals, which lead to increases of 91.6%, 89.8%, and 74.6%, respectively. Interestingly, no significant differences were observed between control and data obfuscation in terms of file size overhead.

5.3.2 Hash Rate Overhead

Figure 15 shows the relative hash rate (Definition 9) for each obfuscation method, transformation, and iteration. Tigress, emcc-obf, and wasm-mutate average 63.1%, 95.2%, and 99.8% of the original hash rate, respectively. Notably, two-thirds of the transformations do not significantly impact the hash rate. Surprisingly, constant encryption and basic block splitting increase the hash rate by 110.7% and 104.8%, respectively. Moderate overheads are seen in transformations like flattening, control flow flattening, indirect branches, and anti-alias analysis, delivering 70.2% to 88.1% of the original hash rate. Significant overhead is introduced by Tigress' virtualization, encode arithmetic, and encode literals transformations, achieving 1% to 21.1% of the original hash rate. Wasm-mutate imposes negligible hash rate overhead, which persists even with stacked transformations. However, the hash rate fluctuates between iterations, ranging from 93.6% to 100.8% of the original hash rate, averaging 99.8%.

Figure 16 shows the relative hash rate (Definition 9) after obfuscation for each obfuscation method, transformation, and CryptoNight variant. There are noticeable differences between the variants. Generally, cn-0 retains a higher hash rate compared to the other variants, indicating that it is less impacted by obfuscation. There are also distinct



(a) Hash rates for each transformation.

(b) Hash rates for each iteration.

Figure 15: Relative hash rates for each obfuscation method, transformation, and iteration sorted in descending order. The error bars shown are indicative of a 95% confidence interval.

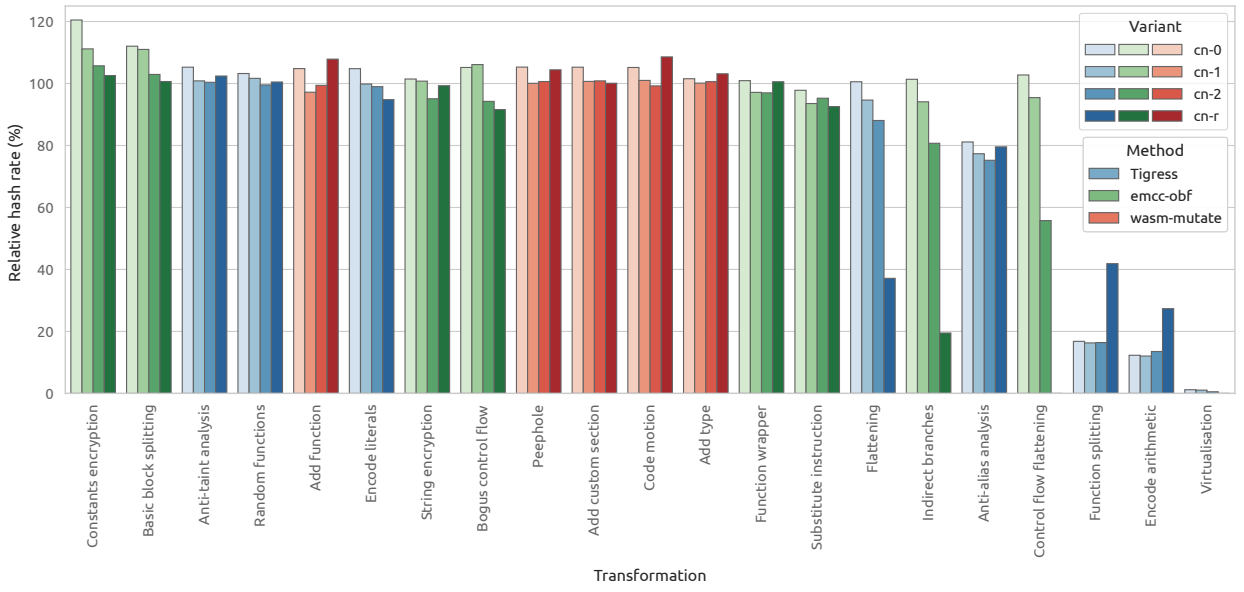


Figure 16: Relative hash rates for each obfuscation method, transformation, and CryptoNight variant sorted in descending order.

differences between cn-r and the other variants. Transformations like flattening, control flow flattening, and indirect branches bring about considerable overhead for cn-r but less so for the other variants. Contrastingly, function splitting and encode arithmetic impose less overhead on cn-r but significantly impact the other variants. This suggests that the transformation overhead is largely dependent on the specific CryptoNight variant being obfuscated.

5.3.3 Summary

Tigress consistently introduces the largest overheads, reflected in both the file size, averaging a 53% increase, and the hash rate, averaging a reduction to 63.1% of the original hash rate. Moreover, Tigress' virtualization and encode arithmetic transformations consistently introduce the largest overheads for both file size and hash rate. On the contrary, emcc-obf exhibits considerably less overhead, with a 19% increase in file size and retention of 95.2% of the original hash rate. Remarkably, emcc-obf's constants encryption and basic block splitting transformations even increase the hash rate. Although wasm-mutate introduces large file size overheads when stacking the transformations, ranging from 12% to 121%, it demonstrates minimal hash rate overhead, retaining 99.8% of the original hash rate on average. Additionally, variations between CryptoNight variants are observed, with cn-0 generally retaining a higher hash rate than other variants, while significant differences in hash rate are evident between cn-r and other variants, highlighting the importance of considering the specific variant when assessing the impact of obfuscation.

6 Discussion

In this section, we discuss the implications of our findings and provide insights into the effectiveness, detectability, and overhead of WebAssembly obfuscation. We analyze the factors that contribute to the varying effectiveness of the obfuscation methods and transformations, and we interpret the results in the context of real-world scenarios. We also discuss the limitations of our study and potential avenues for future research.

6.1 Effectiveness

Upon analyzing the WebAssembly binaries and resulting native code after obfuscation, we find that Tigress is generally the most effective method, followed by emcc-obf and wasm-mutate. This effectiveness is likely due to Tigress implementing more advanced transformations, as evidenced by the larger distances, increased binary size, and larger native code produced after compilation in the browser. These findings align with the research by Suresh et al. [68], which concluded that Tigress generally applies more advanced transformations than OLLVM, the basis for emcc-obf.

We do not attribute the effectiveness of obfuscation methods to the abstraction level at which it is applied – be it source code, LLVM, or WebAssembly – but rather the implementation of the transformations themselves. Implementing advanced transformations is generally simpler at higher abstraction levels due to the powerful abstractions provided by the programming language. For example, implementing virtualization is simpler at the source code level, compared to implementing it in the low-level WebAssembly language. However, lower abstraction levels allow for more targeted transformations, as seen with wasm-mutate's direct manipulation of the WebAssembly module. Despite this, these targeted transformations are not advanced enough to be effective for obfuscation purposes when applied individually.

The ineffectiveness of individual transformations in wasm-mutate likely stems from its design as a diversifier rather than an obfuscator. Wasm-mutate makes subtle modifications to the WebAssembly module, allowing it to produce many diversified variants. This is in contrast to Tigress and emcc-obf, which implement advanced transformations that significantly modify the program. However, the results suggest that stacking wasm-mutate transformations can be an effective approach, which aligns with previous studies [14].

Our findings indicate that the most effective transformations are encode arithmetic, encode literals, and virtualization. In contrast, Bhansali et al. [9] found anti-alias analysis to be the most effective, followed by virtualization and flattening. We both find that the application's content significantly impacts transformation effectiveness. In our findings, crypto miners benefit the most from the encode arithmetic transformation, likely due to the large number of

arithmetic operations required for calculating the hashes. Games benefit most from encode literals due to the large number of integer values used for scores, health points, and colours, to name a few. Lastly, utilities benefit most from string encryption due to the large number of strings used for printing to the console and interacting with the user. Contrastingly, Bhansali et al. identified anti-alias analysis as the most effective for all application types, except crypto miners, where virtualization was deemed most effective. These differences likely stem from our divergent comparison methods; Bhansali et al. used cosine similarity, while we used a distance-based metric more suitable for WebAssembly.

The error bars in the plots further highlight the crucial influence of the application content on the effectiveness of the transformations. For instance, Figure 8 has a significant error bar for string encryption within the games category, implying that the effectiveness of this transformation depends on the amount of text used in each game. As text content in games can significantly vary, the impact of string encryption will differ accordingly.

It is also imperative to highlight that the significant error bars, such as those observed in Figure 10, stem from the heterogeneity of the dataset and the consequent variation in application sizes. Games, for instance, tend to be larger than utilities due to the incorporation of external libraries. Longer sequences tend to produce larger distances, accounting for the substantial error bars in these plots.

Tigress consistently produced more native code than emcc-obf and wasm-mutate, but this increase is not directly reflected in the WebAssembly binary size. We hypothesize that this discrepancy is due to Liftoff's lazy compilation strategy, which only compiles invoked functions. Emcc-obf likely generates code that is never executed and thus never compiled to native code, resulting in large WebAssembly binaries but smaller native code, while Tigress generates code that is executed and compiled to native code.

TurboFan is more efficient at optimizing native code obfuscated using Tigress than with emcc-obf or wasm-mutate, likely because Tigress-produced native code contains more "junk" code that is easier to optimize away. Although TurboFan decreased the native code size by an average of 30%, it is doubtful that it entirely eliminated the instructions introduced by obfuscation. Even for individual wasm-mutate transformations, TurboFan was unable to completely remove the extraneous instructions. Considering the significant increase in native code even after optimization, it is likely that at least some obfuscation remained. These findings contrast with previous studies on LLVM diversification for WebAssembly, where the native code was found to be identical after TurboFan optimization [4].

6.2 Detectability

The experiments demonstrate that obfuscation techniques can successfully evade state-of-the-art cryptojacking detectors. For MINOS, this is not surprising, as obfuscation significantly alters the WebAssembly binaries, which are converted into grayscale images and used as input for the classifier. The obfuscation process likely leads to misclassification of the obfuscated program because the resulting images deviate considerably from the ones used during the training of the neural network. This hypothesis is supported by the findings of Loose et al. [42], who demonstrated that modifying the grayscale image input of MINOS could lead to misclassification.

WASim takes a feature-based approach, analyzing properties like the binary size. As more transformations are applied (up to iteration 500), the crypto miner probability gradually increases, potentially because the classifier has learned that miners typically have larger binary sizes than other applications. However, as the binary size continues increasing beyond iteration 500, the probability decreases, possibly because the program becomes too large and complex to fit the expected profile of a miner. WASim's neural classifier proves particularly vulnerable to obfuscation, consistent with the known sensitivity of neural networks to adversarial inputs [69, 30, 42].

Notably, decreased detection rates often stemmed from benign applications being misclassified as miners rather than miners evading detection. This is likely attributed to obfuscation increasing program complexity, leading to an over-classification of benign programs labeled as malicious. Similar findings were reported by Bhansali et al. [9], who found that obfuscation increased the false positive rate to 70%. It is important to stress that in real-world scenarios, false positives are detrimental, as they could result in harmless programs being unnecessarily blocked, degrading the

user experience. Since obfuscation can have legitimate uses, like safeguarding intellectual property, it is crucial to minimize its potential for inducing false positives.

In terms of transformations that were effective in evading detection, we generally observed that control obfuscation was more effective than data obfuscation. This can be attributed to the fact that control obfuscation generally impacts the entire program, while data obfuscation targets specific parts, that is, the data. As such, control obfuscation often results in more significant changes to the program, which is more likely to evade detection. Intriguingly, anti-alias analysis, a preventive transformation, was the most effective for malware evasion. Although neither MINOS nor WASim directly performs alias analysis, it was the only effective transformation for both detection methods. Evidently, anti-alias analysis significantly impacts the features that the neural networks rely on, leading to misclassification. On the contrary, Bhansali et al. found every Tigress transformations to be effective in evading MINOS [9]. However, they used the original MINOS implementation, while we used a reproduction by Cabrera et al. [14], which could explain the difference in results. In addition, they used a small dataset of only two crypto miners, which threatens the validity of their results.

Wasm-mutate was found to be ineffective at evading detection when applying individual transformations. However, when the transformations were stacked, the accuracy of the detection methods significantly decreased. We found that code motion and peephole were the most effective transformations. Similarly, Cabrera et al. [14] found that peephole, add function, and code motion were the most effective transformations for evading detection. They also found that it took between 120 and 635 iterations to evade detection, akin to our average of 550 iterations. However, Cabrera et al. could completely evade MINOS detection using random transformations in under 1000 iterations. We were unable to reproduce this, as we were only able to reduce the F_1 score to 0.67 after 1000 iterations. This difference, we believe, arises from using different datasets.

6.3 Overhead

We found that Tigress and emcc-obf, on average, increased the WebAssembly binary size by 53% and 19%, respectively. These observations align with Suresh et al.'s research [68], wherein the code generated by Tigress was 5% to 78% larger compared to the code generated by OLLVM. Notably, even for similar transformations like control flow flattening, Tigress produces 11.3% larger binaries than emcc-obf. We attribute this to Tigress likely using a more complex control flow flattening algorithm.

While prior work did not measure wasm-mutate's impact on binary size [13], we found that stacking transformations can induce considerable overhead (12% to 120%, averaging 59%). However, strategically selecting transformations to minimize overhead or applying fewer transformations can mitigate this issue, as demonstrated in other domains like Rosette language diversification [44].

In terms of performance overhead, Tigress performed worst (averaging 63.1% of original hash rate), followed by emcc-obf (95.2%) and wasm-mutate (99.8%). These findings align with prior observations of Tigress-generated code causing a 4% to 55.4% larger performance overhead compared to OLLVM-generated code [68]. Function splitting, encode arithmetic, and virtualization induced the largest hash rate overheads and native code size increases. This correlation is unsurprising, as more native code means more instructions for the CPU to execute, reducing performance and hash rate. Tigress's advanced transformations significantly increase native code generation, resulting in lower hash rates.

Interestingly, despite the considerable native code increase from stacking 1000 wasm-mutate transformations, the performance overhead was negligible. We attribute this to hardware optimizations (e.g., pipelining, instruction reordering, branch prediction, cache optimization) applied to the native code, streamlining execution. Tigress's advanced transformations likely generate code too complex for the same degree of hardware optimization, resulting in larger performance overheads even with similar native code increases.

Another noteworthy observation is that constants encryption and basic block splitting increase the hash rate to 110.7% and 104.8% of the original hash rate, respectively. This may be due to emcc-obf's clean-up passes designed to

remove the intermediate values used for obfuscation. We hypothesize that these clean-up procedures inadvertently optimize the code, leading to increased performance.

The WebAssembly binaries that were obfuscated with wasm-mutate fluctuated around the original hash rate, ranging from 93.6% to 100.8% of the original hash rate. We attribute this to wasm-mutate performing transformations in the executable code of the module, which effectively work as optimizations. For example, loop unrolling and code replacements can lead to smaller binaries and inadvertently increase the hash rate. Conversly, certain transformations applied by wasm-mutate may have a detrimental effect on performance, accounting for the observed reductions in hash rate. This hypothesis aligns with the findings of Cabrera et al. [13], who reported wide fluctuations in performance overhead caused by wasm-mutate.

We discovered differences in performance overhead among CryptoNight variants. CryptoNightR (cn-r) is designed to be more resistant to application-specific integrated circuit (ASIC) than its counterparts. This is accomplished by embedding a random component into the algorithm, requiring miners to perform different operations for each block. As a result, CryptoNightR uses a combination of arithmetic and branching operations, with the sequence randomized for each block. Some of these arithmetic operations are generated at runtime, meaning these operations cannot be statically encoded, which explains why encode arithmetic is ineffective for CryptoNightR. Additionally, CryptoNightR has 1.7 times more branch instructions than other variants, explaining the larger impact of flattening, control flow flattening, and indirect branches.

The increasing complexity of CryptoNight algorithms to ensure ASIC-resistance correlates with greater performance overheads from obfuscation. More complex algorithms with more operations and intricate control flows provide more "surface area" for obfuscation to slow down the program. The results show that cn-0 is usually least affected by obfuscation, succeeded by cn-1, cn-2, and cn-r, supporting this hypothesis.

6.4 Interpreting the Findings

The results derived from this paper are intricate and influenced by many factors. Although several transformations can evade cryptojacking detection, they often introduce significant overhead. This raises the question of whether obfuscation can be used for evading cryptojacking detection in real-world scenarios with justifiable overhead. Our assessment suggests that it is feasible, but it depends on the obfuscation and detection methods, as well as the specific crypto miner algorithm.

For instance, anti-alias analysis can evade detection for both MINOS and WASim with every CryptoNight variant, albeit with a 22% reduction in the original hash rate and a 51% increase in file size. However, more desirable results can be achieved by adapting the obfuscation strategy to the specific use case. As an example, the original CryptoNight algorithm can be obfuscated using indirect branches, effectively evading detection by WASim, improving the hash rate to 102% of its initial value while only increasing the file size by 19%. This demonstrates that obfuscation can be viable in real-world scenarios, but it requires careful consideration of the specific use case.

Additionally, wasm-mutate presents potential as a tool for evading detection. Interestingly, in many cases, the performance overhead is negligible, and in some instances, wasm-mutate can even improve the hash rate. Although the file size overhead can be substantial, it can be mitigated by selectively applying transformations that do not significantly increase the file size.

While this paper focuses on crypto mining WebAssembly binaries, we have also included benign applications in our dataset. We find that benign applications can also be effectively obfuscated, and we have highlighted the differences between the different application categories. Although we have not extensively evaluated reverse engineering, the insights gained from this paper can likely be extended to benefit this domain as well.

6.5 Limitations

The dataset used in this paper covers a wide range of applications, but it is not exhaustive and only contains 22 applications. This is primarily due to the constraints imposed by Tigress, which necessitates the use of open-source C projects compatible with the C99 standard. Additionally, they must be compatible with CIL so they can be parsed

and merged into a single source file. However, we were able to include all the prominent CryptoNight variants in the dataset. This should provide extensive coverage of crypto mining malware, as several studies have found that in-the-wild cryptojacking implementations are all based on the CryptoNight algorithm [14, 33]. Moreover, with its diverse set of use cases, the dataset stands comparable to, and in some cases exceeds, the datasets of other obfuscation studies for WebAssembly [9, 14, 42].

The obfuscation methods have been tested on static detection methods and, despite their effectiveness in evading them, are unlikely to be equally effective for dynamic detection methods. Although the transformations sometimes alter the observable behaviour of the program, as evidenced by the increase in native code, dynamic methods based on API calls or similar will observe the same behaviour with or without obfuscation. However, as discussed in Section 2.3.1, dynamic methods are complex to set up, can impose a significant performance overhead, and are not widely used in practice. Therefore, we believe that the static detection methods that we evaluated are the most relevant for the purposes of this paper.

We could not extract the native code compiled by the browser, even after consulting the V8 developers. This is unfortunate, as exploring the semantic differences in the native code after obfuscation would have provided valuable insights. Still, we were able to extract the size of the native code, which indicates how obfuscation potentially affects the resulting native code and whether it is optimized away or not. While we find this useful, we acknowledge that this is a limitation of our research.

7 Future Research

Although this paper has made significant contributions towards understanding WebAssembly obfuscation, there are still avenues for future research. One crucial direction is the development of more robust detection methods that can effectively counter the obfuscation methods explored in this paper. A promising solution is to preprocess the WebAssembly binaries by de-obfuscating them.

Moreover, the need for improved detection methods, irrespective of their obfuscation resistance, cannot be overstated. In this paper, we discovered that more than half (four out of seven) of the detection methods implemented were unable to detect the crypto miners, even before obfuscation.

Another avenue for future research is investigating more advanced obfuscation techniques and possibly developing novel ones designed specifically for WebAssembly. Obfuscation methods tailored to WebAssembly, leveraging its unique features such as the stack machine architecture, would likely be even more effective than the methods explored in this paper. In practice, this could be implemented as optimization passes for Binaryen, similar to what was done for OLLVM. Moreover, combining several obfuscation transformations, possibly at multiple abstraction levels, merits further investigation.

Finally, there is extensive research on cryptojacking but not on other malicious use cases for WebAssembly. WebAssembly can also be used for other malicious purposes, like tech support scams, browser exploits, and script-based keyloggers [41], and it has been used for hacking games [29, 6]. The full potential of WebAssembly for malicious purposes has not been extensively explored yet, and we believe that this is an exciting direction for future research. Besides malicious use cases, obfuscating benign programs to prevent reverse engineering is another interesting direction for future research.

8 Conclusion

In this paper, we have conducted a comprehensive evaluation of code obfuscation techniques for WebAssembly, covering multiple abstraction levels. This represents the most extensive assessment of WebAssembly obfuscation to date. We have demonstrated the effectiveness of obfuscation in producing dissimilar WebAssembly binaries and analyzed the impact on the resulting native code. The results show that obfuscation can successfully evade state-of-the-art cryptojacking detectors, although the effectiveness largely depends on the specific obfuscation transformation, detection method, and crypto mining algorithm employed. Despite the observed performance overheads, we have

demonstrated how obfuscation can be used to evade detection with justifiable overhead in real-world scenarios through careful selection of transformations. These findings are significant for researchers and academics. Our findings offer insights into which transformations are most effective in evading detection, which can inform the development of more robust detection methods. The novel obfuscation method introduced, along with the extensive dataset of obfuscated WebAssembly binaries, provides a solid foundation for future research in this domain as WebAssembly continues to gain widespread adoption across diverse platforms and use cases.

References

[1] John Aach and George M Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, 17(6):495–508, 2001.

[2] Cyber Threat Alliance. The Illicit Cryptocurrency Mining Threat. https://cyberthreatalliance.org/wp-content/uploads/2018/09/CTA-Illicit-CryptoMining-Whitepaper.pdf, 2018. [Accessed 25th Nov. 2022].

[3] Javier Cabrera Arteaga, Nicholas Fitzgerald, Martin Monperrus, and Benoit Baudry. Wasm-mutate: Fuzzing webassembly compilers with e-graphs. In *E-Graph Research, Applications, Practices, and Human-factors Symposium*, 2022.

[4] Javier Cabrera Arteaga, Orestis Malivitsis, Oscar Vera Perez, Benoit Baudry, and Martin Monperrus. Crow: Code diversification for webassembly. *arXiv preprint arXiv:2008.07185*, 2020.

[5] Javier Cabrera Arteaga, Martin Monperrus, and Benoit Baudry. Scalable comparison of JavaScript v8 bytecode traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, oct 2019.

[6] Jack Baker. Hacking WebAssembly Games with Binary Instrumentation. https://av.tib.eu/media/48379, June 2023. [Accessed 13. Jun. 2023].

[7] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, dec 2016.

[8] Zahra Bazrafshan, Hashem Hashemi, Seyed Mehdi Hazrati Fard, and Ali Hamzeh. A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology*. IEEE, may 2013.

[9] Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A. Selcuk Uluagac. 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, page 140–145, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Weikang Bian, Wei Meng, and Mingxue Zhang. MineThrottle: Defending against Wasm In-Browser Cryptojacking. In *Proceedings of The Web Conference 2020*, WWW '20, page 3112–3118, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Sandrine Blazy, Stephanie Riaud, and Thomas Sirvent. Data tainting and obfuscation: Improving plausibility of incorrect taint. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, sep 2015.

[12] bytecodealliance. wasm-tools, May 2023. [Accessed 25. May 2023].

[13] Javier Cabrera Arteaga. Artificial Software Diversification for WebAssembly, 2022. Doctor thesis (CROW + MEWE).

[14] Javier Cabrera-Arteaga, Martin Monperrus, Tim Toady, and Benoit Baudry. Webassembly diversification for malware evasion. *Computers & Security*, 131:103296, 2023.

[15] Joseph C Chen. Cryptocurrency Miner Script Found on AOL Ad Platform. https://www.trendmicro.com/en_us/research/18/d/cryptocurrency-web-miner-script-injected-into-aol-advertising-platform.html, April 2018. [Accessed 16. May 2023].

[16] Ericka Chickowski. Container Supply Chain Attacks Cash In on Cryptojacking. *Dark Reading*, September 2022.

[17] Clang. clang: lib/Basic/Targets/WebAssembly.h Source File. https://clang.llvm.org/doxygen/Basic_2Targets_2WebAssembly_8h_source.html, May 2023. [Accessed 16. May 2023].

[18] Christian Collberg. Home. https://tigress.wtf, May 2023. [Accessed 24. May 2023].

[19] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, 01 1997.

[20] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, dec 2015.

[21] Google Earth. *Google Earth comes to more browsers, thanks to WebAssembly.* Google Earth and Earth Engine, December 2021.

[22] EBay. WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case, May 2019. [Accessed 16. May 2023].

[23] Emscripten. Main — Emscripten 3.1.26-git (dev) documentation. https://emscripten.org, November 2022. [Accessed 1 Dec. 2022].

[24] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction , analysis and simplification tools.* PhD thesis, Université Paris-Saclay, 2017.

[25] Fastly. Fastly Docs. https://docs.fastly.com/products/compute-at-edge, May 2022. [Accessed 23. Nov. 2022].

[26] Figma. Figma is powered by WebAssembly. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x, June 2017. [Accessed 16. May 2023].

[27] The Rust Foundation. WebAssembly. https://www.rust-lang.org/what/wasm, May 2023. [Accessed 16. May 2023].

[28] Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson. Matryoshka: Strengthening software protection via nested virtual machines. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, may 2015.

[29] GitHub. (wasm2c) Re-compiling to WASM. https://github.com/WebAssembly/wabt/issues/1950#issuecomment-1455110508, June 2023. [Accessed 13. Jun. 2023].

[30] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, December 2014.

[31] Google. WebAssembly compilation pipeline · V8. https://v8.dev/docs/wasm-compilation-pipeline, May 2023. [Accessed 16. May 2023].

[32] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., 1977.

[33] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*, WWW '21, page 2696–2708, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 ieee/acm 1st international workshop on software protection*, pages 3–9. IEEE, 2015.

[35] Yuichiro Kanzaki, Clark Thomborson, Akito Monden, and Christian Collberg. Pinpointing and hiding surprising fragments in an obfuscated program. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, dec 2015.

[36] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *The World Wide Web Conference on - WWW '19*. ACM Press, 2019.

[37] Dmitry Kondratyev. The state of cryptojacking in the first three quarters of 2022. *Kaspersky*, November 2022.

[38] Shannon Liao. UNICEF wants you to mine cryptocurrency for charity. *Verge*, April 2018.

[39] Kyeonghwan Lim, Jaemin Jeong, Seong je Cho, Jongmoo Choi, Minkyu Park, Sangchul Han, and Seongtae Jhang. An anti-reverse engineering technique using native code and obfuscator-LLVM for android applications. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. ACM, sep 2017.

[40] Renju Liu, Luis Garcia, and Mani Srivastava. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 94–105. IEEE, 2021.

[41] Aishwarya Lonkar and Siddhesh Chandrayan. The dark side of WebAssembly. *Virus Bulletin*, 2018.

[42] Nils Loose, Felix Mächtle, Claudius Pott, Volodymyr Bezsmertnyi, and Thomas Eisenbarth. Madvex: Instrumentation-based adversarial attacks on machine learning malware detection. In Daniel Gruss, Federico Maggi, Mathias Fischer, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 69–88, Cham, 2023. Springer Nature Switzerland.

[43] Pedro Daniel Rogeiro Lopes. Discovering vulnerabilities in webassembly with code property graphs . *Técnico Lisboa*, 2021. WASMATI (Master thesis/Specialization project).

[44] Gilmore R Lundquist, Vishwath Mohan, and Kevin W Hamlen. Searching for software diversity: attaining artificial diversity through program synthesis. In *Proceedings of the 2016 New Security Paradigms Workshop*, pages 80–91, 2016.

[45] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[46] Anders Møller. Technical perspective: WebAssembly: A quiet revolution of the Web. *Communications of the ACM*, 61(12):106–106, 2018.

[47] Monero. Mining Monero. https://www.getmonero.org/get-started/mining, May 2023. [Accessed 16. May 2023].

[48] Monero. The Monero Project. https://www.getmonero.org, May 2023. [Accessed 16. May 2023].

[49] Jonathon Giffin Monirul Sharif, Andrea Lanzi and Wenke Lee. Impeding malware analysis using conditional code obfuscation. *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[50] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, dec 2007.

[51] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–42. Springer, 2019.

[52] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[53] Faraz Naseem Naseem, Ahmet Aris, Leonardo Babun, Ege Tekiner, and A Selcuk Uluagac. MINOS: A Lightweight Real-Time Cryptojacking Detection System. In *NDSS*, 2021.

[54] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, pages 213–228. Springer, 2002.

[55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[56] Lindsey O'Donnell. Cryptojacking Attack Found on Los Angeles Times Website. *Threatpost*, February 2018.

[57] Daniel Park, Haidar Khan, and Bulent Yener. Generation: Evaluation of adversarial examples for malware obfuscation. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, dec 2019.

[58] Vasile Adrian Bogdan Pop, Seppo Virtanen, Petri Sainio, and Arto Niemi. Secure migration of WebAssembly-based mobile agents between secure enclaves. *Master of Science in Technology Thesis, University of Turku*, 2021.

[59] Jon Porter. Popular 'cryptojacking' service Coinhive will shut down next week. *Verge*, February 2019.

[60] Juan D. Parra Rodriguez and Joachim Posegga. RAPID: Resource and API-Based Detection Against In-Browser Miners. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, dec 2018.

[61] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1574–1589, 2022.

[62] Alan Romano and Weihang Wang. WASim. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, dec 2020.

[63] Alan Romano, Yunhui Zheng, and Weihang Wang. MinerRay: Semantics-aware analysisfor ever-evolving cryptojacking detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, dec 2020.

[64] rustwasm. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen, November 2022. [Accessed 23 Nov. 2022].

[65] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[66] Fabian Scheidl. Valent-Blocks: Scalable high-performance compilation of WebAssembly bytecode for embedded systems. In *2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, pages 119–124. IEEE, 2020.

[67] Stratum. StratumV2. https://stratumprotocol.org, May 2023. [Accessed 18. May 2023].

[68] Anjali J Suresh and Sriram Sankaran. Power profiling and analysis of code obfuscation for embedded devices. In *2020 IEEE 17th India Council International Conference (INDICON)*. IEEE, dec 2020.

[69] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, December 2013.

[70] Justin Thiel. An Overview of Software Performance Analysis Tools and Techniques: From GProf to DTrace. https://www.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1/index.html, May 2023. [Accessed 30. May 2023].

[71] Iain Thomson. Pulitzer-winning website Politifact hacked to mine crypto-coins in browsers. *The Register*, January 2018.

[72] Verus. Verus - Truth and Privacy for All. https://verus.io, May 2023. [Accessed 16. May 2023].

[73] VirusTotal. VirusTotal - Home. https://www.virustotal.com/gui/home/upload, December 2022. [Accessed 2 Dec. 2022].

[74] World Wide Web Consortium (W3C). World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation. https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en, November 2019. [Accessed 16 Nov. 2022].

[75] Shuai Wang, Guixin Ye, Meng Li, Lu Yuan, Zhanyong Tang, Huanting Wang, Wei Wang, Fuwei Wang, Jie Ren, Dingyi Fang, and Zheng Wang. Leveraging WebAssembly for numerical JavaScript code virtualization. *IEEE Access*, 7:182711–182724, 2019.

[76] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks. In *Computer Security*, pages 122–142. Springer International Publishing, 2018.

[77] Wasmer. Wasmer - The Universal WebAssembly Runtime. https://wasmer.io, May 2023. [Accessed 16. May 2023].

[78] Ethereum WebAssembly. Ethereum WebAssembly (ewasm) - Ethereum WebAssembly. https://ewasm.readthedocs.io/en/mkdocs, January 2021. [Accessed 7. Nov. 2022].

[79] Chris Williams. UK ICO, USCourts.gov... Thousands of websites hijacked by hidden crypto-mining code after popular plugin pwned. *The Register*, February 2018.

[80] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys*, 50(3):1–40, jun 2017.